A Program Behavior Model Based on Event Grammar and its Application for Debugging Automation[1]

Mikhail Auguston
Department of Computer Science
New Mexico State University
Las Cruces, NM 88003-0001, USA
e-mail: mikau@cs.nmsu.edu

Abstract.

The notion of event constitutes the basis of the target program behavior model. This model, called H-space (History-space), is formally defined through a set of axioms (event grammar) for two basic relations, which may hold between two arbitrary events: the events may be sequentially ordered or one of them might be included in another composite event. Target program execution history is represented as a set of nested events. A general paradigm of "computation over execution history" is suggested in this paper as a basis for debugging automation. A language to describe computations over execution histories of target programs provides means for assertion checking, debugging queries, profiles, and performance measurement. Event patterns and aggregate operations over event traces, can be used in order to describe typical bugs and debugging rules. The knowledge about typical bugs and debugging rules can be formalized in libraries of assertions and debugging rules and can be used for run-time detection of bugs and bug localization. Examples of assertions and debugging rules are presented and prototype implementation issues are briefly discussed.

Key words.

Testing and debugging automation, assertion languages, computations over traces

1. This is an extended version of a paper presented at the 2nd International Workshop on Algorithmic and Automated Debugging AADEBUG'95, Saint-Malo, France, May 1995.

1 Overview

Testing and debugging are still the most effort consuming activities during software life cycle. In this paper we describe an approach to the testing and debugging automation based on the specialized assertion language. FORMAN (FORMal ANnotation) language [2], [3], [4], [12] is intended to be used in conjunction with some high-level programming language called the *target language*. Our approach is based on the introduction of a precise model of target program behavior. The execution of the target program is represented by a set of events. Models of intended or faulty behavior can be specified in FORMAN. These models are then compared with the actual behavior in order to localize possible bugs. A language is provided to describe the properties of the target program behavior that may be expressed at the source code level.

FORMAN is more than just an assertion language. In fact, it is a general language for describing computations over target program execution histories such as profiles, debugging queries, performance measurements. Auxiliary computations over target program execution histories provide more comprehensive and informative error messages. Some of the previous approaches to debugging automation, e.g. Algorithmic Debugging could be expressed within FORMAN framework (see Sec. 6.1).

1.1 Event Notion

The Event Based Behavioral Abstraction (EBBA) method suggested in [6], [7] characterizes the behavior of the whole program in terms of both primitive and composite events. Context conditions involving event attribute values can be used to distinguish events. Events can be defined by the user and do not necessarily have a relation to the semantics of the programming language to be debugged. EBBA defines two higher level means for modeling system behavior -- clustering and filtering. Clustering is used to express behavior as composite events, i.e. aggregates of previously defined events. Filtering serves to eliminate from consideration events which are not relevant to the model being investigated. Both event recognition and filtering can be performed at run-time.

An Event Based debugger for C language called Dalek [22] provides means for description of user-defined events which typically are points within program execution trace. Target program has to be instrumented in order to collect values of event attributes. Composite events can be recognized at run-time as collections of primitive events.

FORMAN has a more comprehensive modelling approach than EBBA or Dalek, based on the event grammar. A language for expressing computations over execution histories is provided, which is missing in EBBA and Dalek. The event grammar makes FORMAN suitable for automatic source code instrumentation to detect all necessary events. FORMAN supports design of universal assertions and debugging rules that could be used for debugging of arbitrary target programs. This generality is missing in EBBA and Dalek approaches.

1.2 Path Expressions

Data and control flow descriptions of the target program are essential for testing and debugging purposes. It is useful to give such a description in an explicit and precise form. The path expression technique introduced for specifying parallel programs in [10] is one such formalism. Trace specifications also are used in [21] for software specification. This technique has been used in several projects as a background for high-level debugging tools, e.g. in [8], where path rules are suggested as a kind of debugger commands. Path expressions have also been used for concurrent system formal specification purposes in the Aleph project [28]. FORMAN provides flexible language means for trace specification including event patterns and regular expressions over them.

1.3 Assertion Languages

Assertion languages provide yet another approach to debugging automation. The approaches currently in use are mostly based on boolean expressions attached to selected points of the target program. The ANNA [19] annotation language for the Ada target language, supports assertions on variable and type declarations. In the TSL [17], [20] and RAPID [18] specification languages for Ada, the notion of event is introduced in order to describe the behavior of tasks. Patterns can be written which involve parameter values of task entry calls. Assertions are written in Ada itself, using a number of special pre-defined predicates. Assertion-checking is dynamic at run-time, and does not need post-mortem analysis. An annotation language for C programming language is described in [24].

In [5] events are introduced to describe process communication, termination, and connection and detachment of process to channels. A language of Behavior Expressions (BE) is provided to write assertions about sequences of process interactions. BE is able to describe allowed sequences of events as well as some predicates defined on the values of the variables of processes. Event types are process communications and interactions such as send, receive, terminate, connect, detach. Evaluation of assertions are done at run-time. No composite events are provided.

Another recent debugging tool is based on trace analysis with respect to assertions in temporal interval logic. This work is presented in [14], [15] where four types of events are introduced - assignment to variables, reaching a label, interprocess communication and process instantiation or termination. Composite events cannot be defined.

