A Framework for Automatic Debugging

Mikhail Auguston, Clinton Jeffery, Scott Underwood
Department of Computer Science, New Mexico State University
{mikau, jeffery, sunderwo}@cs.nmsu.edu

Abstract

This paper presents an application framework in which declarative specifications of debugging actions are translated into execution monitors that can automatically detect bugs. The approach is non-intrusive with respect to program source code and provides a high level of abstraction for debugging activities.

1. Motivation

Debugging is one of the most challenging, and least developed areas of software engineering. Debugging activities include queries regarding many aspects of target program behavior: sequences of steps performed, histories of variable values, function call hierarchies, checking of pre- and post-conditions at specific points, and validating other assertions about program execution. Performance testing and debugging involves a variety of profiles and time measurements.

We are building automatic debugging tools based on precise program execution behavior models that enable us to employ a systematic approach. Our program behavior models are based on events and event traces [1][2][3].

Debugging automation refers to a computation over an event trace. *Program execution monitors* are programs that load and execute a target program, obtain events at run-time, and perform computations over the event trace. Computations are performed during execution, post-mortem, or in any mixture of both times.

Any detectable action performed during a target program's run time is an *event*. For instance, expression evaluations, statement executions, and procedure calls are all examples of events. An event has a beginning, an end, and some duration; it occupies a time interval during program execution. This leads to the introduction of two basic binary relations on events: partial ordering and inclusion. Those relations are determined by target language syntax and semantics, e.g. two statement execution events may be ordered, or an expression evaluation event may occur inside a statement execution event. The set of events produced at the program run time, together with ordering and inclusion relations, is called an *event trace* and represents a model of program behavior. An event trace forms an acyclic directed graph (DAG) with two types of edges corresponding to the basic relations.

The language UFO (from Unicon-FORMAN) integrates the experience accumulated in the FORMAN [1] language and the Alamo monitoring architecture [4] to provide a complete solution for development of an extensive suite of automatic debugging tools. UFO is an implementation of FORMAN for debugging programs written in the Unicon and Icon programming languages [5][6].

2. Unicon and Alamo

Unicon is an imperative, goal-directed, object-oriented superset of Icon. Unicon's syntax is similar to Pascal or Java; its semantics features built-in backtracking, heterogeneous data structures and string scanning facilities. Unicon extends Icon's reach with elegant object-orientation, high level networking, messaging, and database facilities.

The reference implementation of Unicon is a virtual machine. Virtual machines (VMs) are attractive to language implementers because they provide portability and a vastly simpler implementation of very high level language features such as backtracking. As a result, event detection is an integral part of the VM.

VMs are ideal for developing debugging tools; they provide an appropriate level of abstraction for behavior models that describe program executions in a processor independent manner, as illustrated by the JPAX tool [7].

In Alamo, monitors and the target program execute as (sets of) coroutines with separate stacks and heaps inside a common VM. The Unicon VM is instrumented with over 100 kinds of atomic events, each one capable of reporting a <code,value> pair to monitors with interest in that event. Event reports are coroutine context switches.

Monitors are written independently from the target program, and can be applied to any target program without recompiling the monitor or target program. Monitors dynamically load target programs, and can easily query the state of arbitrary variables at each event report. Multiple monitors can monitor a program execution, under the direction of a monitor coordinator.

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 task of writing automatic debuggers to the task of specifying generic assertions about program behavior.

3. An Event Grammar for Unicon

Event grammars provide a model of program run time behavior. Monitors do not have to parse events using this grammar, since event detection is part of VM and UFO runtime system functionality. The following description provides a "lightweight" semantics of the Unicon programming language tailored for specification of debugging activities.

An event corresponds to a specific action of interest performed during program execution. Each event has one or more types and related attributes associated with it.

Universal attributes are found in every event. They are frequently used to narrow assertions down to a particular domain (function, variable, value) of interest. Some of the universal attributes are:

source_text: in canonical form (i.e. with redundant spaces eliminated, etc.)
line_num, col_num: source text locations
time_at_end, time_at_begin, duration: timing attributes
value_at_begin (Unicon-expression),
value_at_end (Unicon-expression): these attributes provide access to the program states

The event types, and type-specific attributes they provide, are summarized in the table below.

