New directions in software testing automation, test oracle design, and safety assessment

Auguston, Mikhail
Monterey, California, Naval Postgraduate School
New Directions in Software Testing Automation, Test Oracle Design, and Safety Assessment

Mikhail Auguston

Computer Science Department
Naval Postgraduate School, Monterey, California
maugusto@nps.edu
Software testing is a challenge...

- The goal of testing is to expose yet undiscovered errors. A successful test is one that “catches” an error...

- Testing cannot prove the correctness of a software; it can only demonstrate the presence of a bug...

Example: a program that is intended to check the equality of three input numbers.

\[
\text{if } (a + b + c)/3 == a \text{ then print “equal”}
\]
\[
\text{else print “not equal”}
\]

- Exhaustive testing is practically unfeasible (the number of executable paths may be astronomical)

- Testing and debugging of a software system requires more than 50% of total time and effort (Brooks’ rule)
Black Box Testing

The main problems:

🔹 How to create test cases
🔹 How to run a test case
🔹 How to verify the results of a test run
Black Box testing

Environment

The SUT may be a complex reactive real-time C3I system
Outlook of this presentation

Chapter 1. Automated test generation based on environment models (How to create test cases)

Chapter 2. Software safety assessment

Chapter 3. Implementation (How to run a test case)

Chapter 4. Program monitoring and test oracles (How to verify the results of a test run)
Chapter 1
Automated test generation based on environment models
(How to create test cases)
Testing methodology

- Test cases should be carefully designed using “white box” (e.g., branch coverage) or “black box” (e.g., equivalence partition, boundary conditions) methods. This is like “sharp-shooting” for bugs...

- Test cases may be generated at random. This is like a “machine gun” approach...

- We suggest an “intelligent” random generation based on the environment models. It is best suited for a very special class of programs: reactive and real-time. These programs are of special interest for DoD-related applications.
The model of environment
(a novel approach to behavior modeling)

An event is any detectable action that is executed in the “black box” environment

- An event is a time interval
- An event has attributes; e.g., type, timing attributes, etc.
- There are two basic relations for events: precedence and inclusion
- The behavior of environment can be represented as a set of events (event trace)
1) Mutual exclusion of relations
   a \text{ PRECEDES} b \rightarrow \text{not} (a \text{ IN} b)
   a \text{ IN} b \rightarrow \text{not} (a \text{ PRECEDES} b)

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)

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)

Both \text{PRECEDES} and \text{IN} are \text{irreflexive partial orderings}
The model of environment

Usually event traces have a certain structure (or constraints) in a given environment.

Examples:
1. **Shoot_a_gun** is a sequence of a **Fire** event followed by either a **Hit** or a **Miss** event.
2. **Driving_a_car** is an event that may be represented as a sequence of zero or more events of types **go_straight**, **turn_left**, **turn_right**, or **stop**.
The model of environment

The structure of possible event traces for a given environment can be specified using event grammar

1. \texttt{Shoot\_a\_gun}::= \texttt{Fire} ( \texttt{Hit} | \texttt{Miss} )
   \texttt{Shooting}::= \texttt{Shoot\_a\_gun} *

2. \texttt{Driving\_a\_car}::= 
   \texttt{go\_straight}
   ( \texttt{go\_straight} | \texttt{turn\_left} | \texttt{turn\_right} ) *
   \texttt{stop}
   \texttt{go\_straight}::= ( \texttt{accelerate} | \texttt{decelerate} | \texttt{cruise} )
Sequential and parallel events

The precedence relation defines the partial order of events.

Two events are not necessarily ordered; i.e., they can happen concurrently.

Examples

\[
\text{Shoot\_a\_gun} ::= \text{Fire ( Hit | Miss )}
\]

\[
\text{Shooting} ::= (* \text{Shoot\_a\_gun} *)
\]

\[
\text{Shooting\_Competition} ::= (* \text{Shooting} *)
\]

This is a sequence.

Those events may be parallel.
Visual representation of event trace (not all events and relations are shown...)