1.4 Algorithmic Debugging

The original algorithmic program debugging method was introduced in [27] for the Prolog language. In [26] and [13] this paradigm is applied to an imperative target language (subset of PASCAL). The algorithmic program debugger executes the program and builds a trace execution tree at the procedure level. Then the debugger traverses the execution tree and interacts with the user by asking about the intended behavior of each procedure. The user has the possibility to answer "yes" or "no" about the observed results of the procedure. The search finally ends and a bug is localized within a procedure *p* when one of the following holds:

- Procedure *p* contains no procedure calls.
- All procedure calls performed from the body of procedure *p* fulfill the user's expectations.

Algorithmic debugging can be considered as an example of a debugging strategy, based on some assertion language (in this case assertions about results of a procedure call). This is a step towards turning a debugger role from plain observation to a more active user partner.

The notion of computation over execution trace introduced in FORMAN may be a convenient basis for describing debugging strategies.

2 Brief Exposition of the Approach

We expect the advantages of FORMAN approach to be in the following.

- The notion of **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 variables values are attributes of event 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 relationship between different execution states may be checked.
- The IN relation yields a **hierarchy of events** at appropriate level of granularity.
- A language for **computations over execution histories** 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 **universal assertions** can be used in order to define **debugging rules and strategies.**
- The fact that assertions and other computations over target program history can be **separated from the text of the target program** allows us to accumulate formalized knowledge about concrete programs and about the whole target language in separate files. This makes it easy to control the amount of assertions to be checked.

2.1 H-Space

The semantic model of a target program execution can be represented by a set of events. An *event* occurs when some action is performed in the program execution process, for instance, some statement is executed or some expression is evaluated. A particular action may be performed many times, but every performance is denoted by a unique event. Every event defines a time interval which has a beginning and an end. A *composite event* contains a set of other events. All events defined in the model should be detectable by some means, e.g. by an appropriate target program instrumentation that will generate at the run time messages about beginning and end of each of the relevant events.

We introduce H-space (the History-space) of program execution, which is a set of events with two basic binary relations:

1) `a << b` (a precedes b)

The target language semantics usually determines which events can precede others, e.g. for sequential statement execution, or for compound statement parts evaluation. There may be cases when the order of events is not predicted by target language semantics, e.g. the order of subexpression evaluation in some imperative languages is not determined. For such occasions the precedence relation for corresponding pair of events is not defined. In general case the precedence relation defines a partial order on the set of events.

2) `a` **`IN`** `b` (b contains a)

This relation usually holds between statement and compound statement execution events, between subexpression and expression evaluation events, i.e. is mostly implied by target language syntax.

For each pair of events in H-space no more than one of these relations can be established. The presence or absence of any of the basic relations for any pair of events in the H-space may be checked by an appropriate implementation, e.g. by instrumentation of the target program. The relations defined for elements in H-space must satisfy the following axioms:

1) Mutual exclusion of relations <<, **`IN`**.

```
a << b  =>   not (a IN b) and not ( b IN a)
a IN b  =>  not(a << b) and not (b << a)
```

2) Noncommutativity

```
a << b  =>  not( b << a)
a IN b  =>  not( b IN a)
```

3) Transitivity

```
(a << b ) and ( b << c ) => ( a << c)
```

The `IN` relation is not transitive since `a IN b` denotes that `a` is an immediate component of `b`.

4) Some other axioms

```
(a IN b) and (b << c) => (a << c)
(a << b) and (c IN b) => (a << c)
(for each a IN b ( for each c IN d  ( a << c) ))  =>  ( b << d )
```

2.2 Event Grammar -- An Operational Model of Program Semantics

Each event in the H-space belongs to some **event type** that can be determined by appropriate implementation.

The target program's operational model is defined by the *event grammar*. An event may be a compound object and the grammar describes how the event is split into other events. Events that contain other events are called *composite events*. Some events (e.g. `variable`, `constant` in the grammar below) are considered to be *elementary* (nondecomposable).

It is assumed that all events in the H-space *can be detected by some implementation*. The event grammar **is not provided to be used for event trace parsing**. Rather it presents a set of axioms describing properties of the H-space. The event grammar may be used for **automatic instrumentation** of the target program in order to detect event appearances at the run time.

As an example, the following rule states that if an event `a` of the type A occurs in the H-space, there necessarily also exist events `b` and `c` of types `B` and `C`, such that the relations `b` **`IN`** `a`, `c` **`IN`** `a`, and `b << c` hold.

```
A :: (B  C)
```

A small subset of PASCAL is used here for the target language example. Metasymbols as used for BNF, are used to define the event grammar: '*' denotes repetition zero or more times,'+' denotes repetition one or more times, alternatives are separated by the symbol '|', optional elements are denoted by '?'. Parentheses '(' and ')' are used for grouping.

