Language Support for Event-Based Debugging

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

Abstract
An event-based debugging framework provides high level facilities for debuggers that observe, monitor, control, and change the state and behavior of a buggy program. This paper introduces a set of additions to the Unicon programming language that enables debuggers to be written at a high level of abstraction. The extensions provide in-process debugging support with simple communication and no intrusion on the buggy program space. These language extensions have been tested and refined within a multi-agent debugging architecture called IDEA, and an extensible source-level debugger called UDB.

1. Introduction
Unicon [7] is an object-oriented dialect of Icon [3], a very high level programming language with dynamic and polymorphic structure types, along with generators and goal-directed evaluation. Historically, the Icon language community had no formal debugging tool, only a trace facility. A rationale for this was that the very high level nature of Icon reduces the need for conventional debugging because programs are shorter. However, Unicon programs are often much larger than was common for Icon, and very high level languages’ advanced features may introduce special kinds of bugs and create special needs for debugging tools.

Icon and Unicon are supported by the Alamo monitoring framework [6,8,9]. Alamo was originally designed for passive observation of program execution, suitable for software visualization tools but not adequate for a source-level debugger. This paper presents recent extensions that enable Alamo to serve as a debugging framework. The result of these extensions is called AlamoDE (Alamo Debug Enabled). AlamoDE is a framework that 1) includes debugging-oriented virtual machine instrumentation, 2) supports additional execution state inspection and source code navigation, 3) provides debugging tools with the ability to change the execution state by safely assigning to a buggy program’s variables and procedures, and most importantly, 4) facilitates on the fly debugging extensions and cooperation. Different AlamoDE-based debugging tools can be written and tested as standalone programs and then loaded into a debugger to work in concert with each other.

This paper presents the design and implementation of AlamoDE and shows the novelty of its debugging support. Section 2 introduces Alamo’s features that are ideal for debugging needs. Section 3 discusses AlamoDE, and its support for classical and advanced debugging techniques. Section 4 describes technical issues and features added, in both the Unicon virtual machine and the Alamo framework, to implement different functionalities for debugging reasons. Section 5 provides an evaluation, and introduces a source-level debugger that utilizes AlamoDE’s features. Section 6 compares Unicon’s debugging support with related work. Section 7 highlights planned future work. The conclusion from our experiment is in Section 8.

2. Background
Alamo is a lightweight architecture for monitoring developed originally to support program visualization. Alamo provides the Unicon virtual machine with monitoring facilities, and the capability for a program to load another program and execute it in a controlled environment.

The Alamo architecture is based on the thread model of execution monitoring, where a monitor program and the monitored program are in separate threads in a shared address space. Unicon’s threads are called co-expressions. A co-expression is a synchronous thread inside the virtual machine. The evaluation of a co-expression requires both an interpreter stack and a C stack that are separate from the stacks of the main program. Alamo extended the Unicon co-expression facility with the ability to load a program, and sets it up with its own code, static data, stack, and heap, without linking symbols into the current program. This execution model provides no intrusion on the monitored program space, which is ideal for classical debugging, and simplifies the process of extending a debugger with external standalone debugging and visualization tools.

Switching between any two co-expressions is done through a small piece of assembler code that performs a lightweight context switch. The state of the program is saved and the control is transferred into the other program without the involvement of the operating system. Because they are synchronous, co-expression switches are much faster than typical thread switches such as those provided by the pthreads library.
3. AlamoDE

Many debugging architectures are based on inter-process communication which is good for remote debugging, but imposes an extra layer of operating system overhead in other contexts. Based on Alamo, AlamoDE provides an in-process debugging architecture with modest communication overhead. AlamoDE debugging support is provided through execution events and high level built-in functions that allow a debugging tool to inspect, change, observe, analyze, and control the state and behavior of the buggy program.

