Some Axioms and Issues in the UFO Dynamic Analysis Framework

Clinton Jeffery
Department of Computer Science
New Mexico State University
jeffery@cs.nmsu.edu

Mikhail Auguston
Department of Computer Science
Naval Postgraduate School
maugusto@nps.navy.mil

Abstract

UFO is a framework for constructing dynamic analysis tools that require varying degrees of access and control over program executions. UFO combines run time and post-mortem techniques to perform required analyses. Declarative and imperative notations are provided for constructing monitors at appropriate semantic levels. Multiple analyses can be bundled into a given monitor, and multiple monitors can be applied to a given target program execution. This paper presents the central tenets of UFO, along with our current set of research challenges.

1. Motivation

Automatic debugging and program visualization are two of the most promising application areas of dynamic analysis, with potential to impact on crucial areas of software development and maintenance. We believe the slow rate of advancement in these areas is due to the high cost of developing new tools. We have previously focused on a language (FORMAN) and an architecture (Alamo) that reduce these costs [1][2][4]. FORMAN is a special-purpose language for expressing dynamic analyses; it has been implemented previously for subsets of Pascal and C. Alamo is a lightweight architecture for program execution monitoring; it has been implemented for a subset of C and for the virtual machine used by the Icon and Unicon programming languages. The virtual machine implementation of Alamo is attractive for research because it provides high performance and superior ease of use for a full-size "real" programming language, allowing testing on large programs and the possibility of deploying successful tools to a user community.

We recently merged the FORMAN and Alamo efforts to produce UFO (Unicon-FORMAN), a framework for rapidly constructing dynamic analyzers [3][4]. We have used UFO to construct a variety of simple automatic debuggers and visualization tools that run well on small and medium sized applications. Our next efforts must walk the tightrope of scaling up to production tools for large applications, while retaining the power and ease of use that are characteristic of the current research UFO system. With that in mind, this paper presents the central tenets of the UFO system, and concludes with an exploration of the current research problems and our plans to address them.

2. Axioms

UFO is primarily an implementation of FORMAN built on top of the Alamo monitor architecture. Early experiments showed the marriage to improve FORMAN speed by two orders of magnitude and shorten the lines of code necessary to write Alamo monitors by one order of magnitude. This section sketches the primary characteristics of UFO.

- A precise program behavior model, in which semantics of the monitored language are mapped to directed acyclic graphs of events. These graphs are defined using an *event grammar*, a notation that approximates the semantics of the language to be monitored. The behavior model is essential to provide general purpose capabilities for a wide range of tools.
- A declarative special-purpose monitoring language, tailored specifically for dynamic analyses expressed in terms of patterns within the graphs of events. This component is necessary to reduce the cost of developing new tools. Section 4 provides some examples; shorthand refinements to improve the syntax could be explored after the main semantics and performance issues are resolved.
- An hybrid execution model, in which most analysis work is performed at run-time, and more complex analyses transparently combine run-time collection and partial analysis with more extensive post-mortem analysis. This element is necessary but not sufficient by itself to achieve acceptably high performance for large scale production systems. This important element is new in UFO, compared with previous FORMAN and Alamo efforts. It provides high performance.
- Automatic instrumentation provided by special-purpose virtual machine support; static or dynamic configuration of VM instrumentation; no recompilation, relinking, or alteration of target program executables to be monitored. This provides substantial ease of use.

3. Some Research Issues and Challenges

UFO's chief design goals revolve around notational power and ease of use. The current prototype implementation of UFO [5][5] processes millions of events per minute. But, for large programs higher performance is needed. This goal motivates several open problems we are pursuing.

Minimizing the number of context switches. UFO's run-time execution model is based on lightweight coroutine switches between monitors and the program being observed. This separation is a compromise between intrusive in-line single-thread execution used in low-cost analysis tools such as profilers, and the complete separation imposed by high-cost analysis tools such as debuggers. One research goal is to retain the abstraction and low-intrusion benefits of the coroutine model without having to pay (so much) for it.

Virtual machine configuration and customization. The VM instrumentation can be turned off at multiple levels, including compile-time via #ifdef and run-time via a dynamic filter that controls whether instrumented or uninstrumented versions of functions are called, and whether an event report (via lightweight context switch) is performed for a given instrumentation site. This configuration can be further exploited by having the UFO compiler generate a custom VM with exactly the instrumentation it needs for a particular monitoring application. The central VM interpreter function (interp()) can benefit from a finer granularity of customization than the current instrumented-versus-uninstrumented options; it is critical to performance and contains 30 of the 119 types of events instrumented in the VM. Generating a custom VM may greatly improve monitoring performance within this VM interpreter loop. The VM generation system needs to make it easy and convenient for the UFO compiler to generate custom VM's and associate them with generated analyzers in a persistent manner. Custom VM's should be shareable by monitors that use the same events.