The names, such as `execute-program`, `ex-stmt` and other in the following grammar denote event types.

```
execute-program  :: ( ex-stmt * )
```

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

```
ex-righthand-part::(eval-expression )
destination ::(variable |  array-elt )
```

The event vocabulary has to be rich enough to describe the situations in program execution history. This is the reason why many aliases are introduced in the event grammar. Some elementary events are included in the event grammar to make the identification of other events easier, e.g. `file-name` in the following rules.

```
ex-read-stmt     :: ( file-name  destination +)
ex-write-stmt   :: ( file-name  eval-expression + )
ex-reset-stmt   :: ( file-name  )
ex-rewrite-stmt :: ( file-name  )
ex-close-stmt   :: ( file-name  )
ex-cond-stmt    :: ( eval-condition (ex-then-branch | ex-else-branch))
```

When conditional statement is executed the condition is evaluated first and then according to the value of the condition either `ex-then-branch`, either `ex-else-branch` event takes place (but not both at the same time). The event grammar does not say anything about values of expressions etc. It describes sets of events and basic relations between them that could be observed at the run time.

```
eval-condition     :: ( eval-expression )
ex-then-branch     :: ( ex-stmt*  )
ex-else-branch     :: ( ex-stmt*  )

ex-loop-stmt       :: ( ex-one-iteration +)
ex-one-iteration   :: ( eval-condition  ex-loop-body? )
ex-loop-body       :: ( ex-stmt* )
```

Any event `ex-loop-stmt` contains at least one `ex-one-iteration` since the loop condition has to be evaluated at least once (this event corresponds to the execution of while-do loop). The `ex-loop-body` event will be dropped only at the last loop iteration.

```
call-procedure ::( procedure-name eval-actual-parameter* ex-proc-body)
eval-actual-parameter      :: ( parameter-name  eval-expression)
ex-proc-body               :: ( ex-stmt *  )

eval-expression    :: ( eval-simple-expr |  eval-bin-op)
eval-simple-expr   :: ( constant | variable | array-elt)
array-elt          :: ( array-name eval-index)
eval-index         :: ( eval-expression )
eval-bin-op        ::(  eval-1st-arg eval-2nd-arg perform-bin-op )
```

```
eval-1st-arg       :: ( eval-expression )
eval-2nd-arg       :: ( eval-expression )
```

According to this grammar, an event can be considered as an ordered tree containing other events. All possible (terminating) target program execution histories can be described by this grammar. Certainly, not all events satisfying this grammar can really be observed in the trace of some target program execution. It should also be stressed that an event is not an isolated object. It is always considered in the context of enclosing events with `execute-program` at the top.

2.3 Event Attributes

A number of functions in FORMAN are predefined for most events in the H-space. They are called *event attributes*. In the sequel, we let E denote an event.

`Source-text``(E)` is a function that given an event E returns the corresponding piece of target program text. For instance, the *`Source-text`* for the event `eval-expression` yields the text of the expression, the *`Source-text`* for the event `ex-stmt` yields the text of the statement, and so on. The only operation allowed is '=', i.e. comparison for equality.

`Value``(E)` is a partial function that returns the value (in the sense of the target language) at the end of the event. For such events as `eval-expression, destination,` and `constant` the meaning is clear: it is equal to the value of the corresponding evaluated object. For some other events it is better to assume that *`Value`* is undefined.

`Value-at-end``(E, X)` denotes the value of target program variable X at the moment of the end of the event `E`, and *`Value-at-begin`*`(E, X)` denotes the value of variable X at the moment of the beginning of the event E. These event attributes provide an access to the target program execution state.

`Prev-path``(E)` has a range which is a set of events. This function yields the minimal set of all completed events in the target program execution history preceding the E event. If events `a` and `b` both precede `E`, and `b IN a` holds, then only `a` is included in the *`Prev-path`*`(E)`.

`Duration``(E)` is used for performance measurement modeling. This value is attached to each elementary event. For composite events the duration is the sum of its constituent durations plus some overhead. This attribute can be either implemented using real machine timer, either modeled time. In the later case duration estimates can be tuned to approximate some real run-time environment.

3 A Language for Computations over H-space

Since we have defined precisely the model of target program execution, some means to describe computations over target program execution history can be introduced. The reach enough vocabulary of event types makes it possible to refer to any "interesting" event in the history. Event attributes provide access to the program states (variable values) at different points within history. Such "computational’ point of view to the execution history provides a general framework for assertion checking, program profiles, performance measurement, and debugging queries.

FORMAN provides means to write assertions about events and event sequences. These include *quantifiers* and other aggregate operations over events, e.g. sequence, bag and set *constructors*, boolean operations and operations of the target language in order to express assertions about target program variables. Events can be described by *patterns* which match the structure of events and context conditions. Program paths can be described by regular path expressions over events.

3.1 Event Patterns

Event patterns in FORMAN are used in operations and predicates to distinguish events which satisfy some special conditions. The simplest form of event pattern is:

```
E
```

where `E` is an event type name as it is defined in the event grammar of the target language, e.g. `eval-expression`. An event satisfies this pattern *if and only if* it is of the type `E`.

Metavariables are introduced in FORMAN to give a virtual name to the event under consideration while matching it with the pattern. A metavariable name is an identifier written before the pattern and is delimited from it by the symbol ':'. The scope of the metavariable is the whole debugging rule (see Sec. 5).

A *context condition* is a Boolean expression written after the pattern and delimited from the pattern by the '&' symbol. Attributes and target language operations are used to describe context conditions. For example, the pattern below matches an event of the type `eval-expression` such that the source text of the whole expression is the variable `ABC` itself.

`X:eval-expression & ` *`Source-text`*`(X)= 'ABC'`

Also, sequences of events can be matched with patterns. The common BNF notation may be used to construct appropriate patterns, as in the following regular expression over events.

```
ex-reset-stmt  ex-read-stmt *        ex-close-stmt?
```

Patterns to describe the internal structure of an event are frequently needed. Let E be an event name, and P1, P2 and P3 be patterns.

