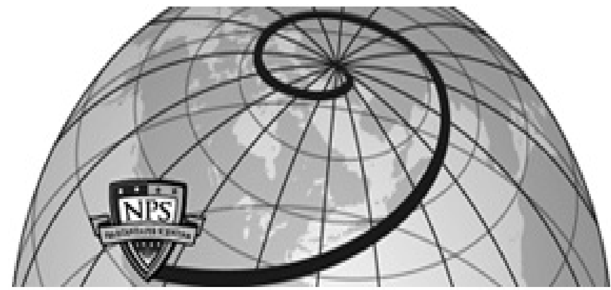
Calhoun
Institutional Archive of the Naval Postgraduate School



Calhoun: The NPS Institutional Archive

DSpace Repository

Faculty and Researchers | Faculty and Researchers Collection

2006

Environment behavior models for automation of testing and assessment of system safety

Auguston, Mikhail; Michael, James Bret; Shing, Man-Tak

Elsevier

Auguston, Mikhail, James Bret Michael, and Man-Tak Shing. "Environment behavior models for automation of testing and assessment of system safety." Information and Software Technology 48.10 (2006): 971-980.
http://hdl.handle.net/10945/59398

Downloaded from NPS Archive: Calhoun

Calhoun is a project of the Dudley Knox Library at NPS, furthering the precepts and goals of open government and government transparency. All information contained herein has been approved for release by the NPS Public Affairs Officer.

Dudley Knox Library / Naval Postgraduate School
411 Dyer Road / 1 University Circle
Monterey, California USA 93943

http://www.nps.edu/library

Available online at www.sciencedirect.com

ScienceDirect

ELSEVIER

Information and Software Technology 48 (2006) 971–980

INFORMATION AND SOFTWARE TECHNOLOGY

www.elsevier.com/locate/infsof

Environment behavior models for automation of testing and assessment of system safety

Mikhail Auguston *, James Bret Michael, Man-Tak Shing

Department of Computer Science, Naval Postgraduate School, 833 Dyer Road, Monterey, CA 93943, USA

Received 27 February 2006; accepted 21 March 2006
Available online 12 May 2006

Abstract

This paper presents an approach to automatic scenario generation from environment behavior models for testing of real-time reactive systems. The model of behavior is defined as a set of events (event trace) with two basic relations: precedence and inclusion. The attributed event grammar (AEG) specifies possible event traces and provides a uniform approach for automatically generating and executing test cases. The environment model includes a description of hazardous states in which the system may arrive and makes it possible to gather statistics for system safety assessment. The approach is supported by a generator that creates test cases from the AEG models. We demonstrate the approach with a case study of a software prototype of the computer-assisted resuscitation algorithm for a safety-critical casualty intravenous fluid infusion pump.
Published by Elsevier B.V.

Keywords: Model-based testing; Testing automation; Reactive and real-time system testing

1. Introduction

Testing is both a time- and effort-consuming process. Testing real-time reactive systems is complicated: such systems continuously interact with their environment and both their inputs and outputs should satisfy timing constraints. Interactions with the tester often introduce unacceptable overhead that render the test results meaningless. Such systems can only be tested via an automated testing environment with processing characteristics sufficiently close to the actual operating environment [16]. Modeling can be used to gain a better understanding of the environment.

Until recently, most approaches to test automation have been based on some form of formal specification of the requirements [8–10] and/or assertions describing the correct behavior of program code segments [7,15]. System-safety constraints and safety-related functional requirements can only be tested by evaluating the system within the context of its operating environment. For example, a common approach to verifying safety requirements involves developing two separate models: one for the system under test (SUT) and the other for the environment (or equipment) under its control. The two models are then exercised in tandem to check whether the simulation ends up in known hazardous states under normal operating conditions and under various failure conditions. Hence, correct modeling of the environment is as important as the correct analysis of the system requirements.

This paper builds on our previous work on environment models for testing automation and safety assessment [6].

2. Environment versus system modeling

Traditionally, modeling approaches used for software development focus on the system under development. These models emphasize the reactive aspects of the system behavior, which are typically modeled using statechart formalism. In contrast, the purpose of the environment model is to generate stimuli for the system under test.

* Corresponding author. Tel.: +1 831 656 2607.
E-mail addresses: maugusto@nps.edu (M. Auguston), bmichael@nps.edu (J.B. Michael), shing@nps.edu (M.-T. Shing).

0950-5849/$ - see front matter Published by Elsevier B.V.
doi:10.1016/j.infsof.2006.03.005

An environment model emphasizes the productive aspects of the behavior.

It has become a common practice for engineers to analyze system behaviors from an external point of view using use cases. In UML (Unified Modeling Language) [14] use case scenarios are written in natural language and focus on the events and responses between the actors and the system. Functional requirements can be derived from the description of events received by the system and the expected responses generated by the system.

