

“Lightweight” Semantics Models for Program Testing and Debugging Automation

(Extended Abstract)

Mikhail Auguston

Computer Science Department, New Mexico State University,

Las Cruces, NM 88003, USA

mikau@cs.nmsu.edu

<http://www.cs.nmsu.edu/~mikau>

1 Introduction

We suggest an approach to the development of software testing and debugging automation tools based on precise program behavior models. The program behavior model is defined as a set of events (event trace) with two basic binary relations over events -- precedence and inclusion, and represents the temporal relationship between actions. A language for the computations over event traces is developed that provides a basis for assertion checking, debugging queries, execution profiles, and performance measurements.

The approach is nondestructive, since assertion texts are separated from the target program source code and can be maintained independently. Assertions can capture both the dynamic properties of a particular target program and can formalize the general knowledge of typical bugs and debugging strategies. An event grammar provides a sound basis for assertion language implementation via target program automatic instrumentation. Event grammars may be designed for sequential as well as for parallel programs. The approach suggested can be adjusted to a variety of programming languages. We illustrate these ideas on examples for the Occam and C programming languages.

Dynamic program analysis is one of the least understood activities in software development. A major problem is still the inability to express the mismatch between the expected and the observed behavior of the program on the level of abstraction maintained by the user [9]. In other words, a flexible and expressive specification formalism is needed to describe properties of the software system's implementation. Program testing and debugging is still a human activity performed largely without any adequate tools and consuming more than 50% of the total program development time and effort [8]. Debugging concurrent programs is even more difficult because of parallel activities, non-determinism and time-dependent behavior.

One way to improve the situation is to partially automate the debugging process. Precise *model of program behavior* becomes the first step towards debugging automation. It appears that traditional methods of programming language semantics definition don't address this aspect. In building such a model several considerations were taken in account. The first assumption we make is that the model is discrete, i.e. comprises a finite number of well-separated elements. This assumption is typical for Computer Science methods used for static and dynamic analysis of programs. For this reason the notion of *event* as an elementary unit of action is an appropriate basis for building the whole model. The event is an abstraction for any detectable action performed during the program execution, such as a statement execution, expression evaluation, procedure call, sending and receiving a message, etc.

Actions (or events) are evolving in time and the program behavior represents the temporal relationship between actions. This implies the necessity to introduce an ordering relation for events. Semantics of parallel programming languages and even some sequential languages (such as C) don't require the total ordering of actions, so *partial event*

ordering is the most adequate method for this purpose [11].

Actions performed during the program execution are at different levels of granularity, some of them include other actions, e.g. a subroutine call event contains statement execution events. This consideration brings to our model *inclusion relation*. Under this relationship events can be hierarchical objects and it becomes possible to consider program behavior at appropriate levels of granularity.

Finally, the program execution can be modeled as a set of events (*event trace*) with two basic relations: partial ordering and inclusion. The event trace actually is a model of program's behavior temporal aspect. In order to specify meaningful program behavior properties we have to enrich events with some attributes. An event may have a type and some other attributes, such as event duration, program source code related to the event, program state associated with the event (i.e. program variable values at the beginning and at the end of event), etc.

The next problem to be addressed after the program behavior model is set up is the formalism specifying properties of the program behavior. Since our goal is debugging automation, i.e. a kind of program dynamic analysis that requires different types of assertion checking, debugging queries, program execution profiles, and so on, we came up with the concept of a *computation over the event trace*. It seems that this concept is general enough to cover all the above mentioned needs in the unifying framework, and provides sufficient flexibility. This approach implies the design of a special programming language for computations over the event traces. We suggest a particular language called FORMAN [1], [3], [10] based on functional paradigm and the use of event patterns and aggregate operations over events.

Patterns describe the structure of events with context conditions. Program paths can be described by path expressions over events. All this makes it possible to write assertions not only about variable values at program points but also about data and control flows in the target program. Assertions can also be used as conditions in rules which describe debugging actions. For example, an error message is a typical action for a debugger or consistency checker. Thus, it is also possible to specify debugging strategies.