Here is an example of a composite pattern:

```
E :: ( P1 P2 P3 )
```

An event X satisfies this pattern *if and only if* it is an event of the type E and at the next level contains events which match with P1, P2, and P3 in the same order (the event X can contain some other elements at this level too). Ordering with respect to the relation << is assumed. For example,

```
array-elt :: (    N: array-name
                  I: eval-index) & Source-text(N) = 'M' AND Value(I) = 1
```

means that something is done with the element of array M with the index value 1.

Another example of a pattern is:

```
E   CONTAINS   [ P1 P2 P3 ]
```

An event X satisfies this pattern iff it is an event of the type E and at some level the event X is constituted exactly from the events that satisfy P1, P2 and P3 in the same order. For example,

```
eval-expression  CONTAINS [ V: variable & Source-text(V) = 'Y' ]
```

describes the evaluation of the expression which coincides with the single variable Y.

Patterns of the form:

```
X: E   &   Source-text(X) = 'A'
```

are used so often that an abbreviation is introduced. This pattern can be written as:

```
E   IS   A
```

3.2 Predicates, Functions and Aggregate operations

There are special pattern-matching predicates for events such as:

```
Event-set  SATISFIES  Pattern
```

Sequences, bags (multivalued sets) and sets of events and other values can be considered. The value of the *Prev-path* attribute is an example of a sequence of events. Common set-theoretic operations are available for such data structures.

Let P be an event pattern and F be any function or predicate. Also, let A be an event or event set. The *set constructor*

```
{ P   FROM   A   }
```

defines a loop over A. For example,

```
{ ex-read-stmt ::( file-name IS X ) FROM  execute-program }
```

yields a set of all events of the type "read from the file X" from the whole target program execution history.

```
{ P  FROM  A  APPLY   F  }
```

The events are chosen from `A` in the same manner as above, but this time the function `F` is applied to every chosen event immediately and the resulting set is obtained from values produced by the function `F`. For example,

```
{ex-assignment ::( destination CONTAINS[ N: variable])
                        FROM Prev-path(E) APPLY  Source-text(N) }
```

yields a set of variable names which have been assigned values before the event E appeared.

Let us also consider a *sequence* constructor:

```
[  P  FROM  A   ]
```

where `A` may be an event sequence, bag or set. The sequence constructor is defined *iff* all elements in the resulting set are ordered w.r.t. the relation <<. For this aggregate operation we may use the same generalizations as for the set constructor.

Let `A` be any sequence, bag (multivalued set) or set, and let `Op` be a commutative and associative binary operation defined for the elements of `A`. *Reduction* of `Op` over `A` is denoted by `Op/A.`

Let $[a_1, a_2,..., a_N]$ denote a sequence, bag or set containing elements $a_1, a_2,..., a_N$. Let the unity element for the `Op` operation be denoted by 1. Then:

$$Op/[] = 1$$
$$Op/[\ a\ ] = a$$
$$Op/[a_1,a_2,...,\ a_N] = a_1\ Op\ a_2\ Op...Op\ a_N$$

For instance, `+ / [ ]  = 0 ; +/[ 3 ]  = 3`, and `+ / [ 1, 3, 5 ]  = 9`

The following two forms are often used:

```
AND / [ Pattern FROM  A  APPLY  Pred ]
OR  / [ Pattern FROM  A  APPLY  Pred ]
```

where `Pred` is a predicate and '[' ']' stands for any aggregate operation. These expressions can be written in short form as follows:

```
  FOREACH  Pattern FROM  A  Pred
  EXISTS   Pattern FROM  A  Pred
```

4 Simple Examples of Computations Over Histories

The assertions about target program behavior can be written as an appropriate computations over H-space.

Example 1: *"All variables in the program must be initialized before being used in some expression".*

```
FOREACH  E: eval-expression FROM execute-program
 (    { V: variable FROM E APPLY Source-text(V) }  ⊆
```

```
{ D: destination FROM Prev-path(E) APPLY Source-text(D) } )
```

This means that the set of variables referenced in the expression should be a subset of the variables which earlier have been assigned values. Note that this assertion does not depend on some specific target program. It is an example of a ***universal assertion*** which must be true for arbitrary target program.

Example 2:*"After the execution of the subprogram P the value of variable X remains unchanged"*.

```
FOREACH E: call-procedure:: (proc-name IS P) FROM execute-program
        Value-at-begin(E, X) = Value-at-end(E, X)
```

4.1 Profiles and Performance Measurement

It is possible to use FORMAN expressions to obtain target program execution statistics, e.g.

```
CARD [ALL ex-stmt FROM execute-program]
```

which yields the number of statements actually executed. **ALL** means that nested events `ex-stmt` also should be counted.