- IN relation
- PRECEDES relation
Event attributes

Shoot_a_gun ::= Fire (Hit /Shoot_a_gun.points = Rand[1..10];
  ENCLOSING Shooting.points += Shoot_a_gun.points; /
  Miss /Shoot_a_gun.points = 0;)

Shooting ::= / Shooting.points = 0; /
  (* Shoot_a_gun
     /Shooting.ammo -=1; *) While (Shooting.ammo > 0)

Shooting_Competition ::= /num = 0; /
  { /* /Shooting.id = num++; 
     Shooting.ammo = 10; */
     Shooting */} (Rand[2..100])
Attribute event grammars (AEG) are intended to be used as a vehicle for automated random event trace generation.

It is assumed that the AEG is traversed top-down and left-to-right and only once to produce a particular event trace.

Randomized decisions about what alternative to take and how many times to perform the iteration should be made during the trace generation.

Attribute values are evaluated during this traversal.
Using AEG to generate event traces and inputs to the SUT

We can provide the probability of selecting an alternative

\[
\text{Shoot\_a\_gun ::= Fire ( P(0.3) Hit } \\
\text{ Send\_input\_to\_SUT( ENCLOSING Shooting .id, Hit .time);/ | } \\
\text{ -- this simulates SUT sensor input } \\
\text{ P(0.7) Miss )}
\]

We can generate a large number of event traces satisfying the constraints imposed by the event grammar
The grammar can be used in order to generate event traces and SUT inputs, for example:

- Shooting_Competition:

  - Shooting: Shoot_a_gun:
    - Fire
      - Hit
        - /Send_SUT_input( Hit.time )/
    - Miss

- Timeline:

  - Shoot_a_gun: Fire
  - Shoot_a_gun: Fire
  - Shoot_a_gun: Fire
    - Hit
      - /Send_SUT_input( Hit.time )/
Use cases

- Event traces are essentially use cases

- Examples of event traces can be useful for requirements engineering, prototyping, and system documentation
Example when SUT outputs are incorporated into the environment model

\text{Attack::= \{\ast \text{Missile\_launch} \ast\}\ (\text{Rand}[1..5])}
\text{Missile\_launch::= boost \text{middle\_stage} \text{When(middle\_stage.completed)} \text{Boom}}
\text{middle\_stage::= \text{middle\_stage.completed} = true; /}
  \text{(* CATCH interception\_launched (hit\_coordinates)}
  \text{ -- this external event intercepts SUT output}
  \text{When (hit\_coordinates == middle\_stage\_coordinates)}
  \text{[ P(0.1) hit\_hard}
  \text{ / middle\_stage\_completed= false;}
  \text{send\_SUT\_input(middle\_stage\_coordinates);}
  \text{ -- this simulates SUT sensor input}
  \text{Break; / -- breaks the iteration}
\text{]}
\text{OTHERWISE move}
  \text{*)}
\text{move ::= /adjust (ENCLOSING middle\_stage\_coordinates)}
  \text{send\_SUT\_input( ENCLOSING middle\_stage\_coordinates);}
  \text{ -- this simulates SUT sensor input}
\text{DELAY(50 msec); /}
Chapter 2
Software safety assessment
In the previous example, the Boom event will occur in certain scenarios depending on the SUT outputs received by the test driver and random choices determined by the given probabilities.

If we run large enough number of (automatically generated) tests, the statistics gathered gives some approximation for the risk of getting to the hazardous state. This becomes a very constructive process of performing experiments with SUT behavior within the given environment model.
Qualitative Risk Analysis

- The environment model can contain description of hazardous states in which system could arrive, and which can not be easily retrieved from SUT requirements specifications.
- We can do qualitative analysis as well... It is possible to ask questions, such as “what has contributed to this outcome?”
- We can change some probabilities in the environment model, or change some parameters in the SUT and repeat the whole set of tests. If the frequency of reaching a hazardous state changes, we can answer the question asked...
- The changes in the model could be done automatically in a some systematic way.
Qualitative Risk Analysis

Attack ::= \{ Missile_launch \} * (<=N)
Missile_launch ::= boost middle_stage Boom
middle_stage ::= (CATCH interception_launched(hit_coordinates))