Inter-monitor optimizations. When multiple analyses are compiled together, substantial cost savings might be obtained by factoring common tasks such as event data collection. For example, a profiler that computes summaries and a visualizer that shows run-time details might operate on the same information, and might even share some common analysis structures.

Meta-events and analysis hierarchies. UFO's event model composes higher level events from lower level ones, but analysis tools create additional information which may constitute the input for higher level analyses. This facilitates the sharing of analysis information among tools, reducing the cost of running multiple tools.

4. Examples of debugging rules

Alamo's goal was to reduce the difficulty of writing execution monitors to be just as easy as writing other types of application programs. UFO supports FORMAN's more ambitious goal of reducing the difficulty of writing automatic debuggers to the task of specifying generic assertions about program behavior.

This section presents formalizations of typical debugging rules. UFO supports traditional precondition checking, or print statement insertion, without any modification of the target program source code. This is especially valuable when the precondition check or print statement is needed in many locations scattered throughout the code.

Example #1: Tracing. Probably the most common debugging method is to insert output statements to generate trace files, log files, and so forth. It is possible to request evaluation of arbitrary Unicon expressions at the beginning or at the end of events. The virtual machine evaluates these expressions at the indicated time moments.

```
FOREACH  A: func_call &
        A.func_name == "my_func"
     FROM prog_ex
  A.value_at_begin(
     write("entering my_func, value of X is:", X) ) AND
  A.value_at_end(
    write("leaving my_func, value of X is:", X) )
```

This debugging rule causes calls to write() to be evaluated at selected points at run time, just before and after each occurrence of event A.

Example #2: Profiling. A myriad of tools are based on a premise of accumulating the number of times a behavior occurs, or the amount of time spent in a particular activity or section of code. The following debugging rule illustrates such computations over the event trace.

```
SAY( "Total number of read() statements: "
      CARD[ r: input & r.filename == "xx.in"
           FROM prog_ex ]
       "Elapsed time for read operations is: "
       SUM [ r: input & r.filename == "xx.in"
            FROM prog_ex   APPLY r.duration] )
```

Example #3: Pre- and Post- Conditions. Typical use of assertions includes checking pre- and post-conditions of function calls.

```
FOREACH A:func_call & A.func_name=="sqrt"
      FROM prog_ex
   A.paramlist[1] >=0 AND
   abs(A.value*A.value-A.paramlist[1]) < epsilon
WHEN FAILS SAY("bad sqrt(" A.paramlist[1
                    ") yields " A.value)
```

4.1   Generic Bug Descriptions

Another prospect is the development of a suite of generic automated debugging tools that can be used on any Unicon program. UFO provides a level of abstraction sufficient for specifying typical bugs and debugging rules.

Example #4: Detecting Use of Un-initialized Variables. Reading an un-initialized variable is permissible in Unicon, but often leads to errors. In this debugging rule all variables in the target program are checked to ensure that they are initialized before they are used.

```
FOREACH V: variable  FROM prog_ex
       FIND D: lhp FROM V.prev_path
             D.source_text == V.source_text
       WHEN FAILS SAY( " uninitialized variable "
                      V.source_text)
```

Example #5: Empty Pops. Removing an element from an empty list is typical of expressions that fail silently in Unicon. While this can be convenient, it can also be a source of difficult to detect logic errors. This assertion assures that items are not removed from empty lists.

```
FOREACH   a: func_call &
       a.func_name == "pop" AND
       a.value_at_begin( *a.paramlist[1] == 0)
     SAY("Popping from empty list at event " a)
```

5.   Implementation Issues

The most important of these issues is the translation model by which FORMAN assertions are compiled down to Unicon Alamo monitors. Debugging activities are written as if they have the complete post-mortem event trace, the DAG with events, event attributes, and precedence and containment relations, available for processing. This generality is extremely powerful; however, for most practical uses we have seen, assertions can be compiled down into monitors that execute entirely at runtime. Runtime monitoring saves enormously on memory and I/O requirements and is the key to practical implementation. For those assertions that require post-

mortem analysis, the UFO runtime system computes a projection of the execution DAG necessary to perform the analysis.

The UFO compiler generates Alamo Unicon monitors from FORMAN rules. Each FORMAN statement is translated into a combination of initialization, run-time, and post-mortem code. Monitors are executed as coroutines with the Unicon target program.

Monitors generated by the UFO compiler reduce complex assertions to the single event loop. Keeping event detection in a single loop allows uniform processing of multiple event types used by multiple monitors. The code generated by the UFO compiler integrates event detection, attribute collection, and aggregate operation accumulation in the main event loop.