In order to count the number of executions of a statement, labeled in the target program text by the label M, the FORMAN expression can be evaluated:

```
CARD [ ex-stmt :: (  label IS   M )    FROM execute-program ]
```

It is possible to express, for instance, the time spent in the procedures `A` execution by the expression:

```
+/ [D: call-procedure:: ( procedure-name IS A)
      FROM  execute-program  APPLY    Duration(D) ]
```

5 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 goal is to generate informative messages and provide user with some values obtained from the trace in order to detect and localize bugs. Rules can provide dialog with the user as well. An assertion is a boolean expression which may contain some of the notation described in the previous sections. An assertion in FORMAN deals with 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 rule has the form:

```
 assertion
      SAY (expression_1  expression_2  ...  expression_N)
      ONFAIL SAY (expression_N+1  expression_N+2  ...  expression_N+M )
```

In expressions both text constants and FORMAN operations and metavariables can be used. Either SAY clauses, either ONFAIL SAY, or both could be dropped.

The presence of metavariables in the assertion gives a possibility 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. Note that the texts of assertions and debugging queries are separated from the text of target program.

Example 1.

The following 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` is as follows:

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

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

```
TRUE
   SAY ( 'The history of variable X is:'
      [D: destination  IS X FROM execute_program  APPLY VALUE(D) ]  )
```

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 3 45 8 754 45567 0
```

Assertion 2. Let’s write and check the assertion: *"The value of variable X doesn't 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 message about assertion violation more informative we include the embracing event of the type `ex-stmt`. Metavariables `S` and `D` help to 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 metavariable when printed by 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 particular implementation are simply values of step counter.

Since we could expect that the assertion might be violated when executing Read statement, it makes sense to report the record number on 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 all 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 input record on which the variable X has been first time 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 2.

The following program is executed on the same input file `xx.txt` as the previous program.

```
program e3;
var REC,COUNT: integer;
    XX: file of text;
 begin
     COUNT:=1; (*** ERROR! Must be 0 ***)
     REC:= 1;
     reset (XX, 'xx.txt');
     while REC <> 0 do begin
                    COUNT:= COUNT + 1;
                    read(XX, REC) end;
 end.
```

Assertion 1: "The number of records read equals the value of COUNT at the end of program execution."

```
  CARD [ ex_read_stmt ::( file_name IS XX) FROM  execute_program ]
      = VALUE  ( AT END execute_program  COUNT)

   SAY( 'The value of COUNT at program execution end is '
         VALUE( AT END execute_program COUNT))
   SAY( 'It equals to the number of records processed')

   ONFAIL SAY( 'The value of counter COUNT is'
                VALUE(AT END execute_program COUNT))
          SAY( 'It differs from the number of records read:'
        CARD [ ex_read_stmt ::( file_name IS XX) FROM  execute_program ]);
```

When executed on the prototype the following output was produced.

```
   Assertion # 1 violation!
The value of counter COUNT is 16
It differs from the number of records read: 15
```

Assertion 2: "There exists a moment when 0 is assigned to the REC variable." The value of COUNT variable at this moment is shown.

```
EXISTS * S: ex_stmt CONTAINS ( X: destination IS REC)
                      FROM execute_program
                 VALUE( X ) = 0
     SAY ( 'the value of COUNT is' VALUE( AT END X COUNT) 'after stmt')
    SAY( S ) SAY(  'was executed');
```

The assertion checker provides the following output.

```
   Assertion # 2 checked successfully...

the value of COUNT is 16 after stmt
ex_stmt :> Read( XX , REC )    Time= 155 .. 162
was executed
```

Query 1. Get some statistics.

```
TRUE
   SAY( 'Total number of statements executed is:'
      CARD [ ALL ex_stmt FROM execute_program ] )
   SAY( 'Total number of Reads from file XX is:'
      CARD [ ex_read_stmt ::( file_name IS XX) FROM  execute_program ])
   SAY( 'Last value assigned to REC is:'
      VALUE( AT END execute_program REC) )
   SAY('The history of variable REC is:'
      [D: destination IS REC FROM execute_program APPLY VALUE(D)]
```

When executed on the prototype the following output was produced.

```
   Assertion # 3 checked successfully...

Total number of statements executed is: 34
Total number of Reads from file XX is: 15
Last value assigned to REC is: 0
The history of variable REC is: 1 11 5 3 7 8 9 3 13 2 3 45 8 754 45567 0
```

Example 3.

The following program calculates the greatest common divisor for two integers X and Y.

```
program e4;
var X,Y: integer;
 begin
      X:= 840;
      Y:= 625;
      while X<>Y do
               if X > Y  then X:= X - Y
              else   Y:= Y - X;
 end.
```

Assertion 1. Loop invariant: at the beginning of loop body execution always must be X > 0 and Y > 0.

```
FOREACH LB: ex_loop_body FROM execute_program
       VALUE( AT BEGIN LB X) >0 AND VALUE( AT BEGIN LB Y) >0

   ONFAIL SAY( 'X<=0 OR Y<=0 before loop body execution')
     SAY( 'Value of X is:' VALUE( AT BEGIN LB X))
     SAY( 'Value of Y is:' VALUE( AT BEGIN LB Y));
```

Since there is only one loop statement in the program we can refer to the execution event simply by `ex-loop-body`. In the case of multiple loop statements in the program text unique labels should be introduced in the source text, and event pattern should contain the event `label` in order to distinguish different loop statement execution event. Let's suppose that the loop statement in the program above has been marked by label L1. Then the assertion above could be rewritten in the following way.