AlamoDE’s goals include: 1) the ability to write debugging tools at a high level of abstraction, 2) all the usual capabilities of classical debuggers, 3) support for the creation of advanced debugging features such as automatic debugging, and dynamic analysis techniques, 4) the ability to debug novel language features such as generators, goal-directed evaluation, and string scanning, and 5) extensibility that allows different standalone debugging tools to work in concert with each other.

3.1. Debugging Events

Considering the many millions of events produced by Alamo’s detailed VM instrumentation, which provides 118 kinds of events, an efficient filtering mechanism is needed. Alamo used a simple bit vector called an event mask to specify event types of interest. For AlamoDE, the filtering was extended so that each event type of interest could have an associated value mask, a set of event values of interest which further restricts whether an event is reported; see Figure 1. Both the event mask and value mask are dynamic. This allows a debugging tool to change and customize the monitored events on the fly during the course of execution; any change on either of the two masks will immediately change the set of prospective events.

For example, placing a breakpoint on one or more line numbers requires the E_Line event to be part of the event mask. The value mask provides the ability to limit the reported E_Line events to those line numbers that have breakpoints on them. To clear a breakpoint, a tool removes the line number from the value mask. The E_Line event is removed from the event mask only if there are no more breakpoints and no other requests for E_Line events by the debugging core or any of its cooperative tools.

Even though debugging and visualization serve many common goals, for AlamoDE, the underlying instrumentation was extended with two additional event types that are needed for debugging. The new events are: 1) E_Deref reports when a variable is read (dereferenced). This event is needed to implement watchpoints on specific variable(s), and 2) E_Syntax reports when a major syntax construct such as a loop starts or ends. This event was inspired by the needs of automatic debugging systems [1,2] and required that syntax information be added to the

Figure 1. AlamoDE Architecture

Unicon virtual machine bytecode architecture. See Section 4.3 for more syntax instrumentation details.

3.2. Debugging Functions

The Unicon language provides some reserved global names prefixed with ampersand (&) called keywords. Some keywords are introduced by Alamo for monitoring needs. For example, &eventsource contains a reference to the currently monitored program. Other keywords are used for error reporting and debugging. For example, the keywords &file, &line, &column, and &syntax report the currently executed file name, line number, column number, and syntax name respectively. These keywords can be inserted directly in the source code of the buggy program for debugging with print statements and assertions.

These keywords are made accessible to debugging tools using the keyword() built-in function, which is used to look up a value of a keyword in the buggy program. For example, keyword("&file", &eventsource, 0) returns the name of the source file that contains the call statement, which instantiated the activation record currently at the top of the buggy program’s call stack. Similarly, keyword("&line", &eventsource, 5) looks up the buggy program’s call stack, and returns the line number of the statement for which the fifth outer most activation record was instantiated.

Likewise, AlamoDE utilizes a set of functions, some of which are needed for observing, inspecting, changing, and controlling the state of the buggy program, while others are needed to support the ability to employ and incorporate external standalone debugging tools into another tool. For example, EvInit() loads and initializes the buggy program under the debugging tool, EvGet() starts/resumes the execution of the buggy program and installs/changes the set of requested events, and EvSend() forwards an event’s code and value from one debugging tool into another; this is mostly used to send the most recent event to an external debugging tool. See Figures 2 and 3.
3.3. Controlling the Buggy Program

In AlamoDE, a debugging tool runs as the main co-expression inside the virtual machine. A buggy program and secondary standalone debugging tools can be loaded into different co-expressions controlled by the debugger. A debugger transfers control to the buggy program using the EvGet() function, which performs a lightweight context switch. After a context switch, the buggy program executes at full speed until there is some event that is of interest to the debugger. Instrumentation in the virtual machine reports an interesting event in the buggy program execution to the debugger by performing a context switch. The return value from EvGet() is an event code; each of which includes an event value that describes its details.

```
while event := EvGet(eventmask, valuemask) do {
  case event of {
    E_Line : { /* handle breakpoints, stepping, etc */ }
    E_Assign | E_Value : { /* Handle assignment watchpoints */ }
    E_Deref | E_Spos | E_Snew : { /* Handle string scanning environments */
    E_Error : { /* Handle a runtime error */
    E_Exit : { /* Handle buggy program normal exit */
  }
  /* Handle other debugging features such as tracing,
    profiling and internal and external debugging tools */
```