Event Type	Description	Attributes
prog_ex	whole program execution	
expr_eval	expression evaluation	value, operator, type, failure_p
func_call	function call	name, paramlist
param	actual parameter evaluation	name
func_body	function body execution	
input, output	I/O	file
variable	variable reference	
literal	reference to a constant value	
lhp	lefthand part, assignment	address
rhp	righthand part, assignment	
clause	then-, else-, or case branch execution	

test	test evaluation	
iteration	loop iteration	
return	return from procedure call	

Event types form a class hierarchy, shown in Figure 1. Subtypes inherit attributes from the parent type.
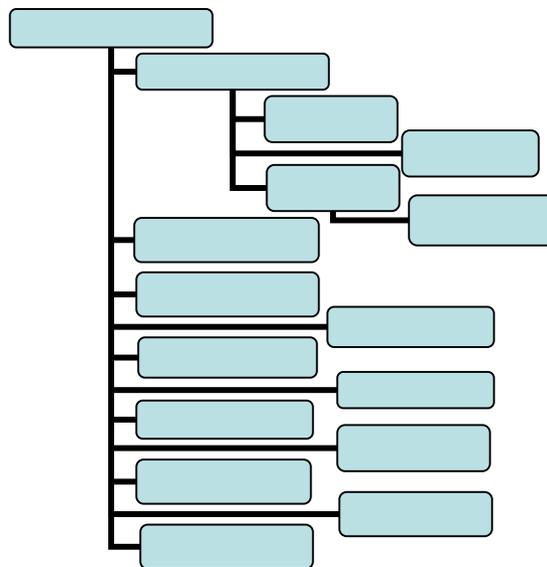

Figure 1. Event Type Inheritance Hierarchy

The UFO *event grammar* for Unicon is a set of axioms describing the structure of event traces with respect to two basic relations: inclusion and precedence. The grammar shown below is one possible abstraction of Unicon semantics; other event grammars might be used. The event grammar limits what kinds of bugs can be detected, so detail is useful. The grammar uses the notation:

Notation	Meaning
A :: (B C)	B precedes A, A includes B and C
A*	Zero or more A's under precedence
A+	One or more A's under precedence
A | B	Either A or B; alternative
A?	A is optional
{ A , B }	Set; A and B have no precedence

```
prog_ex::  ( expr_eval *)
expr_eval::( ( expr_eval ) |           unary op
            ( expr_eval expr_eval ) |  binary op
            ( expr_eval+ ) |
            ( test  clause ) |         conditional/
                                       case expressions
            ( iteration * ) |          loops
            ( { lhp,  rhp} ) )         assignment
```

lhp and rhp are not ordered, beginning of lhp precedes rhp, and end of lhp follows rhp

iteration::	(test expr_eval*)	(expr_eval* test)	(expr_eval *)
func_call::	(param* func_body)
func_body::	(expr_eval* return?)

Execution of a Unicon program produces an *event trace* organized by precedence and inclusion into a DAG. The structure of the event trace (event types, precedence and inclusion of events) is constrained by the event grammar axioms above. The event trace models Unicon program behavior and provides a basis to define debugging activities (assertion checking, debugging queries, profiles, debugging rules, behavior visualization) as appropriate computations over the event traces.

4. FORMAN

Alamo allows efficient monitors to be constructed in Unicon, but using a special-purpose language such as FORMAN, with the rich behavior model described in the preceding section, has compelling advantages. For example, in FORMAN we may refer to target program variable x, while in the Unicon monitor it is referenced as variable("x", &eventsource).

More important than such notational conveniences are FORMAN's control structures that support computations over event traces, centered around the notions of event pattern and aggregate operations over events.

