Scalable Source Code Debugger Agent to Provide Program Debugging in a Script Testing Environment

Jun Xu, Christopher H. Pham, Joe Zhou
ARF, Cisco Systems Inc., San Jose, CA, USA, November 2002
{junxu, chpham, joezhou}@cisco.com

Abstract -- **Sometimes when a test script triggers a failure, one needs a mechanism to debug the application at the source code level. This paper presents the algorithm and case studies of a Debugger Agent (DA) to prove the feasibility to provide a scalable solution on most complex test systems. The DA is independent of the underlying debuggers and the script testing systems. The DA bridges the traffic among the script system/engine and the UUTs (Unit Under Test), and acts transparently while there are no application failures. The DA suspends the first tier scripts driving testing when there are application failures, and could perform automatic debugging from second tier scripts. The DA does not require code instrumentation or further test script development.**

I. THE ISSUE

While there are many tools available to debug the program or script, there is no known commercially available tool to invoke the debugger when a bug-free test script triggers a failure in the UUT's program to allow the developer to perform familiar run-time program debugging.

In addition, many flavors of scripts may target the same UUT. For example, in a multi-host setup, some scripts may be written in Perl, Tcl/Tk or Expect running on Unix platform while some others are written in other script languages in other platforms such as Windows. The program itself may be written in different languages such as C, C++, etc.. A generic source code level program debugging mechanism is therefore needed for scalability to serve a large development and testing scale.

II. OPERATIONS

Typical test setups are portrayed in Figures 3, 4 and 5. Figure 3 is the extreme case consisting multiple computers hosting the test scripts, multiple UUTs, and a Test Script Handler (TSH) acting as the middle agent to direct the traffic between the test hosts and the UUTs. Figure 4 shows only one test host and one UUT with the absence of the TSH. The test script execution is scheduled by the OS on the test host. The Debugger Agent (DA) resides on the test host. Figure 5 shows the simplest configuration where the UUT (i.e. an application) also stays on the same computer hosting the test scripts and the DA. This setup is very common for developers who develop applications or hardware drivers for the same machine, then launch a test script (or batch file) to test out the application. Algorithm is shown in Figure 1 below.

(If the Test Script Handler (TSH) is not used, i.e. the scripts just run in batch mode, then a wrapper can be used to insert the system instructions that supposed to be issued by TSH).

1. Begin
2. Initiate CPU dependent debugger (i.e. GDB) for each UUT
3. Debugger loads symbol file for each UUT
4. Test Script Handler (TSH) instructs debugger to attach to UUT
5. TSH sends further commands to set up debugger (ex: CPU exception, set break points, watched events, etc.)
6. TSH starts regular script execution
7. Watched event occurs? If No goto next state, else goto state 14
8. Script done? If NO goto next state, else goto state 10
9. Continue Script execution, back to state 6
10. All scripts executed? If NO goto next state, else goto state 20 (END)
11. TSH loads next script
12. Clean up debugger
13. Goto step 6
14. TSH suspends UUT application
15. TSH invokes debugger
16. Auto debug mode? If Yes goto next state, else goto state 19
17. TSH sends cmds to debugger & obtain local frame info, stack trace, variables values, etc.
18. Finish debug, out of debug mode, TSH resume script, then goto state 8
19. User manually sends debug cmds, then goto state 18
20. END

Figure 1. Algorithm to attach the Debugger Agent to the existing complex system test

III. CASE STUDY

The authors decided to use GDB as the debugger due to its universality in Unix and Windows platforms for many different development languages. Details on GDB can be found in [REFERENCES]. GDB is stored in a central server. A few points to take are: CPU-specific GDB must be used for a specific computer, UUT's binary specific and symbol file must be loaded for each UUT so GDB can refer to the proper stack trace and variables. To demonstrate the ability of the proposed techniques, the most complex setup as in Figure 3 is chosen.

The DA, now is the GDB Agent, is integrated into the existing TSH. The TSH automates the script executions and controls the traffic among the hosts to the UUTs. The test machines are both Solaris servers and Linux boxes. The UUTs are the Cisco routers running on top of its proprietary IOS (Internet Operating Systems) and on top of different CPUs (MIPS, Intel, ARM, etc.). Upon booting, the DA issues the system commands to retrieve the CPU specific GDB from the central server, based on CPU type of the UUT. The corresponding binary and symbol files are loaded for that UUT. Similar procedure is done for all UUTs. Also debugger feature is disabled by default to have no performance impact. The debugger is only invoked by the DA

Copyright 2002 Chillarege Press

when a failure occurs. The DA itself is not contributing to the program application so it does not impact the program performance. The only minimal impact is introduced by the DA acting as the middle agent between the test scripts and the program. For most complex system, this is the same as the script scheduler which contribute to the testing time, not the application time. When the debugger needs to be turned on, a flag can be set to change the UUT into the debug monitoring mode. All the above were managed by the DA.