-- this external event intercepts SUT output

[ P(p1) hit_hard
  /send_hit_input(middle_stage.coordinates);
  Break; ]
OTHERWISE move
)

Experimenting with increasing or decreasing N and p1 we can conclude what impact those parameters have on the probability of a hazardous outcome, and find thresholds for SUT behavior in terms of N and p1 values.
Chapter 3
Implementation
(How to run a test case)
How it works

Environment model represented as an event grammar

How to create test cases

Test driver (in C or assembly language)

Generator

Run time monitor

How to run test case

How to monitor the results

SUT
Prototype implementation outlook

The first automated test generator based on attribute event grammars has been implemented at NPS. It takes an AEG and generates a test driver in C.

Some highlights:
- Parallel event threads are implemented by interleaving.
- Attributes are evaluated mostly at the generation time, but those dependent on SUT outputs (on CATCH clauses) are postponed until the run time.
- The driver contains only simple assignment statements and C subroutine calls for interface with the SUT, guarded by simple flags, hence is very efficient and can be used for real time SUT testing.
int main() {
    /* declarations */

    ........

    /************** test drivers body *************/

    /* Time stamp 5 */
    /* start iteration */
    iteration_17 = 1;

    ......................

    /* Time stamp 6 */
    if (iteration_17)
        catch__18 = launch_interception(&hit_coordinates);

    ......................

    /* Time stamp 7 */
    if (iteration_17 && !catch__18)
        Middle_stage_1_coordinates_16 = 19;

    ......................

    /* Time stamp 18 */
    if (iteration_42 && catch__46 && when_47)
        /* break the iteration */
        iteration_42 = 0;

    ......................
The main advantages

- The whole testing process can be automated
- The AEG formalism provides powerful high-level abstractions for environment modeling
- It is possible to run many more test cases with better chances to succeed in exposing an error
- It addresses the regression testing problem - generated test drivers can be saved and reused.
- AEG is well structured, hierarchical, and scalable
- The environment model itself is an asset and could be reused
Why it will fly

- Environment model specified by AEG provides for **high-level domain-specific formalism** for testing automation.
- The generated test driver is **efficient** and could be used for real-time test cases.
- Different environment models can be designed; e.g., for **testing extreme scenarios** by increasing probabilities of certain events, or for **load testing**.
- Experiments running SUT with the environment model provide a constructive method for quantitative and even qualitative **software safety assessment**.
- Environment models can be designed on early stages of system design, can provide environment simulation scenarios or **use cases**, and can be used for tuning the **requirements** and for **prototyping efforts**.
Frequently Asked Questions

Q: How to design the environment model?
A: It is similar to the OOA/OOD process. We analyze use cases, requirements, apply Abbot’s method, interview experts about environment behavior and related attributes, then gradually build the model. The generator may be helpful in order to test and debug the model: we can generate traces and verify them. The strongly hierarchical nature of event grammar also helps. The good news is that the environment model could be reused.

Q: Can the model capture synchronization events?
A: Yes, an event (e.g., the synchronization event) can be shared by two or more other events. For example, two cars can be represented by two parallel event threads; if they collide, both threads share the collision event.
Potential topics for future work (plenty…)

- For the interface between the test driver and the SUT a special set of wrappers or bridges should be provided.
- The test driver generator can enforce grammar branch coverage to ensure that all grammar alternatives have been traversed (a good candidate for test metrics).
- The generated test driver can receive inputs from the SUT, or even from the user; i.e., could implement an interactive test case.
- The generated test driver can interact also with the test oracle or the run time monitor to support the integrity of the testing process.
- Automated software safety assessment both quantitative and qualitative.
- Environment models can be reused.
Chapter 4
Program monitoring and test oracles

(How to verify the results of a test run)
Objective: to develop unifying principles for program monitoring activities

Suggested solution: to define a precise model of program behavior as a set of events - event trace

Monitoring activities in software design can be implemented as computations over program execution traces.

Examples:
- Assertion checking (test oracles)
- Debugging queries
- Profiles
- Performance measurements
- Behavior visualization
Program Behavior Models

Program monitoring activities can be specified in a uniform way using program behavior models based on the event notion.