The notions of event and event type are powerful abstractions which make it possible to write assertions independent of any target program. Such generic assertions can be collected in standard libraries which represent the general knowledge about typical bugs and debugging strategies and could be designed and distributed as special software tools.

FORMAN is a general language to describe computations over program event trace that can be considered as an example of a *special programming paradigm*. Possible application areas include program testing and debugging, performance measurement and modeling, program profiling, program animation, program maintenance and program documentation [5]. A study of FORMAN application for parallel programming is presented in [4]

2 Events, Event Traces, and the Language for Computations Over Event Traces

FORMAN is based on a semantic model of target program behavior in which the program execution is represented by a set of events. An *event* occurs when some action is performed during the program execution process. For instance, a message is sent or received, a statement is executed, or some expression is evaluated. A particular action may be performed many times, but every execution of an action is denoted by a unique event.

Every event defines a time interval which has a beginning and an end. For atomic events, the beginning and end points of the time interval will be the same. All events used for assertion checking and other computations over event traces must be detectable by some implementation (e.g. by an appropriate target program instrumentation.) Attributes attached to events bring additional information about event context, such as current variable and expression values.

The model of target program behavior is formally defined through a set of general axioms about two basic relations, which may or may not hold between two arbitrary events: they may be sequentially ordered (PRECEDES), or one of them might be included in another composite event (IN). For each pair of events in the event trace no more

than one of these relations can be established.

There are several general axioms that should be satisfied by any events a, b, c in the event trace of any target program.

1) Mutual exclusion of relations.

$a \text{ PRECEDES } b \Rightarrow \text{not} (a \text{ IN } b) \text{ and not } (b \text{ IN } a)$
 $a \text{ IN } b \Rightarrow \text{not}(a \text{ PRECEDES } b) \text{ and not } (b \text{ PRECEDES } a)$

2) Noncommutativity.

$a \text{ PRECEDES } b \Rightarrow \text{not}(b \text{ PRECEDES } a)$
 $a \text{ IN } b \Rightarrow \text{not}(b \text{ IN } a)$

3) Transitivity.

$(a \text{ PRECEDES } b) \text{ and } (b \text{ PRECEDES } c) \Rightarrow (a \text{ PRECEDES } c)$
 $(a \text{ IN } b) \text{ and } (b \text{ IN } c) \Rightarrow (a \text{ IN } c)$

Irreflexivity for PRECEDES and IN follows from 2). Note that PRECEDES and IN are irreflexive partial orderings.

4) Distributivity

$(a \text{ IN } b) \text{ and } (b \text{ PRECEDES } c) \Rightarrow (a \text{ PRECEDES } c)$
 $(a \text{ PRECEDES } b) \text{ and } (c \text{ IN } b) \Rightarrow (a \text{ PRECEDES } c)$
 $(\text{FOR ALL } a \text{ IN } b (\text{FOR ALL } c \text{ IN } d (a \text{ PRECEDES } c))) \Rightarrow (b \text{ PRECEDES } d)$

In order to define the behavior model for some target language, types of events are introduced. Each event belongs to one or more of predefined event types, which are induced by target language abstract syntax (e.g. execute-statement, send-message, receive-message) or by target language semantics (rendezvous, wait, put-message-in-queue).

The target program execution model is defined by an event grammar. The event may be a compound object and the grammar describes how the event is split into other event sequences or sets. For example, the event execute-assignment-statement contains a sequence of events evaluate-right-hand-part and execute-destination. The evaluate-right-hand-part, in turn, consists of an unique event evaluate-expression. The event grammar is a set of axioms that describe possible patterns of basic relations between events of different type in the program execution history, it is not intended to be used for parsing actual event trace.

The rule $A :: (B C)$ establishes that if an event a of the type A occurs in the trace of a program, it is necessary that events b and c of types B and C , also exist, such that the relations $b \text{ IN } a, c \text{ IN } a, b \text{ PRECEDES } c$ hold.