The major paradigms for modeling system behavior are based on different variations of finite state machines. Active research in this area focuses on different aspects of behavior specification based on UML statecharts, message sequence diagrams, or other types of extended finite state machines, like timing automata [12] or Petri nets.

State machines are typically used for modeling systems. System models are built around the notion of a transition in response to the environment stimulus. Grammars are common vehicles for generating structured sets of inputs.

While grammars and state machines are considered to be dual, researchers have long recognized the power of state machines as acceptors and grammars as generators.

A major feature of our approach presented in this paper is the notion of an event trace as a formal model of behavior. Event grammars are one of the possible frameworks to utilize this notion. They are text-based, have a smaller semantic distance from the use case scenarios than the state machines, and are well suited to model environments described via use case scenarios. Event grammars are convenient in specifying dynamic environments with an arbitrary number of actors (and concurrent events), whereas state machines are effective for modeling static environments (with a predetermined numbers of actors).

Wang and Parnas [30] proposed to use trace assertions to formally specify the externally observable behavior of a software module and presented a trace simulator to symbolically interpret the trace assertions and simulate the externally observable behavior of the module specified. Their approach is based on algebraic specifications and term rewriting techniques and is only applicable to non-real-time applications.

Alfonso et al. [2] presented a formal visual language for expressing real-time system constraints as event scenarios (events and responses) and provided a tool to translate the scenarios into observer timed automata, which can be used to study properties of the formal model of the system under analysis via model checking and run-time verification. While there are a lot of similarities between the approach presented in [2] and ours, the former is effective for modeling static environments (with fixed scenarios) whereas ours, which is based on event grammar, is more effective in specifying dynamic environments with an arbitrary number of actors (and concurrent events).

Behavior models based on event grammars can be designed not only for the environment, but for the SUT as well, and used for run-time verification and monitoring. Such software behavior models may be used to automate system test-result verification. Details can be found in previously published papers on event grammars for program testing, monitoring, and debugging automation [3–5] and will not be discussed in this paper.

Context-free grammars have been used for test generation, in particular, to check compiler implementation, such as in [21], and [20] provides an outlook in the use of enhanced context-free grammars for generation of test data.

3. Event traces and event grammars

3.1. Event trace as a model of behavior

Our approach focuses on the notion of an *event*, which is any detectable action in the environment that could be relevant to the operation of the SUT. A keyboard button pressed by the user, a group of alarm sensors triggered by an intruder, a particular stage of a chemical reaction monitored by the system, and the detection of an enemy missile are examples of events. An event usually is a time interval, and has a beginning, an end, and duration. An event has *attributes*, such as type and timing attributes.

Two basic relations are defined for events: *precedence* (PRECEDES) and *inclusion* (IN). Two events may be ordered in time, or one event may appear inside another event. The behavior of the environment can be represented as a set of events with these two basic relations defined for them (*event trace*). The basic relations define a partial order of events. Two events are not necessarily ordered, that is, they can happen concurrently. Both relations are transitive, non-commutative, non-reflexive, and satisfy distributivity constraints. The following axioms should hold for any event trace.

Let a, b, c, and d be events of any type.

Mutual Exclusion of Relations
Axiom 1.1: a PRECEDES b $\Rightarrow \neg$(a IN b)
Axiom 1.2: a IN b $\Rightarrow \neg$(a PRECEDES b)

Non-commutativity
Axiom 2.1: a PRECEDES b $\Rightarrow \neg$(b PRECEDES a)
Axiom 2.2: a IN b $\Rightarrow \neg$(b IN a)

Irreflexivity for PRECEDES and IN follows from non-commutativity.

Transitivity
Axiom 3.1: (a PRECEDES b) $\wedge$ (b PRECEDES c) $\Rightarrow$ (a PRECEDES c)
Axiom 3.2: (a IN b) $\wedge$ (b IN c) $\Rightarrow$ (a IN c)

Distributivity

Axiom 4.1: (a IN b) ∧ (b PRECEDES c) ⇒ (a PRECEDES c)

Axiom 4.2: (a PRECEDES b) ∧ (c IN b) ⇒ (a PRECEDES c)

Axiom 4.3: (∀a IN b (∀c IN d (a PRECEDES c))) ⇒ (b PRECEDES d)

3.2. Event grammar and trace generation

Usually event traces have a specific structure (or constraints) in a given environment. The structure of possible event traces can be specified by an *event grammar*. Here identifiers stand for event types, sequence denotes precedence of events, (...|...) denotes alternative, (*...*) means a sequence of zero or more ordered events, [...] denotes an optional element, {a, b} denotes a set of two events a and b without an ordering relation between them, and {*...*} denotes a set of zero or more events without an ordering relation between them.

The rule A: B C means that an event of the type A contains (IN relation) ordered events of types B and C, correspondingly (PRECEDES relation).