An event corresponds to any detectable action; e.g., subroutine call, expression evaluation, message passing, etc. An event corresponds to a time interval.

Two partial order binary relations are defined for events: precedence and inclusion.

An event has attributes: type, duration, program state at beginning or end of the event, value,...
Program Behavior Models

- **Event grammar** specifies the constraints on configurations of events generated at the run time (in the form of axioms, or “lightweight semantics” of the target language)

- Some axioms are generic; e.g., transitivity and distributivity

\[
A \text{ PRECEDES } B \text{ and } B \text{ PRECEDES } C \implies A \text{ PRECEDES } C
\]

\[
A \text{ IN } B \text{ and } B \text{ PRECEDES } C \implies A \text{ PRECEDES } C
\]
Example of an Event Grammar

\[
\begin{align*}
\text{ex\_prog} &::= \text{ex\_stmt} \,* \\
\text{ex\_stmt} &::= \text{ex\_assignmt} \mid \text{ex\_read\_stmt} \mid \ldots \\
\text{ex\_assignmt} &::= \text{eval\_expr} \ \text{destination}
\end{align*}
\]
Program Monitoring

- Monitoring activities: assertion checking, profiles, performance measurements, dynamic QoS metrics, visualization, debugging queries, intrusion detection
- Program monitoring can be specified in terms of computations over event traces
- We introduce a specific language FORMAN to describe computations over event traces (based on event patterns and aggregate operations over events)
FORMAN language

Event patterns

- \[ x: \text{func\_call} \& x.\text{name} == \text{"A"} \]
- \[ \text{eval\_expr} :: ( \text{variable} ) \]

List of events

- \[ [ \text{exec\_assignmt} \text{ FROM ex\_prog} ] \]

List of values

- \[ [ x: \text{exec\_assignmt} \text{ FROM ex\_prog APPLY x.value} ] \]
FORMAN language

Aggregate Operations

MAX/\[ x: exec_assignmt FROM ex_prog APPLY x.value]\]

AND/\[ x: exec_assignmt FROM ex_prog APPLY x.value > 17]\]

Or

FOREACH x: exec_assignmt FROM ex_prog x.value > 17
Examples

1) Profile

SAY( "Number of function A calls is "
    CARD[ x: func_call & x.name == "A"
        FROM ex_prog ]
)

2) Generic debugging rule (typical error description)

FOREACH e: eval_expr :: (v: variable)
    FROM ex_prog
    EXISTS d: destination FROM e.PREV_PATH
    v.source_code = d.source_code
    ONFAIL SAY("Uninitialized variable "
        v.source_code "is used in expression " e)
Examples

3) Debugging query
SAY("The history of variable x 
[d: destination & d.source_code == "x" FROM ex_prog
  APPLY d.value ]

4) Traditional debugging print statements
FOREACH f: func_call & f.name == "A"
  FROM ex_prog
  f.value_at_begin(
    printf("variable x is %d\n", x)
  )
Example of event trace representing a synchronization event (send/receive a message)

par -- launches two parallel processes
seq -- first parallel thread
stmt1
channel1 ! Out-expr -- sends a message
...
seq -- another parallel thread
stmt2
channel1 ? Var -- receives a message
...

[Diagram showing event trace with parallel threads and synchronization events]
Program visualization (UFO project)

Visualization prototype for Unicon/ALAMO (Jointly with C. Jeffery, NMSU)

Point plot example for a binary search program
The novelty claims of our approach

- **Uniform framework** for program monitoring based on precise behavior models and event trace computations
- Computations on the event traces can be implemented in a **nondestructive** way via automatic instrumentation of the source code or even of the executables (Dyninst approach)
- Can specify **generic trace computations**: typical bug detection, dynamic QoS metrics, profiles, visualization, ...
- Both **functional** and **non-functional** requirements can be monitored
- Yet another approach to the **aspect-oriented** paradigm
Accomplished projects and work in progress