```
FOREACH ex_stmt::( label IS L1 LOOP: ex-loop-stmt) FROM execute_program
   FOREACH LB: ex_loop_body FROM LOOP
       VALUE( AT BEGIN LB X) >0 AND VALUE( AT BEGIN LB Y) >0

   ONFAIL SAY( 'X<=0 OR Y<=0 before loop body execution')
     SAY( 'Value of X is:' VALUE( AT BEGIN LB X))
     SAY( 'Value of Y is:' VALUE( AT BEGIN LB Y));
```

When run on the assertion checker prototype this assertion is satisfied.

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

Query 2. To get statistics about program execution history we can write the following rule.

```
TRUE
   SAY( 'Total number of statements executed is:'
      CARD [  ALL ex_stmt FROM execute_program ] )
   SAY( 'Total number of loop iterations executed is:'
       CARD [ ex_one_iteration FROM execute_program ])
   SAY( 'The history of X is:')
   SAY( [ S: destination IS X FROM execute_program
            APPLY VALUE( AT END S X) ])
   SAY( 'The history of Y is:')
   SAY( [  S: destination IS Y  FROM execute_program
            APPLY VALUE( AT END S Y) ])
```

After execution we obtain:

```
 Assertion # 2 checked successful...

Total number of statements executed is: 35
Total number of loop iterations executed is: 17
The history of X is:
840 215 20 5
The history of Y is:
625 410 195 175 155 135 115 95 75 55 35 15 10 5
```

Example 4.

This example demonstrates how assertions and debugging queries could be used in order to detect an error in the program. The following program is intended to perform binary search in the integer array.

```
program Binary_Search;
   var   from, to, target, mid, top, bottom: integer;
         found: boolean;
         A: array [ 1.. 10] of integer;
begin
   from:= 1;  to:= 10;

(* Seed the array *)
   A[1] := 3;          A[2] := 6;
   A[3] := 19;         A[4] := 43;
   A[5] := 58;         A[6] := 112;
   A[7] := 367;        A[8] := 390;
   A[9] := 427;        A[10] := 471;

   target := 367;

   found:= FALSE;
```

```
     bottom := from;
     top := to;
     mid := (top + bottom) div 2;

     while bottom < top do
            begin
               if target < A[mid]        (**** ERROR: must be <= *)
                          then top := mid
                          else bottom := mid +1 ;
               mid := (top + bottom) div 2
            end;
    if A[mid] = target then found := TRUE;
  end.
```

The following rules and queries may be useful for the detection of the program bug.

Assertion 1. Searching intervals have to decrease. This example demonstrates that value which can not be obtained directly from the history could be evaluated in some auxiliary computation over history.

```
FOREACH LP: ex_loop_body  FROM execute_program
     VALUE(AT BEGIN LP (top - bottom)) >  VALUE(AT END LP (top - bottom))
    SAY( 'The history of search intervals (top - bottom) is:'
       [  LP1: ex_loop_body  FROM execute_program  APPLY
                                 VALUE( AT END LP1 (top - bottom)) ] )
    ONFAIL SAY (   'Searhing interval does not decrease.'
               'Value before is:' VALUE(AT BEGIN LP (top - bottom))
               'Value after is:' VALUE(AT END LP (top - bottom)) ) ;
```

The following message was produced by our assertion checker prototype when running the Binary_Search program with the assertions described above.

```
   Assertion # 1 checked successfully...
The history of search intervals (top - bottom) is: 4 2 0
```

*Query 2***.** The following query provides histories of some variables.

```
TRUE
    SAY( 'The number of iterations is:'
       CARD[  ex_loop_body  FROM execute_program ])
    SAY( 'The history of variable bottom  is:'
      [D: destination IS bottom  FROM execute_program  APPLY  VALUE(D) ] )
    SAY( 'The history of variable mid  is:'
     [D:destination IS mid  FROM execute_program  APPLY VALUE(D) ] )
    SAY( 'The history of variable top is:'
     [D: destination IS top  FROM execute_program  APPLY VALUE(D) ] )
    SAY( 'The value of variable found is:'
          VALUE(AT END execute_program found));
```

The result of computation over trace.

```
 Assertion # 2 checked successfully...
```

```
The number of iterations: 3
The history of variable bottom is: 1 6 8
The history of variable mid is: 5 8 7 8
The history of variable top is: 10 8
The value of variable Found at the end of execution is: FALSE
```

The query is a convenient mean to present the result of execution even if the target program does not provide any output.

Assertion **3.** There was a moment in the execution history when the array element under consideration `A[mid]` and the `target` were equal. The loops corresponding to the EXISTS quantifiers are interrupted when the condition is satisfied and the value of metavariable S could be used for message output.

```
      EXISTS  S: ex_stmt FROM  execute_program
         (VALUE( AT BEGIN S target) = VALUE( AT BEGIN S A[mid]))
  SAY( 'Equality of A[mid] with target found at the beginning of event' )
  SAY( S )
  SAY( 'when mid =' VALUE( AT BEGIN S mid))
  SAY( 'bottom =' VALUE( AT BEGIN S bottom) )
  SAY( 'top =' VALUE( AT BEGIN S top) )
```

This last message may provide the insight for localizing the bug.

```
    Assertion #3 checked successfully...