The notion of behavior model based on event traces provides the basis for the automated test generation tool described in the rest of this paper. Attributed event grammars (AEG) are intended to be used as a vehicle for automated random event trace generation. Randomized decisions about what alternative to take and how many times to perform the iteration should be made during the trace generation. All generated concurrent events within sets start simultaneously. The DELAY(t) construct may be used to indicate delays in the event beginning. For example, in the following composite event beginning of each of two parallel events A and B may be delayed for up to 10 time units from the beginning of the composite event.

{DELAY(Rand(0..10)) A, DELAY(Rand(0..10)) B}.

The major difference with traditional attributed context-free grammars is in the nature of objects defined by the grammar: instead of sequences of symbols, AEG deals with event traces, sets with two basic relations, or directed acyclic graphs.

The event grammar defines a set of possible event traces—a model of behavior for a certain environment.

Each event type may have a different attribute set. An event grammar can contain attribute evaluation rules similar to the traditional attribute grammar [23]. Attribute values are evaluated during the AEG traversal. The */action/* is performed immediately after the preceding event is completed. It is assumed that the AEG is traversed top-down and left-to-right and only once to produce a particular event trace.

We start with a simple example of an environment model to introduce the basic notation for the AEG tool.

Example

The interface with the SUT can be specified by an action that sends input values to the SUT. This may be a subroutine in a common programming language like C that hides the necessary wrapping code. In the following example of specifying a variety of use case scenarios for a simple calculator, we suppose that the SUT should receive a message about the button pressed by the user corresponding to the appropriate wrapper subroutine shown in Fig. 1: enter_digit( ), enter_operation( ), and show_result( ).

Some event types in this model have attributes associated with them.

```
RULE Perform_calculation {
                     int   result;}
RULE Enter_number {
                 int   digit, value;}
RULE Enter_operator {
                   char   operation;}
```

The environment model for the calculator specifies event sets and attribute evaluation rules.

```
Use_calculator: (* Perform_calculation *);
```

The whole testing scenario is represented as a sequence of Perform_calculation events.

```
Perform_calculation:
  Enter_number Enter_operator Enter_number
    WHEN(Enter_operator.operation == '+')

      / Perform_calculation.result =
        Enter_number[l].value +
        Enter_number[2].value; /
      ELSE
      / Perform_calculation.result =
        Enter_number[l].value-
        Enter_number[2].value; /
  [P(0.7) Show_result];
```

Perform_calculation is a sequence of three events. The WHEN clause is checked after the last of these three events has been completed and provides for conditional action, Enter_number[l] refers to the first occurrence of an event in the rule Perform_calculation, and correspondingly, Enter_number[2] refers to the second occurrence. In this example all event attribute evaluations can be accomplished at the generation time. The optional event Show_result will be generated according to the probability P(0.7) assigned to it. The value of attribute Perform_calculation.result can be used as a test oracle for this particular part of the test case.

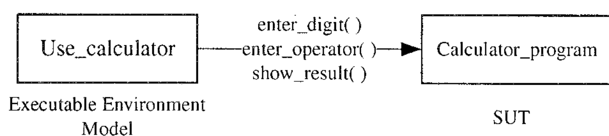


Fig. 1. The environment model for the calculator scenario.

```
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); / *) (1..6);
```

The action Enter_number.digit = RAND[0..9]; assigns a random value from the interval 0..9 to the digit attribute. Each time the rule for Enter_number event is traversed, the number of iterations will be selected at random from interval 1..6.

The traversal of AEG rules is performed top-down and from left to right, and for each iteration the attributes Enter_number.digit and Enter_number.value are recalculated. The action enter_digit(Enter_number.digit) feeds the corresponding value to the SUT.

```
Enter_operator:
   (P(0.6) / enter_operation('+');
      Enter_operator .operation= '+'; / |
   P(0.4) / enter_operation('−');
      Enter_operator .operation= '−'; / );
```

When traversing this rule, the choice of action sending the operator symbol to the SUT is based on the probability assigned to the corresponding alternative. In this example it is assumed that typical usage will more frequently involve the "+" operation neither the "−". The use of probabilities for alternatives in the AEG model demonstrates how the usage or operational profile [24] may be represented in the AEG environment model.

```
Show_result: /show_result();/ ;
```

The event Show_result, when generated, will trigger a call to the wrapper subroutine that sends a message to the SUT.

We can generate a large number of Use_calculator scenarios (event traces) satisfying this AEG and each event trace will satisfy the constraints imposed by the event grammar.

The AEG model does not satisfy the definition of a Markov process as a stochastic process that obeys the Markov property, in which the next event in a series can be determined on the present, without reference to the past [24]. The presence of attribute evaluation statements and the WHEN construct make the event generation dependent on the previous events in the trace. The considerations supporting the use of statistical testing and usage profiles [24] are valid also for the random test suites generated from the AEG models.