The simplest event pattern comprises just an event type and matches successfully an event of this type or an event of a subtype of this type. Event patterns may include event attributes and other event patterns to specify the context of an event under consideration. For example, the event pattern

```
E: expr_eval:: ( R: rhp & is_an_object(R.value))
              & E.operator == ":="
```

matches an event of assignment type where the right hand part evaluates to an object. Temporary variables E and R provide an access to the events under consideration within the pattern.

The following example demonstrates the use of an aggregate operation.

```
CARD[ A: func_call &
          A.func_name == "read" FROM prog_ex ]
```

yields a number of events satisfying given event pattern, collected from the whole execution history. Expression […] is a list constructor and CARD is an abbreviation for a reduction of '+' operation over the more general list constructor:

```
+/[A: func_call & A.func_name == "read"
```

```
FROM prog_ex APPLY 1 ]
```

Quantifiers are introduced as abbreviations for reductions of Boolean operations OR and AND. For instance,

```
FOREACH Pattern FROM event_set Boolean_expr
```

is an abbreviation for

```
AND/[Pattern FROM event_set APPLY Boolean_expr ]
```

Debugging rules in FORMAN usually have the form:
Quantified_expr SAY-clauses ONFAIL SAY-clauses
The Quantified-expr is optional and defaults to TRUE. The execution of FORMAN programs relies on the Unicon monitors embedded in a virtual machine environment.

5. Examples of Debugging Rules

UFO supports and improves upon the most common application-specific debugging techniques. For example, UFO supports traditional precondition checking, or print statement insertion, without any modification of the target program source code. This is useful 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. It is possible to request evaluation of arbitrary Unicon expressions at the beginning or at the end of events.

```
DO AT EVERY   A: func_call &
         A.func_name == "my_func"
  FROM prog_ex {
   BEFORE  A
     { write("entering my_func, value of X is:", X) }
   AFTER  A
     { write("leaving my_func, value of X is:", X) }
}
```

This debugging rule causes run time instrumentation with calls to write() at selected points, 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 comprises several 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: "
       +/ [ r:input & r.filename=="xx.in"
              FROM prog_ex APPLY r.duration]
```

Another interesting 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. So far, the automatic debugging encyclopedia at http://www.cs.nmsu.edu/please/bugs.html has entries for 53 common bugs.

Example #3: Detecting Use of Un-initialized Variables. Reading an un-initialized variable is allowed in Unicon, but often leads to errors. Therefore, in this debugging rule all variables within the target program are checked to ensure that they are initialized before they are used.

```
FOREACH E: expr_eval CONTAINS (V: variable)
FROM prog_ex
  EXISTS D: lhp FROM E.prev_path
    D.source_text == V.source_text AND
      V.source_text BELONGS_TO
      ( E.scope SCOPE_INTERSECTION D.scope )
  ONFAIL SAY("Expression" E " contains the "
              " uninitialized variable " V.source_text)
```

SCOPE_INTERSECTION is similar to a set intersection, except that it takes into account scoping and visibility rules of the source language.

Example #4: Closed Files. Failure to close files that have been opened is an easily overlooked error. This assertion detects this event and warns the user. The temporary variable NumberOfClose holds the cardinality of the close() event set.

```
FOREACH a: func_call::(b:param) &
                a.func_name == "open"
  LET NumberOfClose =
    CARD[c:func_call::(d:param) &
          c.func_name == "close"  &
          b.source_text == d.source_text]
  IN IF NumberOfClose == 0 THEN
      SAY("Failed to close file" b.source_text
              "after opening at event " ,a)
      ELSEIF NumberOfClose > 1 THEN
        SAY("Attempt to close file " b.source_text
              "more than once") ENDIF
```

6. Implementation Issues

This section describes issues that have arisen during the implementation of UFO. 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,

precedence and containment relations, available for processing. This generality is extremely powerful; however the vast majority of 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 will compute a projection of the execution DAG necessary to perform the analysis.

The first step in generating code under the UFO translation model is to categorize each assertion as either "runtime", "post-mortem", or "hybrid", denoting the extent to which that assertion can be performed at runtime. Runtime and hybrid categorization is determined by constraints on FORMAN quantifier prefixes and results in more efficient monitor code. Nested quantifiers generally require post-mortem operation.

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.

Implementation of Example #1: Tracing. A single DO AT EVERY quantifier is quite typical of many UFO debugging actions and allows computation to be performed entirely at runtime. The events being counted and values being accumulated are used to construct an *event mask* in the initialization code that defines the Alamo events that will be monitored.

The monitor's event processing loop implements the filter based on procedure name within an if-expression. The Unicon code blocks containing write() expressions are inserted directly into the event loop for the relevant events. The complete monitor is:

```
$include "evdefs.icn"
link evinit
procedure main(av)
  EvInit(av) | stop("can't monitor ", av[1])

  ### initialization for BEFORE and AFTER func_call
  mask:= E_Pcall ++ E_Pret ++ E_Pfail

  while EvGet(mask) do {
    if &eventcode == E_Pcall &
      image(&eventvalue)=="procedure my_func" then
        ### inserted BEFORE clause
        write("entering my_func, value of X is:",
              variable("X", Monitored))
    if &eventcode == (E_Pret | E_Pfail) &
      image(&eventvalue)=="procedure my_func" then
        ### inserted AFTER clause
        write("leaving my_func, value of X is:",
```

```
                variable("X", Monitored))
    }
  end
```

Implementation of Example #2: Profiler. This is another typical situation, which involves an aggregate operation and selection of events according to a given pattern. The SAY expression is implemented by a call to write(); it must be performed post-mortem since it uses parameters whose values are constructed during the entire program execution. CARD denotes a counter, while SUM denotes an accumulator +/; both require a variable that is initialized to zero. The event subtypes and constraints are used to generate additional conditional code in the body of the event processing loop. Lastly, some attributes such as r.duration require additional events and measurements besides the initial triggering event. In the case of r.duration, a time measurement between the function call and its return is needed.