Equality of A[mid] with target found at the beginning of event
ex_stmt :> if ( target < A[ mid ] ) then top := mid ;
            else bottom := ( mid + 1 ) ;    Time= 180 .. 189
when mid = 7
bottom = 6
top = 8
```

6 Libraries of Universal Assertions and Debugging Rules

The following example demonstrates how the knowledge about typical behavior can be formalized in FORMAN. Let us consider a common notion such as a counter. Usually the history of variable X used as a counter looks like:

```
LOOP ...    X := 0;...
            LOOP ...
                  X := X + const;...
            endloop;...
endloop;
```

The predicate `Is-counter` checks whether a variable X is used in the target program according to this pattern.

```
 Is-counter( X )  =
```

```
 [ ex-righthand-part FROM
        [ex-assignment ::( destination IS  X ) FROM  ex-program ] ]

 SATISFIES

 (   R: ex-righthand-part & Value(R) =  0
     ( ex-righthand-part CONTAINS
               [eval-bin-op ::(eval-1st-arg IS X
                eval-2nd-arg CONTAINS [constant]
                perform-bin-op IS  '+' )]
     )+
   )+
```

Such predicates can detect anomalies in the behavior of the counter at the run-time. It is possible to define additional debugging rules that will detect particular violations in the usage of variable declared as a counter, for instance, not assigning 0 to the counter at the beginning of the history. The following rule can check for typical bug when using counter X:

```
[ ex-righthand-part FROM
      [ ex-assignment ::( destination IS  X ) FROM  ex-program ] ]

SATISFIES

(R: ex-righthand-part & Value(R)<> 0     ex-righthand-part *  )

 SAY ( 'Initialisation of counter is missing in the trace'
      'first value assigned is' Value(R)
      'computed from the expression' R )
```

Packages of predicate definitions and debugging rules can be designed for a number of common data structures such as stack, queue, list etc.

Debugging rules can be considered as a mean for 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 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 participating in the expression.

6.1 Integrating Algorithmic Debugging with Evaluation of Operational Assertions

The Algorithmic Debugging paradigm [27], [13], [26] can be expressed informally in our terms as the following rule:

```
Rule A:
IF    some assertion about the event
            call-procedure::( P: procedure-name  B: ex-proc-body)
      is violated THEN
```

```
      FOREACH event of type call-procedure from B    apply rule A;
      IF all of these rule applications are successful
       (i.e. no assertions in the procedures calls were violated) THEN
      REPORT error is localized in the body of procedure P
end of Rule A;
```

Some formalism for describing similar rules remains to be developed in the framework of FORMAN. One of our goals is to formalize, for instance, debugging strategies presented in [25], [16]. Algorithmic Debugging with assertions in Prolog is presented in [11].

7 Principles of Implementation

There are two separate tasks to be done when performing any computations over H-space.

- To generate the event set (the trace).
- To perform necessary computations over it.

For the efficiency reasons these two steps could be combined and optimized. There are at least two possible implementation approaches.

- Instrumentation of the target program source code.
- Instrumentation at the target program implementation level (e.g. design of a specialized interpreter for the target language).

Event grammar is designed in such a way that each event can be detected by appropriate target program instrumentation which can be performed automatically. The instrumented target program generates the trace. Only the data necessary to perform computations in a given FORMAN program are put on the trace. Some computations over traces may be performed at the target program run time, but the rest is postponed to the post-mortem trace analysis. Some technique for *incremental target program execution* still has to be developed to provide a basis for efficient FORMAN implementation.

In general, several passes through the trace may be necessary in order to perform all computations required in the FORMAN expression. But in many real cases appropriate optimizations can reduce the number of passes.

The following example demonstrates how an appropriate instrumentation of the target program can reduce the run-time overhead. Let us have to check the following assertion: *"Each variable has to be assigned a value before it is used in an expression evaluation"*