Figure 3. An example of AlamoDE debugging loop

EvGet() requests the next event by resuming the buggy program that is denoted by &eventsource. A debugging tool can debug multiple buggy programs in one session. This can be used to perform advanced debugging techniques such as relative debugging [16] or delta debugging [17]. Switching between different programs is accomplished by changing the value of &eventsource before the next call to EvGet(). Furthermore, since different loaded programs are independent in their execution state, this architecture allows different debugging tools to be loaded under each other. It is possible for a debugging tool to debug another tool that is debugging the buggy program, or for multiple debugging tools to simultaneously debug the same buggy program during the same session and same run, see Figure 2.

3.4. Execution State Inspection/Modification

AlamoDE provides facilities to inspect the execution stack, check a variable state, and acquire information about the source code of the buggy program. Furthermore, it provides the ability to control and change the state of the buggy program by assigning to variables and redirecting procedures and functions.

3.4.1. Variable State. A variable is either global, or local including static and parameter variables. A local variable value can be obtained using the built-in function variable(name, &eventsource, level), which returns the current value of the variable name in the frame number level of the buggy program’s call stack. If name is a global variable or a keyword, the same function is used without the level parameter (i.e. variable(name, &eventsource)).

The variable() function is also used to assist a debugging tool in assigning to variables in the buggy program space. This mechanism introduces a potential safety problem if a context switch to the buggy program occurs between the time the variable reference is obtained and the time the assignment is complete. This problem is called inter-program variable safety, and it is solved by implementing a trapped variable technique, see Section 4.2.

3.4.2. Stack Frames. Activation records (frames) on the stack are distinguished by a positive integer called level; the most recent stack frame is at level zero, whereas the highest level value is for the activation record of procedure main(). The proc() built-in function was extended for AlamoDE to allow the debugging tool to identify which procedure is currently active on a specific stack level. For example, proc(&eventsource, 7) returns a pointer to the procedure/method, which lives on the seventh outer most level of the buggy program’s call stack. The depth of the call stack can be checked using the keyword &level. The keyword("&level", &eventsource) returns the number of frames currently on the buggy program’s interpreter stack.

Furthermore, the Unicon language allows programmers to replace a procedure with another procedure during the execution. This feature is very useful for some debugging tools. For example, if the buggy program contains two versions of a sorting algorithm, in different procedures, the debugger can replace one by the other on the fly during the execution.
The procedure/method pointer obtained by the proc() function allows a debugging tool to place a call to that procedure as an *inter-program procedure call*. This mechanism is very useful for interactive source-level debuggers. For example, the buggy program may contain a procedure that prints the elements of a linked list, which is being debugged by the user. The debugger can place a call to that procedure, from any point during the debugging session, without modifying the buggy program source code. Moreover, a source-level debugger may incorporate general service procedures that can be plugged in to the buggy program source code on the fly during the debugging session. For example, the following assignment

```
variable("sort1",&eventsoruce):=proc("qsort",&current)
```

replaces the buggy program’s procedure sort1() with the debugger service procedure qsort(). Of course, the two procedures’ formal parameters must be compatible in their order and type.

### 3.4.3. Executed Source Code

Unicon’s executable bytecode contains information about the linked source files including any used library modules. For AlamoDE, a class library was developed to analyze the bytecode and generate a list of its source file names, and their static source code properties such as packages, classes, global variables, and user defined functions. Another class library maintains a list of all source files in use. Those library classes provide a debugging tool with the buggy program’s source code static information.

Furthermore, the debugging tool can inspect the currently executed source code. Using functions such as keyword() as discussed in Section 3.2, and by monitoring runtime events such as E_Line, and E_Syntax that report the currently executed line number, and source code syntax construct respectively.