4. The CARA infusion pump example

The CARA system is a safety-critical software developed by the Walter Reed Army Institute of Research. The computer-assisted resuscitation algorithm (CARA) has been developed for an infusion pump [27–29], and has been used as a case study by several software engineering research groups [1,19,11]. The algorithm is designed to monitor a patient's blood pressure and automatically signal to a computer-controlled pump to administer intravenous (IV) fluids in order to restore a desired intravascular volume and blood pressure.

The main responsibilities of the CARA system include:

1. Monitoring a patient's blood pressure.
2. Controlling a high-output patient resuscitation infusion pump.
3. Displaying (to a human caregiver) vital information about the patient and the system.
4. Logging all data.
5. Generating an alarm to warn the caregiver that an emergency situation exists.

We will use CARA to demonstrate automatic scenario generation from environment models for testing real-time reactive systems. We have extracted from [29] a subset of requirements that corresponds to: (i) infusion pump functionality and (ii) blood pressure monitoring, and developed an OMNeT++ [22] simulation program for the expected behavior of the CARA software. For space and simplicity reasons we have excluded those requirements pertaining to message display and logging.

4.1. The CARA implementation in OMNeT++

We have used OMNET++ to design an example of reactive real-time SUT for the experiments with AEG test generator. This section describes the design of SUT itself, and Section 4.2 presents an environment model in AEG notation for testing this SUT.

OMNeT++, which stands for *Objective Modular Network Testbed in C++*, is an object-oriented discrete-event simulator primarily designed for the simulation of communication protocols, communication networks and traffic models, and models of multiprocessor and distributed systems [26]. An OMNeT++ simulation model consists of a set of modules communicating with each other via the sending and receiving of messages. Modules can be nested hierarchically. The atomic modules are called *simple modules*; they are coded in C++ and executed as coroutines on top of the OMNeT++ simulation kernel. *Gates* are the input and output interfaces of the modules. Messages are sent out through output gates of the sending module and arrive through input gates of the receiving module. Input and output gates are linked together via connections. *Connections* represent the communication channels and can be assigned properties such as propagation delay, bit error rate and data rate.

Fig. 2 shows the top level of the OMNeT++ simulation model for the CARA test environment. It consists of two modules, *ENV* and *CARA*. The *ENV* module contains

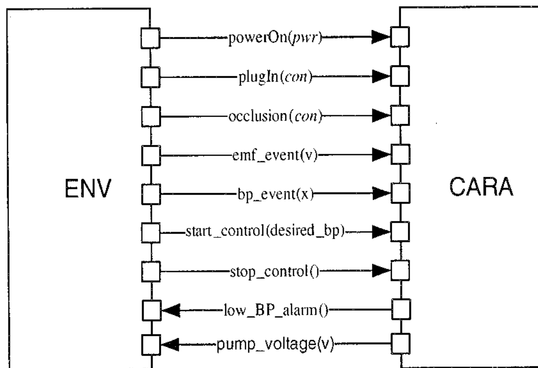


Fig. 2. The OMNeT++ model for the CARA software.

the C++ code of the test driver generated from the AEG environment model. The *CARA* module contains the C++ code that simulates the functional behavior of the CARA software and determines the status of the pump and the patient via the *power_on*, *plug_in*, *occlusion*, *bp_event*, *emf_event*, *start_control* and *stop_control* messages from the *ENV* module and controls the rate of infusion via the *CARA_ready* and *pump_voltage* messages to the *ENV* module. It also alerts the environment via the *low_BP_alarm* message if it determines that the patient's blood pressure has fallen below a pre-set critical value.

4.2. The environment model

The following basic CARA environment model has been used as a starting point to create several variations of environment models for CARA testing purposes, in particular by selecting different probabilities for occlusion events, and by simulating random increase in the bleeding rate (the BR global parameter).

The environment model specification in AEG starts with declarations of global parameters.

Global parameters:

```
GLOBAL {
double MIN_BP; /* minimal blood pressure threshold */
double BR; /* patient's bleeding rate */
double RR; /* initial pump rotation rate */
double V; /* initial pump voltage */
double VRR; /* voltage to rotation rate coefficient */
double RRF; /* rotation rate to flow coefficient */
double EMF; /* voltage coefficient */
double DESIRED_BP; /*patient's ideal blood pressure */
}
```

Attributes for each rule are defined in corresponding declaration sections.

Event attributes:

```
RULE CARA_environment {
   double pump_flow;
   double blood_pressure;
   double bleeding_rate;
   double volume;
   bool occlusion_on;
   bool patient_death_flag;
}

RULE Patient {
   bool resuscitation_success;
}

RULE Pump {
   double rotation_rate;
   double voltage;
}
```

The environment model:

```
CARA_environment:
/
        /* Initialize Globals */

        MIN_BP = getMIN_BP();
        BR = getBR();
        RR = getRR();
        V = getV();
        VRR = getVRR();
        RRF = getRRF();
        EMF = getEMF();
        DESIRED_BP = getDESIRED_BP();

 /* Initialize CARA_Environment attributes */

CARA_environment.pump_flow      = getpump_flow();
CARA_environment.bleeding_rate      = BR;
CARA_environment.volume         = get_volume();
CARA_environment.blood_pressure =
                                  get_initBP();
CARA_environment.patient_low_bp          = false;
CARA_environment.patient_death_flag      = false;
CARA_environment.occlusion_on     = false;  /

        { Patient, LSTAT, Pump };
```

Initial settings for global parameters and attributes are obtained from calling auxiliary subroutines for the convenience of changing them dynamically during a long series of test runs. The probabilities of events, such as the probability of occlusion events, should be available when tests are generated. In the next version of AEG generator this will be replaced by a description of ranges and increment values for changing global parameter values for long series of simulations.