```
 FOREACH * E: eval-expression CONTAINS ( V: variable )
                            FROM execute-program
    EXISTS D: destination FROM prev-path( E )
       source-text(D) = source-text(V)
```

The straightforward solution requires at least two passes through the trace. A more clever instrumentation could introduce a bitmap array where each variable in the target program is represented by a single bit. Appearance of a destination event in the trace causes the corresponding bit to be set on. For each `eval-expression` event the corresponding bits have to be checked up, so the whole assertion can be evaluated in a single pass through the trace.

Prototype implementation of assertion checker for a subset of PASCAL runs on SUN/UNIX platform. We have implemented our own PASCAL subset interpreter for this purpose. The prototype is implemented using the RIGAL language [1] which is a convenient tool for language processor rapid prototyping.

8 Conclusions and Future Work

The approach, outlined above, is based on the notion of computation over target program execution history. It provides uniform language means for assertion checking, debugging queries, and profiles and performance measurements. The implementation could be based on automatic target program instrumentation and after run-time trace analysis.

The FORMAN approach may be applied to a broad variety of target programming languages including PASCAL, C, and others.

The text of FORMAN program describing some computations over target program traces can be maintained separately from the text the target program. This allows to accumulate formalized knowledge about the program and about the whole target language in separate libraries. Different aspects of program behavior can be described in different assertion files. This solves the problem of selective assertion checking, since by grouping assertion files it is possible to control the amount of assertions to be checked.

Universal assertions and formalized debugging rules and strategies may be integrated with existing debugging tools.

This paper provides an overview of basic principles and examples. The prototype implementation gave the possibility to experiment with assertions and debugging queries and to evaluate some of the implementation techniques. Topics for future work include assertion language tuning, in particular, extension of the event grammar by user-defined events, optimizations of trace collecting and analysis methods, integrating of assertion checker into traditional debugging environment, and the design of methodology for assertion and debugging query writing.

References

[1] Auguston M., Programming language RIGAL as a compiler writing tool, ACM SIGPLAN Notices, December 1990, vol.25, #12, pp.61-69

[2] Auguston M.: FORMAN - A Program Formal Annotation Language, In Proceedings of the 5:th Israel conference on Computer Systems and Software Engineering, Gerclia, May 1991, IEEE Computer Society Press, pp.149-154, 1991.

[3] Auguston M., A Language for Debugging Automation, in Proceedings of 6th Intl Conference on Software Engineering and Knowledge Engineering SEKE’94, Jurmala, 1994, pp.108-115

[4] M. Auguston, P. Fritzson, PARFORMAN - an Assertion Language for Specifying Behavior when Debugging Parallel Applications. in Proceedings of the Euromicro Workshop on Parallel and Distributed Processing, Gran Canaria, Jan 27-29, 1993. IEEE Computer Society Press.

[5] F.Baiardi, N. De Francesco, G. Vaglini: Development of a Debugger for a Concurrent Language, IEEE Transactions on Software Engineering, vol. SE-12, No 4, pp.547-553, April 1986.

[6] Peter C. Bates: Debugging Heterogeneous Distributed Systems Using Event-Based Models of Behavior: SIGPLAN Notices vol. 24, No 1, pp.11-22, 1989.

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

[8] B.Bruegge, P.Hibbard: Generalized Path Expressions: A High-Level Debugging Mechanism, The Journal of Systems and Software 3, pp. 265-276, 1983.

[9] B.Bruegge, T.Gottshalk, B. Luo, A Framework for Dynamic Program Analyzers, 8th Annual Conference on Object-Oriented Programming Systems, SIGPLAN Notices, vol. 28, #10, Oct. 1993, pp.65-82.

[10] R.H. Campbell, A.N. Habermann: The specification of process synchronization by path expressions, Lecture Notes in Computer Science, vol. 16, pp. 89-102, 1974.

[11] W.Drabent, S. Nadjm-Tehrani, J.Maluszinski, Algorithmic Debugging with Assertions, in: Meta-Programming in Logic programming (eds. H.Abramson, M.H.Rogers), MIT Press, 1989, pp.501-521.

[12] P. Fritzson, M. Auguston, N. Shahmehri: Using Assertions in Declarative and Operational Models for Automated Debugging, Journal of Systems and Software, v.25, #3, June 1994, pp.223-239.

[13] P. Fritzson, T. Gyimothy, M. Kamkar and N. Shahmehri: Generalized Algorithmic Debugging and Testing. In Proceedings of the ACM PLDI’91, SIGPLAN Notices, June 1991,pp 317-326.

[14] G.Goldszmidt, S.Katz, S.Yemini: Interactive Blackbox Debugging for Concurrent Languages, SIGPLAN Notices vol.24, No 1, pp. 271-282, 1989.

[15] G.Goldszmit, S.Yemini, S.Katz, High-Level Language Debugging for Concurrent programs, ACM Transactions on Computer Systems, vol.8, No.4, november 1990, pp.311-336.

[16] Korel B., PELAS- Program Error Locating Assistant System, IEEE Transactions on Software Engineering, vol.14, #9, 1988, pp.1253-1260

[17] D. C. Luckham, D.Bryan, W.Mann, S.Meldal, D.P.Helmbold, An Introduction to Task Sequencing Language, TSL version 1.5 (Preliminary version), Stanford University, February 1, 1990, pp. 1-68.

[18]D.C.Luckham, J.J.Kenney, L.M.Augustin, J.Vera, D.Bryan, W.Mann, Specification and Analysis of System Architecture Using Rapid, IEEE Transactions on Software Engineering, vol.21, No.4, April 1995, pp.336-355.

[19] David C. Luckham, Sriram Sankar, Shuzo Takahashi: Two-Dimensional Pinpointing: Debugging with Formal Specifications. IEEE Software, January 1991, pp.74-84.

[20] D.Luckham, J.Vera, D.Bryan, L.Augustin, F.Belz, Partial Ordering of Event sets and Their Application to Prototyping Concurrent, Timed Systems, Journal of Systems and Software, vol 21, 1993, pp.253-265

[21] J.McLean, A Formal Method for the Abstract Specification of Software, Journal of the ACM, vol.31, #3, July 1984, pp.600-627.

[22] 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.

[23] D.Rosenblum, Specifying Concurrent Systems with TSL, IEEE Software, May 1991, pp.52-61.

[24] D.Rosenblum, A Practical Approach to Programming with Assertions, IEEE Transactions on Software Engineering, vol.21, #1, 1994, pp.1253-1260

[25] R.Seviora, Knowledge-Based Program Debugging Systems, IEEE Software, vol.4, #3, May 1987, pp.20-32

[26] N.Shahmehri, *Generalized Algorithmic Debugging*. Ph.D. Thesis No 260, Dept. of Computer and Information Science, Linköping University, S-581 83 Linköping, Sweden, 1991.

[27] E.Shapiro. Algorithmic Program Debugging. MIT Press, May 1982.

[28] T.Winograd, Aleph: a system specification language. 1988, Stanford University, CSLI.