### 3.5. Advanced Debugging Support

AlamoDE provides underlying infrastructure for automatic debugging, dynamic analysis, profiling, and visualization. It utilizes many kinds of execution events that cover a wide range of execution behaviors, and gives the user the opportunity to try new debugging techniques. AlamoDE puts execution events in the hands of programmers, who can use events, event sequences, and event patterns to write their own automated debugging and dynamic analysis tools.

#### 3.5.1. Loading Secondary Debugging tools on the Fly

AlamoDE allows advanced debugging techniques to coexist along with a classical debugger. It provides an on the fly extension mechanism. The load() function allows a debugger to incorporate external (secondary) standalone debugging tools under its control. The debugger coordinates and plays the role of a central server for other debugging tools. The debugger and its loaded tools work synchronously on the same buggy program. For example, standalone debugging tools can be loaded on the fly during a source-level debugging session; without previous initialization. Loaded tools receive execution events from the host debugger, which multiplexes event codes and values using the function EvSend(), see Figure 2. Events are sent based on each tool event mask. An event is sent only to the tools that request it in their event mask. For example, the code in Figure 4 shows a toy example of an AlamoDE-based debugging tool. It captures the number of garbage collections that happened during the execution of the buggy program, and finds the total and average of collected data from the string and block regions. This provides a rough measure whether the buggy program is mostly doing string processing or not. This example program can be used as standalone tool, or loaded into another debugging tool on the fly without any source code modification at all.

```
#include "evdefs.icn"
link event

class Example ( eventMask, gc, lastStr, lastBlk, collectedStr, collectedBlk, avgStr, avgBlk )
method handle_E_Collect() 
    local Storage := []
    gc := gc + 1
    every put(Storage, keyword("storage", Monitored))
    lastStr := Storage[2]; lastBlk := Storage[3]
end
method handle_E_EndCollect() 
    local Storage := []
    every put(Storage, keyword("storage", Monitored))
    collectedStr := lastStr - Storage[2]
    collectedBlk := lastBlk - Storage[3]
end
method analyze_data() 
    if gc > 0 then return 0
    avgStr := collectedStr / gc; avgBlk := collectedBlk / gc
end
method write_data() 
    write(" # Garbage Collections : ", gc)
    write(" Collected Strings : ", collectedStr, " Avg: ", avgStr)
end
initially() 
    eventMask := cset(E_Collect || E_EndCollect)
    gc := 0; collectedStr := collectedBlk := 0.0
end
procedure main(arg)
    EvInit(argv)
    obj := Example()
    while event := EvGet(obj.eventMask) do{
        case event of {
            E_Collect: { obj.handle_E_Collect() }
            E_EndCollect: { obj.handle_E_EndCollect() }
        }
        obj.analyze_data()
        obj.write_data()
    }
end
```

**Figure 4.** An AlamoDE debugging agent.
4. Implementation

This section provides an overview of the implementation of the most important underlying extensions. Some of the extensions are general additions to the Unicon virtual machine and its runtime system, while the rest are extensions to the Alamo monitoring framework.

4.1. Virtual Machine Instrumentation

Event-based debugging support needs instrumentation, which can be inserted into the source code, the bytecode, or implicit in the virtual machine itself. Implicit instrumentation is attractive to the end user, who needs a simple mechanism of getting events. However, instrumentation is always associated with overhead in space and processing time. Unicon has a small virtual machine (about 700KB with the instrumentation). A top priority for Unicon’s implicit instrumentation is the processing time, which should not be affected for any unmonitored execution.

Originally, Alamo was an optional extension to the Icon virtual machine, because Alamo’s instrumentation imposed a cost even when monitoring was not being performed. Alamo was integrated in the Unicon language with no measurable cost (other than code size) in the production virtual machine. This integration allows the debugger to run on the virtual machine synchronously along with the buggy program. The debugger and the buggy program run in two different co-expressions and the buggy program is the only one affected by the instrumentation.