```
$include "evdefs.icn"
link evinit
procedure main(av)
  EvInit(av) | stop("can't monitor ", av[1])
  ### initialization for CARD and SUM
  cardreads := 0
  sumreadtime := 0
  mask := E_Fcall
  while EvGet(mask) do {
    ### count CARD of r:input...
    if &eventcode == E_Fcall &
      &eventvalue === (read|reads) then
        cardreads +:= 1
    ### add SUM of r.duration for r:input
    if &eventcode == E_Fcall &
      &eventvalue === (read|reads) then {
        thiscall := &time
        EvGet(E_Ffail++E_Fret)
        sumreadtime +:= &time - thiscall
      }
  }
  ### Translation of SAY
  write("Total number of read() statements: ",
      cardreads, "\n",
      "Elapsed time for read operations is: ",
      sumreadtime)
end
```

The advantage of the UFO approach is the combination of 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 FORMAN program, the generated Unicon monitor can be very selective about the behavior that it observes. The compiler merges several computations such as operation reduction or quantifiers present in the FORMAN assertions into a single Unicon

event loop. Since the compiler processes several assertions together, it can merge overlapping constructs (for example, those referring to the same events).

For certain kinds of FORMAN 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, and schedule corresponding computations. The most challenging and interesting remaining part of this compilation effort is to further optimize this analysis.

UFO's goal of practical application to real-sized programs has motivated improvements to the Alamo instrumentation of the Unicon VM. Although UFO is not complete enough to report conclusive results, the following table illustrates the effects of certain optimizations. The program in question is a mail message indexing tool, which processes mail headers and builds indices. For test purposes it is executed on a sample input of 3MB. All results are in seconds. The leftmost column shows the application's normal runtime. Columns 2-5 show runtimes for Implementation Example #2 above (the I/O function profiler) under Alamo, and three levels of optimization under UFO. Alamo imposed a 200% slowdown for comprehensive VM instrumentation, plus less than 100% slowdown for monitor code. Very little of the VM instrumentation is actually needed for this example. UFO-IO shows the effect of instrumentation optimization which UFO does at compile-time, optionally generating a custom VM for a given suite of FORMAN assertions. UFO-CO shows additional compiler optimizations on the monitor code. UFO-VM shows the effect of a runtime optimization called *value masking* on the virtual machine instrumentation. We are working on additional optimizations, and believe the end result will be highly practical execution from our high-level framework.

No monitor	Alamo	UFO-IO	UFO-CO	UFO-VM
1.35	3.64	2.82	2.30	1.87

7. Related Work

See www.cs.nmsu.edu/TechReports/2002/004.pdf for an expansion of this survey of related work.

The Event Based Behavioral Abstraction (EBBA) [8] characterizes program behavior in terms of primitive and composite events. Dalek is an event-based debugger for C built on top of GDB [9].

FORMAN takes a more comprehensive modeling approach than EBBA or Dalek, based on an event grammar and a language for expressing computations

over execution histories. Event grammars make FORMAN suitable for automatic source code instrumentation. FORMAN's abstraction of event as a time interval provides an appropriate level of granularity for reasoning about behavior, in contrast with the event notion in previous approaches where events are considered point-wise time moments.

Monitoring frameworks such as Dalek and COCA [10] use GDB to attain a necessary level of abstraction, which UFO finds in the Unicon virtual machine. While both approaches yield adequate source-level access and control over the monitored program, the virtual machine approach avoids substantial operating system overhead and offers better performance and scalability to larger programs.

Assertion languages provide yet another approach to debugging automation. Most approaches are based on Boolean expressions attached to points in the target program, like the assert() macro in C. [13,14,15] give approaches to programming with assertions for C and Ada. Even local assertions associated with particular points within the program may be extremely useful for program debugging. The DUEL [11] debugging language introduces expressions for C aggregate data exploration, for both assertions and queries.

The notion of computation over execution trace introduced in FORMAN is a generalization of Algorithmic Debugging [21, 22] and may be a convenient basis for describing generic debugging strategies.

PMMS [12] receives queries about target programs written in AP5, instruments source code, and stores data in a database to answer the posed questions. PMMS's domain specific query language is similar to FORMAN but tailored for database-style query processing.

8. Conclusions

The popularity of virtual machines promises to enable dramatic improvements in automatic debugging. These improvements will only occur if debugging is a specific goal of the virtual machine, e.g. as in the case of .net [13].

UFO illustrates what is possible for a broad class of languages such as those supported by the Java VM or the .net VM. Our approach uniformly represents many types of debugging-related activities as computations over traces. We have shown an approach to integrating event trace computations into a monitoring architecture based on a virtual machine. The end result provides a suitable environment for the implementation of automated debugging tools.

Acknowledgements

References

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

[2] M. Auguston, A. Gates, M. Lujan, "Defining a program Behavior Model for Dynamic Analyzers", in Proceedings of SEKE'97, Madrid, Spain, June 1997, pp. 257-262.

[3] 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.

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

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

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

[7] K. Havelund, S. Johnson, and G. Rosu. "Specification and Error Pattern Based Program Monitoring", European Space Agency Workshop on On-Board Autonomy, Noordwijk, Holland, October 2001.

[8] P. C. Bates, J. C. Wileden, "High-Level Debugging of Distributed Systems: The Behavioral Abstraction Approach", The Journal of Systems and Software 3, 1983, pp. 255-264.

[9] R. Olsson, R. Crawford, W. Wilson, "A Dataflow Approach to Event-based Debugging", Software -- Practice and Experience, Vol.21(2), February 1991, pp. 19-31.

[10] M. Ducasse, "COCA: An automated debugger for C", in Proceedings of ICSE 99, Los Angeles, 1999, pp.504-513.

[11] M. Golan, D. Hanson, "DUEL - A Very High-Level Debugging Language", in Proceedings of the Winter USENIX Technical Conference, San Diego, Jan. 1993.

[12] Y. Liao, D. Cohen, "A Specificational Approach to High Level Program Monitoring and Measuring", IEEE Transactions On Software Engineering, Vol. 18, No. 11, November 1992, 969 – 978.

[13] http://www.microsoft.com/net/.