The behavior model is represented by three concurrent threads of events: Patient, LSTAT and Pump. Since each of these events is an iteration with a 1 s periodic rate (see the corresponding rule definitions below), the synchronization is simply implied by this timing constraint: all shared attribute values are updated every 1 s. Since the generated test driver is a sequential C program, this eliminates concerns about data race.

```
Patient:
(* CATCH Low_bp_alarm()
/* Attempt resuscitation by the attending medic */
   ResuscitationAttempt
(P(0.1)  /  ENCLOSING  CARA_environment.volume =
((MIN_BP + 15) * 50);
   /* resuscitation boost */ /
   WHEN (ENCLOSING
      CARA_environment.occlusion_on == false)
      /* assuming no occlusion, restart CARA */
      Restart_CARA
| (P(0.9) Death
   / ENCLOSING
      CARA_environment.patient_death_flag = true;
      BREAK;/ )
)
END_CATCH
   *) (== 600) (EVERY 1 s);
```

This simple model of Patient behavior determines the Patient's reaction to the Low_bp message from CARA in terms of the attempted resuscitation. The CATCH construct represents the external event of receiving a message from CARA. It is implemented as a function Low_bp_alarm() call, which returns a True value when CARA has issued corresponding output. This function encapsulates the interface with the OMNeT++ message queue. If there is no input from the SUT at the moment of CATCH check, the event stream proceeds to the next action. This rule demonstrates the ability of AEG to specify so-called **adaptive test cases** [13] in which the input applied at a step depends upon the output sequence that has been observed.

The construct ENCLOSING CARA_environment provides for the event Patient access to the attributes of the parent event CARA_environment which are not within the scope of this rule, but are available during the event Patient. This reference mechanism is convenient for event-attribute propagation over the generated event trace. The (EVERY 1 s) clause guides the event-trace generation with the desired event time stamps. The (==600) construct determines number of iterations generated, that is, the duration of the test case will be approximately 10 minutes.

```
LSTAT:
   Power_on /send_power_on(true);/
   (* WHEN (ENCLOSING
      CARA_environment.patient_death_flag == true)
         (/ENCLOSING CARA_environment.
            blood_pressure = 0;
         send_bp_event(ENCLOSING
         CARA_environment.blood_pressure);/)
      ELSE ( / ENCLOSING
               CARA_environment. volume +=
         ENCLOSING CARA_environment .pump_flow -
         ENCLOSING CARA_environment.bleeding_rate;
         ENCLOSING
            CARA_environment.blood_pressure =
               (ENCLOSING
            CARA_environment.volume / 50 - 10);
         send_bp_event(ENCLOSING
            CARA_environment.blood_pressure);
      /)
   *) (== 600) (EVERY 1 s);
```

LSTAT is a model of the part of environment (the stretcher) responsible for monitoring the patient's blood pressure measurements and sending the initial messages to CARA. As in the case of CATCH, the send_power_on() subroutine encapsulates the details of the OMNeT++ message queue interface.

```
Pump:
   Plugged_in
   / send_plugIn(true);
     send_start_control(DESIRED_BP);
Pump.rotation_rate = RR;
     Pump.voltage = V; /
   {Pumping, Voltage_monitoring};
```

The Pump event contains two parallel events Pumping, and Voltage_monitoring.

```
Pumping:
(* / ENCLOSING Pump. rotation_rate =
      ENCLOSING Pump. voltage * VRR;
ENCLOSING CARA_environment .pump_flow =
      ENCLOSING Pump. rotation_rate * RRF;/
CATCH pump_voltage(V)
      /ENCLOSING Pump.voltage = V;/
END_CATCH
WHEN (ENCLOSING
      CARA_environment.occlusion_on == false)
[ P(0.05) Occlusion
/ENCLOSING CARA_environment.occlusion_on =
      true;
send_occlusion(true);/ ]
ELSE (/send_occlusion(false);/)
WHEN(ENCLOSING CARA_environment.occlusion_on)
   [P(0.3) / ENCLOSING
    CARA_environment.occlusion_on = false;
    send_occlusion(false); /
    Restart_CARA ]
*) (== 600) (EVERY 1 s);

Voltage_monitoring:
   (* / send_EMF_event( ENCLOSING Pump.voltage);/
   *) (== 120) ( EVERY 5 s);
```