AlamoDE maintains two versions of 30 runtime functions in the binary executable VM that contain instrumentation. One version is uninstrumented and used in any unmonitored execution; the other version is instrumented and used when a program is monitored. Furthermore, not all of the instrumented functions are used when the program is under monitoring; a dynamic binding associates the instrumented or uninstrumented function with the current execution state based on the current event mask, which is specified by the debugger. Table maps event codes into their instrumented functions. Whenever an event is added to the monitored events (event mask), the related instrumented function is used. If an event is removed from the event mask, the original uninstrumented version of the function is restored.

Inside the Unicon virtual machine source code, the name of the instrumented function uses the suffix “_1”, whereas the name of the uninstrumented version of the same function uses the suffix “_0”. Functions that contain instrumentation use macros to maintain one copy of the source code, which simplifies the maintenance effort.

Using this method of dynamic binding, the instrumentation imposes no cost on the execution time of the virtual machine until the program is debugged or monitored, and the only instrumented functions used are the ones relevant to the currently monitored events, which are specified by the event mask and customized by the debugging tool and the programmer.

4.2. Inter-Program Variable Safety

In order for a debugging tool to be able to change the value of a variable inside a buggy program, the tool must have access to the state of the buggy program. However, the debugging tool and its buggy programs are loaded into different co-expressions inside the Unicon’s virtual machine.

Giving a co-expression the ability to access and change another co-expression state is critical. It is possible for one of the co-expressions to obtain a reference for a variable that is either in the stack, in the static data section, or in the heap of the other co-expression. However, while the first co-expression is trying to change a variable in the second co-expression, a context switch may allow control to be transferred to the second co-expression. A memory violation might occur if the second co-expression executes further while the first co-expression has a reference to a local variable; a reference to a variable that lives on the stack might become invalid. For example this can happen if the procedure returned and its activation record is popped off the stack. Since co-expressions are synchronous this is admittedly an unlikely occurrence that would only be caused by a deliberate adversary.

The implemented solution is a trapped variable technique [4]. Whenever one co-expression obtains a reference to the state of another co-expression, a trapped variable is used and the value is stored in a different memory space. The value is then transferred back to the co-expression that originally obtained the reference to the state of another co-expression.
variable block is allocated and the reference is done through that trapped variable, see Figure 5. The first co-expression contains a reference to a trapped variable block, which references the actual variable in the second co-expression. This new block holds information about the current number of context switches between the two co-expressions. This number is compared to the very recent one just before writing to that variable. If there is any difference between the number of context switches when the reference was obtained and when the reference is written, then this technique invalidates the assignment and gives the user a runtime error. This can only happen if a context switch occurs in the middle of an assignment to a monitored trapped variable. This new technique produces a runtime error where a monitor deliberately invokes the subject program in this way. This critical section can occur inside an Alamo monitor in unlikely scenarios such as:

1(variable("x", &eventsource, 1), EvGet()) := 5

It is possible this way to call EvGet() and transfer control to the buggy program between the variable reference and its assignment, but it is not easy. Not surprisingly, the code for a normal debugger does not do any such thing. The safety feature was added to the language to extend the variable() function to produce references to local variables while a program is paused.

4.3. Syntax Instrumentation

Unicon’s bytecode executes as a sequence of virtual machine instructions. Like most binary code formats, the bytecode formerly contained no information about the actual syntax of the source code. However, some automatic debugging facilities need information about the syntax of the running program. For example, an automated debugging technique that locates frequently failed loops needs to know when the execution of the buggy program enters and leaves a loop and what type of loop it is; such as a while loop.

The solution is to add a new pseudo instruction that is managed by the translator and the linker. The new OpSyntax pseudo instruction extends the "line#/column#" table with information about the syntax. It is a reasonable solution because the only cost is a small increase in the size of the table. The cost of retrieving the syntax information from the table is paid for only when a program is monitored and that information is needed.

