New Directions in Software Quality Assurance Automation

Mikhail Auguston

Computer Science Department
Naval Postgraduate School, Monterey, California
maugusto@nps.edu
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

The SUT may affect the inputs
Testing methodology

- We suggest (pseudo-)random test 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
(an 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**)
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. \( \text{Shoot}_\text{a}_\text{gun} ::= \text{Fire} ( \text{Hit} \mid \text{Miss} ) \)
   \( \text{Shooting} ::= \text{Shoot}_\text{a}_\text{gun} * \)

2. \( \text{Driving}_\text{a}_\text{car} ::= \)
   \( \text{go}_\text{straight} \)
   \( ( \text{go}_\text{straight} \mid \text{turn}_\text{left} \mid \text{turn}_\text{right} ) * \)
   \( \text{stop} \)
   \( \text{go}_\text{straight} ::= ( \text{accelerate} \mid \text{decelerate} \mid \text{cruise} ) \)
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

Shoot_a_gun ::= Fire ( Hit | Miss )
Shooting ::= (* Shoot_a_gun *)
Shooting_Competition ::= {* 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

- Shooting_Competition
  - Shooting
    - Shoot_a_gun
      - Fire
      - Miss
    - Fire
    - Miss
  - Shoot_a_gun
    - Fire
    - Hit
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])
Production grammars

- 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

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

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:

1. Shooting: Shoot_a_gun: Fire /Send_SUT_input( Hit.time )/
2. Shooting: Shoot_a_gun: Fire Hit /Send_SUT_input( Hit.time )/
3. Shoot_a_gun: Fire Miss
4. Shoot_a_gun: Fire Miss
5. Shoot_a_gun: Fire Hit

Timeline
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

\begin{verbatim}
Attack ::= \{ * Missile_launch * \} (Rand[1..5])
Missile_launch ::= boost middle_stage WHEN(middle_stage.completed) Boom
middle_stage ::= / middle_stage.completed = true; /
    (* CATCH interception_launched (hit_coordinates) -- this external event intercepts SUT output
        WHEN (hit_coordinates == middle_stage.coordinates )
            [ P(0.1) hit_hard
                / middle_stage.completed= false;
                send_SUT_input(middle_stage.coordinates);
                -- this simulates SUT sensor input
                Break; / -- breaks the iteration
            ]
        OTHERWISE move *)
move ::= /adjust (ENCLOSING middle_stage.coordinates);
    send_SUT_input( ENCLOSING middle_stage.coordinates);
    -- this simulates SUT sensor input
    DELAY(50 msec); /
\end{verbatim}
Prototype implementation

The test generator based on attributed event grammars has been implemented at NPS.

It takes an AEG and generates a test driver in Java.
How it works

- **Environment model represented as an event grammar**
- **Generator**
- **Run time monitor**
- **Test driver** (in C, Java, or assembly language)
- **SUT**

How to create test cases

How to run test case

How to monitor the results
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 ("software-in-the-loop" simulations).
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.
Qualitative Risk Analysis (2)

- We can change some parameters in the model and repeat the set of tests. If the frequency of reaching a hazardous state changes, we can find out how the parameter values influence the probability to reach a hazard state.

- We suggest to use the *combinatorial testing technique* based on orthogonal arrays, an approach well familiar to statisticians.
The same conjecture that stipulates that the fault in behavior of the SUT in most cases depends either on a single parameter value or on an interaction of a pair of parameter values could be applied to the system safety testing. This conjecture still has to be verified by experiments.

Combinatorial approach will significantly reduce the number of experiments needed to establish statistically sound conclusions about probabilities to reach hazard state for different environment model settings.

In order to apply combinatorial testing techniques the values of model parameters have to be split into a finite number of equivalence classes, a technique well known in software component testing.
SUT safety assessment with automated scenario generation

- Environment Model (with parameters)
- Scenario generator
- Insert a set of model parameters (tool under development)
- Parameter tuple combinatorial generator (IBM tool)
- Test driver
- SUT “black box”
- Test Results
- Statistical evaluation of results
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.
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] ;
Backup slides
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

\[
\text{A precedes B and B precedes C } \Rightarrow \text{ A precedes C}
\]

\[
\text{A in B and B precedes C } \Rightarrow \text{ A precedes C}
\]
Example of an Event Grammar

\[
\begin{align*}
\text{ex\_prog} &:: \text{ex\_stmt} \; * \\
\text{ex\_stmt} &:: \text{ex\_assignmnt} \mid \text{ex\_read\_stmt} \mid \ldots \\
\text{ex\_assignmnt} &:: \text{eval\_expr} \; \text{destination}
\end{align*}
\]
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
  \[ x: \text{func\_call} \land x.\text{name} == "A" \]
  \[ \text{eval\_expr} :: (\ \text{variable}) \]

- List of events
  \[ [\ \text{exec\_assignmnt} \text{ FROM ex\_prog}] \]

- List of values
  \[ [x: \text{exec\_assignmnt} \text{ FROM ex\_prog APPLY x.value}] \]
FORMAN language

Aggregate Operations

MAX/\[ x: \text{exec} \_\text{assignmt} \ FROM \ \text{ex} \_\text{prog} \ APPLY \ x.\text{value} \]

AND/\[ x: \text{exec} \_\text{assignmt} \ FROM \ \text{ex} \_\text{prog} \ APPLY \ x.\text{value} > 17 \]

Or

FOREACH \[ x: \text{exec} \_\text{assignmt} \ FROM \ \text{ex} \_\text{prog} \ x.\text{value} > 17 \]
Examples

1) Profile

\[
\text{SAY( "Number of function A calls is " C\text{ARD}[ x: \text{func\_call} \& x.\text{name} == "A" \\
FROM \text{ex\_prog} ] )}
\]

2) Generic debugging rule (typical error description)

\[
\text{FOREACH e: eval\_expr :: (v: variable)} \\
\text{FROM \text{ex\_prog}} \\
\text{EXISTS d: destination FROM e.PREV\_PATH} \\
v.\text{source\_code} = d.\text{source\_code} \\
\text{ONFAIL SAY("Uninitialized variable " v.\text{source\_code} "is used in expression " e)}
\]
3) Debugging query

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

4) Traditional debugging print statements

```sql
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
...

Ex-program
Ex-PAR
Ex-par-process
Ex-stmt1
Ex-stmt2
send
wait
Rendez-vous
Eval-out-expr
Ex-stmt3
Ex-assignment
Parallel thread
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
- 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.