The Voltage_monitoring event thread is responsible for sending the pump EMF voltage measurements to the SUT every 5 seconds. The Pumping event thread is responsible for updating the rotation rate and the IV flow rate based on the voltage set by CARA. The CATCH construct in the Pumping event thread represents an external event of

receiving an input from the SUT. The synchronization is achieved by the identical periodicity of both iterations. This rule also simulates random occurrence of the Occlusion event.

```
Restart_CARA:
    / send_stop_control();
    send_plugged_in(true);
    send_start_control(DESIRED_BP); / ;
```

5. Automated test generation

The event trace generated from the AEG model traversal contains both events and actions that should be performed at corresponding time moments. The AEG environment model specifies a set of possible scenarios (or use cases) for the SUT. The events can be sorted according to the timing attributes and converted into a test driver, which feeds the SUT with inputs and captures SUT outputs. The actions like wrapper subroutine calls can be extracted from the event trace and assembled into test-driver code. The functionality of this generated test driver is limited to feeding the SUT inputs and receiving outputs and may be implemented as a C or even assembly-language program. Actually, only *send* and *catch* actions obtained from the event trace are needed to construct the test driver, the rest of events and actions in the event trace are used as "scaffolds" to obtain the ordering, timing and other attributes of these actions.

Separation of the generation phase from test execution is essential for the performance of the generated test driver: event selection and attribute evaluation can be performed at generation time, with test drivers containing only wrapper calls to interact with the SUT.

The first prototype of an automated test generator based on attributed event grammars has been implemented. It takes an AEG, derives a random event trace from it according to probabilities and iteration guards in the AEG model, and generates a test driver in C. Fig. 3 represents the relationship between the components of testing process.

Some design highlights:

- Parallel event threads (for sets, like {A, B}) are implemented by interleaving events/actions within them.

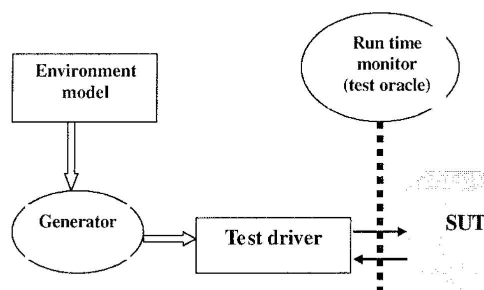


Fig. 3. The use of the automated test generator.

- All loops in AEG model are unfolded either using explicit iteration guards, or by assuming a random number of iterations. Recursion is not supported in the current version.
- Attributes are evaluated mostly at the generation time, but those dependent on SUT outputs (on *catch* clauses) are postponed till the run time. Certain parts of generated event trace may depend on those attribute values (for instance, because the delayed attribute participates in the *when* clause), in this case both alternatives for the expected trace segment are generated but protected by Boolean flags, so that at the test run time only the alternative for which the guard is enabled will be executed.

6. Case studies

We conducted three case studies to investigate the effectiveness of the AEG-based test automation in support of correctness testing, safety assessment, and evaluation of design alternatives for the CARA software.

6.1. Case study 1

The purpose of the first case study is to evaluate the ability of the environment model to detect errors in the CARA simulation. Four hundred test drivers were generated automatically from the AEG model using the following parameters:

- Patient's starting diastolic blood pressure is between 55 and 65 mmHg.
- Patient's desired diastolic blood pressure = 70 mmHg.
- Patient's normal bleeding rate = 5 ml/s.
- Probability of occlusion in the IV line = 0.05.

The test runs discovered several errors and unexpected behaviors in the CARA code. For example, in one of the early versions CARA continued sending voltage commands to the pump even after the patient dies. A closer examination of the code showed that the pending messages in the queue were not cleared after the Patient's Death event.

6.2. Case study 2

The purpose of the second case study is to demonstrate that the environment model can be used to perform quantitative assessment of software system safety. In particular, we wanted to investigate environmental factors, other than

Table 1
Results of Case Study 2

Simulation trial (1000 runs)	Number of cases	Average No. of occlusions	Average starting systolic BP
Death scenarios	134	1.88	109.2
Non-death scenarios	866	1.52	115.3

an excessive bleeding rate, that may increase the probability of the patient's death. The results of generating and executing 1000 test runs, with a varying occlusion rate and starting systolic blood pressures, are shown in Table 1.

The results indicate that CARA's performance may be adversely affected by a large number of occlusions in the IV line or low starting values (i.e., initial samples taken) of blood pressure.

6.3. Case study 3

The purpose of the third case study is to demonstrate the environment model's ability to support the study of design alternatives. In this experiment, 1000 test runs were generated to assess the performance of two pump-control algorithms with the following parameters:

- Patient's starting diastolic blood pressure is between 55 and 65 mmHg.
- Patient's desired diastolic blood pressure = 70 mmHg.