The "line#/column#" table was transformed into a "line#/column#syntax" table without altering its design, see Figure 6. The table entry is a 32-bit integer: the most 16 significant bits were for the column number and the least 16 significant bits were for the line number. The maximum possible line/column number is 65535, which is more than is needed for a column number. We changed the column number bits to be the 11 most significant bits, and the remaining 5 bits are used for syntax information. The new design changed the maximum possible column number to 2048, which is still more than enough for a column number. The newly added pseudo instruction only appears in the object files and is used by the linker while generating the bytecode. The new 5-bit syntax code can hold up to 32 different syntax indicators. AlamoDE presents syntax information as a new selectable event code E_Syntax and its related event value is the syntax code, see Table 1. A library routine was written to translate the syntax code into its symbolic name.

<table>
<thead>
<tr>
<th>Syntax</th>
<th>Integer Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>unidentified syntax</td>
<td>0</td>
</tr>
<tr>
<td>entering case expression</td>
<td>1</td>
</tr>
<tr>
<td>exiting case expression</td>
<td>2</td>
</tr>
<tr>
<td>entering if expression</td>
<td>3</td>
</tr>
<tr>
<td>exiting if expression</td>
<td>4</td>
</tr>
<tr>
<td>entering if/else expression</td>
<td>5</td>
</tr>
<tr>
<td>exiting if/else expression</td>
<td>6</td>
</tr>
<tr>
<td>entering while loop</td>
<td>7</td>
</tr>
<tr>
<td>exiting while loop</td>
<td>8</td>
</tr>
<tr>
<td>entering every loop</td>
<td>9</td>
</tr>
<tr>
<td>exiting every loop</td>
<td>10</td>
</tr>
<tr>
<td>entering until loop</td>
<td>11</td>
</tr>
<tr>
<td>exiting until loop</td>
<td>12</td>
</tr>
<tr>
<td>entering repeat loop</td>
<td>13</td>
</tr>
<tr>
<td>exiting repeat loop</td>
<td>14</td>
</tr>
<tr>
<td>entering suspend loop</td>
<td>15</td>
</tr>
<tr>
<td>exiting suspend loop</td>
<td>16</td>
</tr>
</tbody>
</table>

5. Evaluation

An AlamoDE-based source-level 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. The AlamoDE debugging framework provides
high level facilities to customize and reduce the amount of monitored events and context switches.

AlamoDE was used to build an extensible source-level debugger called UDB [18], which integrates new automatic detection techniques that can be found in trace-based debuggers such as ODB [11,14]. One measurement of the effectiveness of the AlamoDE is that UDB’s source code is less than 8K lines of source code. Furthermore, UDB provides two types of extension agents, externals (secondary debugging tools) and internals, supported by IDEA. IDEA is the Idaho Debugging Extension Architecture, a multi-agent debugging architecture built on top of AlamoDE and used by UDB. Each extension agent is a task-oriented program execution monitor. Debugging agents are standalone tools, which can be loaded on the fly during a UDB debugging session, or incorporated into its debugging core as permanent debugging features. Moreover, UDB’s debugging agents can be enabled and disabled from any point during the debugging session and the user can be selective about which suite of agents to use.

Under UDB, eight different debugging agents were loaded and tested as external agents, and then adopted 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 monitoring 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 [15] imposes a 20 to 50 times slowdown, and it does not provide the interactive debugging environment that AlamoDE-based debugging tools provide, where the user can be selective about which and where to enable/disable agents from within a breakpoint based debugging session.

6. Related Work

Programming languages vary widely in their debugging support. Python provides debugging techniques through a module (PDB) [14], which maintains the classical debugging techniques such as breakpoints, stepping and continuing. It also supports post-mortem debugging and it can be called under program control. However, since PDB is a module, it must be imported into the Python program in order to be used. PDB was not designed with automatic debugging or extension particularly in mind, but the interpretive nature and high level of Python make it a good candidate for research experimentation. PDB’s module architecture suggests that PDB perturbs application behavior such as garbage collection due to a shared heap.