For example, the event grammar describing the semantics of a PASCAL subset may contain the following rules. The names, such as execute-program, and ex-stmt denote event types.

$\text{execute-program} :: (\text{ex-stmt} *)$

This means that each event of the type execute-program contains an ordered (w.r.t. relation PRECEDES)

sequence of zero or more events of the type `ex-stmt`.

```
ex-stmt :: ( label? ( ex-assignment | ex-read-stmt | ex-write-stmt |  
                    ex-reset-stmt | ex-rewrite-stmt | ex-close-stmt | ex-cond-stmt |  
                    ex-loop-stmt | call-procedure ) )
```

The event of the type `ex-stmt` contains one of the events `ex-assignment`, `ex-read-stmt`, and so on. This inner event determines the particular type of statement executed and may be preceded by an optional event of the type `label` (traversing a label attached to the statement).

```
ex-assignment :: (ex-righthand-part destination)
```

The order of event occurrences reflects the semantics of the target language. When performing assignment statement first the right-hand part is evaluated and after this the destination event occurs (which denotes the assignment event itself). The event grammar makes FORMAN suitable for automatic source code instrumentation to detect all necessary events.

An event has attributes, for instance, source text fragment from the corresponding target program, current values of target program variables and expressions at the beginning and at the end of event, duration of the event, previous path (i.e. set of events preceding the event in the target program execution history), etc.

FORMAN supplies a means for writing assertions about events and event sequences and sets. These include quantifiers and other aggregate operations over events, e.g., sequence, bag and set constructors, boolean operations and operations of target language to write assertions on target program variables [2] [3]. Events can be described by patterns which capture the structure of event and context conditions. Program paths can be described by regular path expressions over events.

The main extension for the parallel case [4] consists of the introduction of a new kind of composite event -- "snapshot," which can be considered an abstraction for the notion "a set of events that may happen at the same time." The "snapshot" event is a set of events each pair of which is not under the relation PRECEDES, this makes it possible to describe and to detect at run-time such typical parallel processing faults as data races and deadlock states.

3 Examples of Debugging Rules and Queries

In general, a *debugging rule* performs some actions that may include computations over the target program execution history. The aim is to generate informative messages and to provide the user with some values obtained from the trace in order to detect and localize bugs. Rules can provide dialog to the user as well. An assertion is a boolean expression that may contain quantifiers and sequencing constraints over events.

Assertions can be used as conditions in the rules describing actions that can be performed if an assertion is satisfied or violated. A debugging rule has the form:

```
assertion    SAY (expression sequence)  
  
ONFAIL SAY (expression sequence)
```

The presence of metavariables in the assertion makes it possible to use FORMAN as a debugger query language. The computation of an assertion is interrupted when it becomes clear that the final value will be False, and the current values of metavariables can be used to generate readable and informative messages.

The following examples have been executed on our prototype FORMAN/PASCAL assertion checker [2], [3]. The

PASCAL program reads a sequence of integers from file XX.TXT.

```
program e1;
  var X: integer;
      XX: file of text;
begin
  X:= 7;
  (* initial value is assigned here *)
  reset (XX, 'XX.TXT');
  while X<>0 do
    read(XX, X)
  end.
```

The contents of the file XX.TXT are as follows:

```
11 5 3 7 8 9 3 13 2 3 45 8 754 45567 0
```

Example of a Query 1. In order to obtain the history of variable X the following computation over event trace can be performed. The rule condition is TRUE, and is shown as a side effect the whole history of variable X.

```
TRUE
```

```
SAY ('The history of variable X is:')
```

```
[D: destination IS X FROM execute_program APPLY VALUE(D) ] )
```

The [...] construct above defines a loop over the whole program execution trace (execute_program event). All events matching the pattern destination IS X are selected from the trace and the function VALUE is applied to them. The resulting sequence consists of values assigned to the X variable during the program execution.