Assertions in UFO may use nested quantifiers implying two nested loops, so code generation addresses this issue by flattening the main loop structure, and postponing assertion processing until required information is available. An hybrid code generation strategy performs runtime processing whenever possible, delaying analyses until post-mortem time when necessary. Different assertions require different degrees of trace projection storage; code responsible for trace projection collection is also arranged within the main loop. The following generation template gives a flavor of the UFO trace projection mechanism.

Rules with two nested quantifiers of the form

```
Quantifier A: Pattern_A
     Quantifier B: Pattern_B FROM A
           Body
```

utilize a monitor whose main loop follows the pattern:

```
Main Loop
     Maintain stack of nested A events
     Accumulate events B in a B-list
     If end of event A
       Loop over B-list
             Do Body
     Endif
     If stack of A is empty
             Destroy B-list
End of Main Loop
```

This requires accumulation of a trace projection for B-events and may cause a mild overhead at the run time.

5.1   Optimization Issues

The UFO approach combines an optimizing compiler for monitoring code with efficient run-time event detection and reporting. Since we know at compile time

all necessary event types and attributes required for a given UFO rule, the generated Unicon monitor can be very selective about the behavior that it observes.

For certain kinds of UFO constructs, such as nested quantifiers, the monitor must accumulate a sizable projection of the complete event trace and postpone corresponding computations until all required information is available. The presence of the previous_path and following_path attributes in UFO rules triggers this kind of optimization; previous_path and following_path are used in rules which specify preceding or following contexts for events of interest.

For further optimization, especially in the case of programs containing a significant number of modules, the following FORMAN construct limits event processing to events generated within the bodies of functions F1, F2, … , Fn.

```
WITHIN F1, F2, … , Fn DO
    Rules
END_WITHIN
```

This provides for monitoring only selected segments of the event trace.

Unicon expressions included in the value_at_begin and value_at_end attributes are evaluated at run time.

Some other optimizations implemented in this version are:

- only attributes explicitly used in the UFO rule are collected in the generated monitor;
- an efficient mechanism for event trace projection management, which disposes from the stored trace projection those events that are no longer used after a certain rule has been fully evaluated;
- both event types and context conditions are used to filter events for the processing.

UFO's goal of practical application to real-sized programs has motivated several improvements to the already carefully-tuned Alamo instrumentation of the Unicon virtual machine. We are working on additional optimizations.

We expect that the most promising optimizations are within the generation of instances of Virtual Machine tailored for a particular monitoring task.

6. Conclusions

The architecture employed in UFO could be adapted for a broad class of languages such as those supported by the Java VM or the .net VM. Our approach to dynamic analysis uniformly represents many types of debugging-related activities as computations over traces, including assertion checking, profiling and performance measurements, and the detection of typical errors. We have integrated event trace computations into a monitoring architecture based on a virtual machine.

Preliminary experiments demonstrate that this architecture is scalable to real-world programs.

One of our next steps is to build a repository of formalized knowledge about typical bugs in the form of UFO rules, and gather experience by applying this collection of assertions to additional real-world applications. There remain many optimizations that can improve the monitor code generated by the UFO compiler; for example, merging common code used by multiple assertions in a single monitor, and generating specialized VMs adjusted to the generated monitor.

References

[1] M. Auguston, Program Behavior Model Based on Event Grammar and its Application for Debugging Automation, in the Proceedings of the 2nd International Workshop on Automated and Algorithmic Debugging, AADEBUG'95, Saint-Malo, France, May 22-24, 1995, pp. 277-291.

[2] Clinton L. Jeffery, Program Monitoring and Visualization: an Exploratory Approach. Springer, New York, 1999.

[3] M. Auguston, A. Gates, M. Lujan, "Defining a Program Behavior Model for Dynamic Analyzers", in the Proceedings of the 9th International Conference on Software Engineering and Knowledge Engineering, SEKE'97, Madrid, Spain, June 1997, pp. 257-262.

[4] M. Auguston, "Lightweight semantics models for program testing and debugging automation", in Proceedings of the 7th Monterey Workshop on "Modeling Software System Structures in a Fast Moving Scenario", Santa Margherita Ligure, Italy, June 13-16, 2000, pp.23-31.

[5] M. Auguston, C. Jeffery, and S. Underwood. "A Framework for Automatic Debugging", IEEE 17th Intl. Conf. on Automated Software Engineering, Edinburgh, September 2002, IEEE Computer Society Press, pp.217-222

[6] C. Jeffery and M. Auguston. "Towards Fully Automatic Execution Monitoring". Monterey Workshop 2002, Venice, October 2002, sponsored by US Army Research Office and NSF, pp.232-243

[7] Clinton Jeffery, Shamim Mohamed, Ray Pereda, and Robert Parlett, "Programming with Unicon", http://unicon.sourceforge.net.

[8] Ralph E. Griswold and Madge T. Griswold, The Icon Programming Language, 3rd edition. Peer to Peer Communications, San Jose, 1997.