The Smalltalk system includes very important tools such as a browser, workspace, debugger, and inspector. These tools provide a complete development and testing environment system that assist in the edit-compile-link-run-debug cycle. All Smalltalk objects understand special messages such as doesNotUnderstand and inspect. The doesNotUnderstand message is produced automatically by Smalltalk runtime system as a result of a runtime error. This message causes the Smalltalk system to provide the user with an error notification, which asks the user if he/she is interested in a debugging session. During a debugging session, the programmer is able to modify the program while it is running. In general, Smalltalk runtime errors cause the execution thread to be suspended. In some cases the runtime error can be recoverable. The user may fix the error and continue in the execution. This simplifies the process of reproducing the bug. In fact, it is fairly common for programmers to struggle with this process. In contrast, the inspect message is produced and sent intentionally by the programmer; it allows a user to inspect an object through the inspector window [10]. SmallTalk’s debugger has several similarities and important differences compared with UDB. The most important similarity is that both use a thread model of execution, which provides relatively good, high performance access to program state. Another similarity is that most of the debugger is written in the same language as the program that is being debugged. Compared with UDB, IDEA, and AlamoDE, SmallTalk’s debugger is less separate from the program being debugged, and relies more on manual instrumentation via subclassing and overriding methods to generate events for dynamic analysis.

A debugging architecture such JPDA, with its latest lowest level JVM TI [5], 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 AlamoDE incorporates more than one hundred kinds of events. Unicon programmers use events, event-sequences, and event-patterns to write their own debugging agents that detect specific execution behaviors—which some of they are suspicious behaviors while others are defined bugs. Furthermore, JVM TI is based on the inter-process communication for less intrusive on the buggy application. Unicon debugging support features the in-process communication with no intrusion on the buggy application.

Both AlamoDE 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 different AlamoDE-based standalone debugging tools can work in concert with each other during a debugging session, or be incorporated into the debugger source code as permanent extensions, with little or no source code modification.

7. Future Work

Unicon’s classical debugging support for features, such as breakpoints and watchpoints, is provided through monitored events and event filtering. Even though these
techniques perform well during debugging, improving their performance can be achieved by further implementation of common techniques such as trapped virtual machine instruction for breakpoints, and trapped variable for watchpoints. Moreover, AlamoDE’s support for multiple simultaneous debugging tools can benefit from extending Unix support with real concurrency, where different co-expressions can be off loaded onto different core processors.

AlamoDE’s performance can be improved further by reducing the number of context switches. At present the monitor coordinator plays the role of a central server in a star network. A ring-based architecture where each monitor forwards events to another monitor instead of having a central coordinator would reduce context switches by 50%. Another possible architecture is a broadcasting mechanism where the buggy program broadcasts events to all secondary debugging tools. Furthermore, extending AlamoDE for other languages can benefit from existing instrumentation frameworks such as ASM for Java and PIN and ATOM for C and C++ programs.

8. Conclusion

Unicon’s debugging support provides programmers with high level built-in functions and execution events that allow them to go beyond the standard debugging techniques. AlamoDE is designed to simplify automatic debugging, and dynamic analysis techniques. It allows various analyses to be used as standalone tools, dynamically loaded into a debugging coordinator with no source code alteration, or permanently incorporated into the debugging tool with minimal modifications. Programmers can utilize execution events, event patterns, and event sequences to capture specific behaviors. Some may be considered suspicious behaviors while others are classified as bugs. The implementation of UDB proves that the AlamoDE framework is powerful enough to reduce the development cost of source-level debuggers and simplify its maintenance and extension. Previous event-based source-level debuggers, such as Dalek [12,13] identified performance obstacles. With this approach, AlamoDE provides usable debugging support proved in the implementation of UDB and its multi-agent debugging extension architecture.

9. Acknowledgment

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.

10. References