When executed on our prototype the following output is produced:

```
Assertion #1 checked successfully...
```

```
The history of variable X is: 7 11 5 3 7 8 9 3 13 2 45 8 754 45567 0
```

Example of an Assertion 2. Let's write and check the assertion : "The value of variable X does not exceed 17."

```
FOREACH *S: ex_stmt CONTAINS (D: destination IS X) FROM execute_program
```

```
VALUE(D) < 17
```

```
ONFAIL
```

```
SAY('Value ` VALUE(D) `is assigned to the variable X in stmt `)
```

```
SAY(S)
```

```
SAY('This is record #' CARD[ ex_read_stmt FROM PREV_PATH(S)] + 1 `in the  
file XX.TXT')
```

We check the assertion for all events where the value of X may be altered. These are events of the type `destination` which can appear within `ex_assignment_stmt` or `ex_read_stmt` events. In order to make error messages about assertion violations more informative we include the embracing event of the type `ex_stmt`. Metavariables S and D refer to those events of interest. When the assertion is violated for the first time, the assertion evaluation terminates and current values of metavariables can be used for message output. The value of a metavariable when printed by the SAY clause is shown in the form:

```
event-type:> event-source-text
```

```
Time= event-begin-time .. event-end-time
```

Event begin and end times in this prototype implementation are simply values of step counter.

Since we expect the assertion might be violated when executing a Read statement, it makes sense to report the record number of the input file `xx.txt` where the assertion is violated. The program state does not contain any variables which values could provide this information. But we can perform auxiliary calculations independently from the target program using FORMAN aggregate operations. In this particular case the number of events of the type `ex_read_stmt` preceding the interruption moment is counted. This number plus 1 (since the violation occurs when the read statement is executed) yields the number of an input record on which the variable X was first assigned the value exceeding 17.

```
Assertion # 2 violation!
```

```
Value 45 is assigned to the variable X in stmt
```

```
ex_stmt :> Read( XX , X )      Time= 73 .. 78
```

```
This is record # 11 in the file XX.TXT
```

Example of a Query 3. Profile measurement. In order to obtain the actual number of statements executed, the following query can be performed:

```
TRUE
```

```
SAY('The total number of statements executed is:')
```

```
CARD[ ALL ex_stmt FROM execute_program ])
```

The ALL option in the aggregate operation indicates that all nested events of the type `ex_stmt` should be taken

into account.

Assertion #3 checked successfully...

The total number of statements executed is: 18

Example of a *generic assertion* which must be true for any program in the target language.

“Each variable has to be assigned value before it is used in an expression evaluation.”

FOREACH * S: ex_stmt FROM execute_program

 FOREACH * E: eval_expression CONTAINS (V: variable) FROM S

 EXISTS D: destination FROM PREV_PATH(E) SOURCE_TEXT(D) = SOURCE_TEXT(V)

ONFAIL

 SAY('In event' S)

 SAY('in expression evaluation')

 SAY(E)

 SAY('uninitialized variable' SOURCE_TEXT(V) 'is used')

For the following PASCAL program our prototype detects the presence of the bug described above.

```
program e2;
```

```
var X,Y: integer;
```

```
begin    Y:= 3;
```

```
    if Y < 2 then begin
```

```
        X:= 7; Y:= Y + X
```

```
    else Y:= X - Y (** here the error appears: X has no value! **)
```

```
end.
```

Assertion #4 violation!

```
In event ex_stmt :> If ( Y < 2 ) then X := 7 ; Y := ( Y + X ) ;
```

```
                                else Y := ( X - Y ) ; Time= 10 .. 35
```

```
in expression evaluation
```

```
eval_expression :> ( X - Y ) Time= 20 .. 29
```

```
uninitialised variable X is used
```

Debugging rules can be considered as a way of formalizing reasoning about the target program execution -- humans often use similar patterns for reasoning when debugging programs. For example, if the index expression of an array element is out of the range, the debugger can try a rule for eval-index events that invokes another rule about wrong value of the event eval-expression, which in turn will cause investigation of histories of all variables included in the expression.