To test the ability of the algorithms to cope with changing bleeding rate, we allow the patient's bleeding rate to vary as follows:

5 ml/s with a probability of 0.8.
65 ml/s with a probability of 0.2.

The first algorithm generates the pump voltage using a simple static logic shown in Fig. 4, while the second algorithm computes the pump voltage using the estimated blood loss based on the most recent two patient blood pressure readings.

The experiment has shown that the second algorithm helps to reach the desired blood pressure much faster on the average but also causes more deaths.

A closer examination of the data collected in the experiment showed that the second algorithm pushes less IV fluid to the patients than the first algorithm when the patient's diastolic blood pressure falls below 57.5 mmHg in the test runs that end up in patient death. Hence, we combined the two algorithms into a hybrid algorithm that uses the first algorithm to compute the pump voltage whenever the patient's diastolic blood pressure is below 57.5 mmHg and uses the second algorithm in all other cases. We re-ran the 1000 test cases on the hybrid algorithm. The hybrid algorithm outperformed the other two pump-control algorithms in terms of the algorithm's effectiveness at stabilizing the patient's blood pressure without endangering the patient's life.

```
if (bp > 70) // coasting region
  control_voltage = KVO_Voltage;
else if (57.5 <= bp <= 70) // normal operation region
  control_voltage = (desired_bp - bp) /12.5 *  Max_Voltage();
else  // critical region
  control_voltage = Max_Voltage;
```

Fig. 4. The Static Pump Control Algorithm.

7. Safety assessment

NASA-STD-8719.13A [25] defines *risk* as a function of the possible frequency of occurrence of an undesired event, the potential severity of resulting consequences, and the uncertainties associated with the frequency and severity.

The environment model can contain a description of hazardous states in which the system could arrive, as documented in a system-safety constraints and safety-related functional requirements derived from the preliminary hazard analysis (PHA), and which could not be derived from the model of the SUT itself. For example, the Patient's Death 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 a large enough number of (automatically generated) tests, the statistics gathered gives some approximation of the risk of entering the hazardous state. This becomes a constructive process of performing experiments with SUT behavior within the given environment model.

AEG-based environmental modeling provides yet another approach to specifying a formal model of usage states, failure states, and transition probabilities as suggested in [24], and for design of automatic scripts for testing. Our formalism supports dynamic assessment of safety-critical software systems, in particular, statistical testing for both nominal and off-nominal cases. It is well known, however, that it is not always possible to determine whether a particular test-result is indicative of safe operation, and hence, one can place only limited confidence in the results of dynamic testing of software systems [18].

By performing sensitivity analyses on parameters such as BR, probability of the occlusion, and probability of successful resuscitation, we can learn what impact those parameters have on the probability of hazardous outcome, and identify thresholds for SUT behavior in terms of those values. Estimating the test-adequacy of the generated test-set with respect to assessing safety could be accomplished using a method similar to that proposed in [17].

The results of the testing, based on AEG models, can be integrated into safety reviews and analyses, so that the identified hazards and design features to control them can be further investigated and possibly improved.

8. Conclusions and open issues for the future work

This paper describes an AEG-based approach to automatic generation of scenarios from environment models for use in testing real-time reactive systems. From the limited experience with the case studies main advantages and open questions of the approach may be summarized as follows.

- Environment models specified by attributed event grammars provide for automated generation of a large number of pseudo-random test drivers. Investigation of stochastic models based on AEG and corresponding mathematical structures for reasoning

about sets of event traces generated from AEG models is beyond the scope of this paper and is a subject for future work.

- Generated test drivers could be used for real-time test drivers as demonstrated by Case study 1.
- The version of AEG notation described in this paper provides rudimentary concurrency constructs (concurrent event sets, CATCH clause for synchronizing with the SUT outputs). The event trace model supports the notion of synchronization event, which may be shared by several events (i.e. may be under the IN relation with several concurrent events.) We have not included this construct in the current version of AEG-based test generator, since it requires more sophisticated generation methods and may deteriorate the run time efficiency of generated test drivers. How far to proceed with enhancing the AEG notation with concurrency constructs (e.g. adding constructs for message sending/receiving between concurrent events within the model), possibly extending it to a full-fledged concurrent programming notation, is a topic for future work.
- Generated test drivers can be saved and reused for regression testing. We expect that environment models will be changed relatively seldom unless significant errors in the requirements are discovered during testing. The environment model itself is an asset and could be reused.
- AEG is well structured and hierarchical, as is any formalism based on formal grammars. The rule construct provides for the basic modularity in the building of the model. We have introduced constructs improving the attribute flow between rules (ENCLOSING attribute reference), which alleviate some problems with attribute copying in traditional attribute grammars. These may be factors in favor of scalability of AEG notation for the large environment model design.
- The environment model may contain events which represent hazardous states of the environment (e.g., patient's death condition in CARA environment model). Case studies 2 and 3 provide an example of both quantitative and qualitative assessment of software system safety. Such an approach may be instrumental for identifying, confirming, and mitigating hazards, arising from software faults.
- Different environment models for different purposes can be designed, such as for testing extreme scenarios by increasing probabilities of certain events, or for load testing. More detailed research is needed to develop statistical methods to analyze the results of experiments with SUT based on large number of test cases automatically generated from AEG models.