When the debug monitoring mode is active, the process will run in the background to monitor the exception from the UUT. When a certain script triggers a failure on the UUT, the exception is caught. The DA suspends the operation and places the UUT into the debug mode. Those who have used the GDB will see the familiar debug information dump such as in Figure 4. After debugging the application from the UUT side, the DA will allow the test script to continue the execution from the suspended location.

IV. LIMITATIONS/ROOM FOR IMPROVEMENT

Several limitations and room for improvement were observed.

1. While the DA does not require the symbolic information, the debugger itself may require access to symbol files.
2. The DA neither facilitates script debugging nor requires any script changes.
3. The algorithm in Figure 1 does not have the ability to support the multiprocess systems. The mechanism can be easily expanded to attach the debugger (i.e. the GDB) to each child process.
4. For homogeneous environment, it is possible to use shared library to attach the debugger to all processes. For the heterogeneous systems, in remote debugging case, each process may need to be treated separately. Expansion of the algorithm in Figure 1 is therefore needed.
5. There are performance impact due to the breakpoints setup. It is normally relatively low when the breakpoints are not hit. In any cases, the impact is not from the DA.

V. CONCLUSIONS

A mechanism to allow application source code debugging triggered by system level test scripts has been presented. It is worth the investment to add in a mechanism such as the DA to gain the controllability and observability at the source code level while compromising very low performance or code investment of the existing test setup. It provides the accessibility to application developers at the system level (which translates into field level), so it helps to reduce the service time, thus increase the system up time.

```
UUT_1(config-fr-dlci)#ip rip triggered
UUT_1(config-if)#no shut
Program received signal SIGBUS, Bus error.
(cpu-mips4k-gdb) info frame
Stack level 1, frame at 0x6281aa88:
 pc = 0x604102ec in function1 (../dir1/file1.c:2101);
Saved registers:
  s0 at 0x6281aa70, s1 at 0x6281aa74, s2 at 0x6281aa78, s3 at
0x6281aa7c,
  s4 at 0x6281aa80, ra at 0x6281aa84, pc at 0x6281aa84
(cpu-mips4k-gdb) info locals
vcinfo = (vc_info_t *) 0x1
vcd = 43624
```

Figure 2 - Sample debug screen. Example shows the DA uses the conventional GDB

REFERENCES

[1] Generic GDB with source code http://www.gnu.org/directory/gdb.html,

[2] GDB for Windows http://www.mingw.org/

[3] MinGW: Minimalist GNU For Windows http://www.mingw.org/

[4] Microsoft® Visual Studio™ - Just In-Time Debugging http://msdn.microsoft.com/library/default.asp?url=/library/en-us/vidref98/html/vimsgjust-in-timedebugging.asp

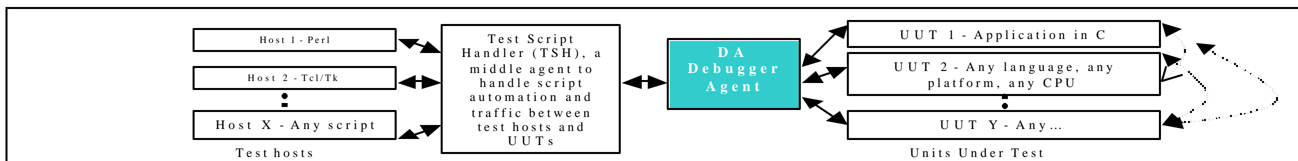

Figure 3 – Extreme case with multiple test hosts and multiple UUTs sharing the same automated system test environment. The Add-in Debugger Agent facilitates the invocation of the debugger on the UUT Y when its failure is triggered by the script from Hostx

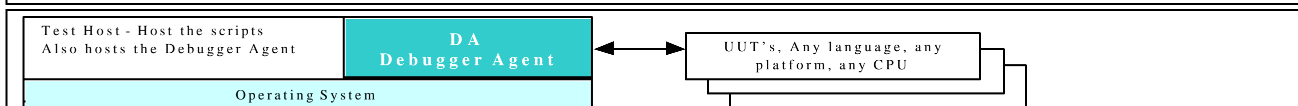

Figure 4 - Only one test host and one UUT with the absence of the Test Script Handler. The test script execution is scheduled by the OS on the same host. The Debugger Agent is also residing on the test host.

Figure 5 - The simplest configuration where the UUT (i.e. an application) also stays on the same computer, which hosts the test scripts and the Debugger Agent.

Copyright 2002 Chillarege Press