Yet another application of generic assertions and debugging rules may be for describing run-time constraints (sequences of procedure calls, actual parameter dependences, etc.) for nontrivial subroutine packages, e.g. for the MOTIF package for GUI design. A library containing assertions and debugging rules relevant to such a package may be useful for writing C programs calling subroutines from the package.

4 Conclusions

In brief, our approach can be explained as “computations over a target program event trace.” We expect the advantages of our approach to be the following:

- The notion of **an event grammar** provides a general basis for program behavior models. In contrast with previous approaches, the **event** is not a point in the trace but an interval with a beginning and an end.
- Event grammar provides a coordinate system to refer to any interesting event in the execution history. Program variable values are attributes of an event’s beginning and end. Event attributes provide complete **access to each target program’s execution state**. Assertions about particular execution states as well as assertions about sets of different execution states may be checked.
- The PRECEDES relation yields a **partial order** on the set of events, which is a natural model for parallel program behavior.
- The IN relation yields a **hierarchy of events**, so the assertions can be defined at an appropriate level of granularity.
- A language for **computations over event traces** provides a **uniform framework** for assertion checking, profiles, debugging queries, and performance measurements.
- The access to the complete target program execution history and the ability to formalize **generic assertions** can be used in order to define **debugging rules and strategies**.
- The fact that assertions and other computations over target program event trace can be **separated from the text of the target program** allows accumulation of formalized knowledge about particular programs and about the whole target language in separate files. This makes it easy to control the amount of assertions to be checked.

According to [7] and [12] approximately 40-50% of all bugs detected during the program testing are logic, structural, and functionality bugs, i.e. bugs which could be detected by appropriate assertion checking similar to the demonstrated above.

References

- [1] M. Auguston, "FORMAN -- A Program Formal Annotation Language", *Proceedings of the 5:th Israel Conference on Computer Systems and Software Engineering*, Gerclia, May 1991, IEEE Computer Society Press, 149-154.
- [2] M. Auguston, "A language for debugging automation", *Proceedings of the 6th International Conference on Software Engineering and Knowledge Engineering*, Jurmala, June 1994, Knowledge Systems Institute, pp. 108-115.
- [3] M. Auguston, "Program Behavior Model Based on Event Grammar and its Application for Debugging Automation", in *Proceedings of the 2nd International Workshop on Automated and Algorithmic Debugging*, Saint-Malo, France, May 1995.
- [4] M. Auguston, P. Fritzson, "PARFORMAN -- an Assertion Language for Specifying Behavior when Debugging Parallel Applications", *International Journal of Software Engineering and Knowledge Engineering*, vol.6, No 4, 1996, pp. 609-640.
- [5] M. Auguston, A. Gates, M. Lujan, "Defining a program Behavior Model for Dynamic Analyzers", *Proceedings of the 9th International Conference on Software Engineering and Knowledge Engineering*, SEKE'97, Madrid, Spain, June 1997, pp. 257-262
- [6] 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.
- [7] B. Beizer, *Software Testing Techniques*, Second Edition, International Thomson Computer Press, 1990.
- [8] F. Brooks, *The Mythical Man-Month*, 2nd edition, Addison-Wesley, 1995.
- [9] B. Bruegge, P. Hibbard, "Generalized Path Expressions: A High-Level Debugging Mechanism", *The Journal of Systems and Software* 3, 1983, pp. 265-276.
- [10] P. Fritzson, M. Auguston, N. Shahmehri, "Using Assertions in Declarative and Operational Models for Automated Debugging", *The Journal of Systems and Software* 25, 1994, pp. 223-239.
- [11] L. Lamport, "Time, Clocks, and the Ordering of Events in a Distributed System", *Communications of the ACM*, vol. 21, No. 7, July 1978, pp. 558-565.
- [12] S. L. Pfleeger, *Software Engineering, Theory and Practice*, Prentice Hall, 1998.