- Assertion checker for a Pascal subset (via interpreter)
- Assertion checker for the C language (via source code instrumentation)
- Assertion checker and visualization tool for the Unicon language (via Virtual Machine monitors)
- Dynamic QoS metrics, UniFrame project (via glue and wrapper instrumentation), funded by ONR
- Intrusion detection and countermeasures (via Linux kernel library instrumentation using NAI GSWTK), funded by the Department of Justice Homeland Security Program
- Ongoing project: C/C++ program monitoring (via Dyninst/DPCL toolset), achieved performance is adequate for monitoring real size programs
- Automated test driver generator for reactive real time systems based on AEG environment models, funded by Missile Defense Agency
Some publications


Summary of the event grammar approach

- Behavior models based on event grammars provide a uniform framework for software testing and debugging automation.
- Can be implemented in a nondestructive way via automatic instrumentation.
- Automated tools can be built to support all phases of the testing process.
- Provides a good potential for reuse: environment models, generic debugging rules, test drivers for regression testing.
- Provides high-level abstractions for testing and debugging tasks, hence is easy to learn and use.
- Well suited for reactive real-time system testing.
Why bother?

Testing and debugging consume more than **50%** of total software development cost.

If the proposed research is transferred into practice and reduces costs by **1%** of the **50%** of the $400 billion software industry, the potential economic impact would be around **$2 billion** per year.
Questions, please!
Backup slides
Example – simple calculator environment model

Use_calculator: (* Perform_calculation *);
Perform_calculation:
  Enter_number Enter_operator Enter_number
  WHEN (Enter_operator.operation == '+')
    / Perform_calculation.result =
      Enter_number[1].value + Enter_number[2].value;
  ELSE
    / Perform_calculation.result =
      Enter_number[1].value - Enter_number[2].value;
[ P(0.7) Show_result ];
Example – simple calculator environment model

Enter_number:  / Enter_number.value= 0; /
  (* Press_digit_button
       / Enter_number.digit = RAND[0..9];
       Enter_number.value =
       Enter_number.value * 10 + Enter_number.digit;
   enter_digit(Enter_number.digit); / *) Rand[1..6];

Enter_operator:
  ( P(0.5) / enter_operation('+');
   Enter_operator.operation= '+'; / |
  P(0.5) / enter_operation('-');
   Enter_operator.operation= '-'; / );

Show_result:  /show_result();/ ;
Example 2 – Infusion Pump model

CARA_environment: { Patient, LSTAT, Pump };

Patient: / Patient.bleeding_rate= BR; /

(* / Patient.volume +=
   ENCLOSING CARA_environment ->
   Pump.Flow - Patient.bleeding_rate;

Patient.blood_pressure =
   Patient.volume/50 - 10;

Patient.bleeding_rate += RAND[-9..9]; /
   WHEN (Patient.blood_pressure > MINBP)

Normal_condition

ELSE

Critical_condition

*) [EVERY 1 sec] ;
Example 2 - Infusion Pump model

LSTAT: Power_on / send_power_on(); /
    (* / send_arterial_blood_pressure(ENCLOSING CARA_environment->Patient.blood_pressure); /
    *) [EVERY 1 sec];

Pump: Plugged_in
    / send_plugged_in();
    Pump.rotation_rate = RR;
    Pump.voltage = V; /
    { Voltage_monitoring, Pumping };
Example 2 - Infusion Pump model

**Voltage_monitoring:**

(* / ENCLOSING Pump.EMF_voltage =
  ENCLOSING Pump.rotation_rate * REMF;
  send_pump_EMF_voltage(
    ENCLOSING Pump.EMF_voltage);
*) [EVERY 5 sec] ;

**Pumping:**

(* / ENCLOSING Pump. rotation_rate =
  ENCLOSING Pump. voltage * VRR;
  ENCLOSING Pump. flow =
  ENCLOSING Pump. rotation_rate * RRF; /
  CATCH set_pump_voltage(ENCLOSING Pump.voltage)
  Voltage_changed
  [ P(p1) Occlusion
    / ENCLOSING Pump.occlusion_on = True;
    send_occlusion_on(); / ]
  WHEN (ENCLOSING Pump.occlusion_on)
  [ P(p2) / ENCLOSING Pump.occlusion_on =False;
    send_occlusion_off(); / ]
*) [EVERY 1 sec] ;