Acknowledgement

The research reported in this article was funded in part by a grant from the U.S. Missile Defense Agency. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the U.S. Government.

References

[1] R. Alur, D. Arney, E. Gunter, I. Lee, W. Nam, J. Zhou, Formal specifications and analysis of the computer assisted resuscitation algorithm (CARA) Infusion Pump Control System, J. Software Tools Technol. Transfer 5 (4) (2004) 308–319.

[2] A. Alfonso, V. Braberman, N. Kicillof, A. Olivero, A. Visual timed event scenarios, in: Proc. 26th Int. Conf. on Software Engineering, ACM Press, Edinburgh, Scot., May 2004, pp. 168–177.

[3] M. Auguston, A language for debugging automation, in: Chang, S.K. (Ed.), Proc. 6th Int. Conf. on Software Engineering and Knowledge Engineering, Skokie, Ill., Knowledge Systems Inc., June 1994, pp. 108–115.

[4] M. Auguston, Lightweight semantics models for program testing and debugging automation, in: Proc. 7th Monterey Workshop, Santa Margherita Ligure, Italy, June 2000, pp. 23–31.

[5] M. Auguston, C. Jeffery, S. Underwood, A framework for automatic debugging, in: Proc. 17th Int. Conf. on Automated Software Engineering, ACM Press, Edinburgh, Sept. 2002, pp. 217–222.

[6] M. Auguston, J.B. Michael, M. Shing, Environment behavior models for scenario generation and testing automation, in: Proc. ICSE 2005 Workshop on Advances in Model-Based Software Testing, ACM, St. Louis, MO, May 2005.

[7] C. Boyapati, S. Khurshid, D. Marinov, Korat: automated testing based on Java predicates, in: Proc. Int. Symposium on Software Testing and Analysis, ACM Press, Rome, Italy, July 2002, pp. 123–133.

[8] M.R. Blackburn, Using models for test generation and analysis, in: Proc. 17th Digital Avionics Systems Conf, vol. 1, IEEE, Bellevue, Wash., Oct., 1998, pp. C45/1–C45/8.

[9] J.L. Crowley, J.F. Leathrum, K.A. Liburdy, Issues in the full scale use of formal methods for automated testing, in: ACM SIGSOFT Software Engineering Notes, vol. 21 (3), 1996, pp. 71–77.

[10] S.R. Dalal, A. Jain, N. Karunanithi, J.M. Leaton, C.M. Lott, C.G. Patton, B.M. Horowitz, Model-based testing in practice, in: Proc. 21st Int. Conf. on Software Engineering, Los Angeles, Calif, May 1999, pp. 285–294.

[11] D. Drusinsky, M. Shing, TLCharts: Armor-plating Harel Statecharts with temporal logic conditions, in: Proc. 15th IEEE Int. Workshop on Rapid Systems Prototyping, IEEE, Geneva, Switz., June 2004, pp. 29–36.

[12] H.S. Hong, I. Lee, Automatic test generation from specifications for control-flow and data-flow coverage criteria, in: Proc. Monterey Workshop, Monterey, Calif.: Naval Postgraduate School, Monterey, CA, June 2001, pp. 230–246.

[13] R.M. Hierons, H. Ural, Concerning the ordering of adaptive test sequences, in: Proc. 23rd IFIP Int. Conf. on Formal Techniques for Networked and Distributed Systems, Berlin, Germany, Sept. 2003, Berlin: Springer, Lecture Notes in Computer Science, vol. 2767, pp. 289–302.

[14] I. Jacobson, G. Booch, J. Rumbaugh, The Unified Software Development Process, Addison-Wesley, Reading, MA, 1999.

[15] B. Korel, A.M. Al-Yami, Assertion-oriented automated test data generation, in: Proc. 18th, in: Int. Conf. on Software Engineering, IEEE, Berlin, Germany, Mar. 1996, pp. 71–80.

[16] C. Kreiner, C. Steger, R. Weiss, Improvement of control software for automatic logistic systems using executable environment models, in: Proc. 24th Euromicro Conf., vol. 2, IEEE, Vasteras Sweden, Aug. 1998, pp. 919–923.

[17] S. Kuball, J. May, Test-adequacy and statistical testing: combining different properties of a test-set, in: Proc. 15th Int. Symposium on Software Reliability Engineering, IEEE, Saint-Malo, France, Nov. 2004, pp. 161–172.

[18] N.G. Leveson, SAFEWARE: System Safety and Computers, Addison-Wesley, Reading, MA, 1995.

[19] Luqi, M. Shing, J. Puett, V. Berzins, Z. Guan, Y. Qiao, L. Zhang, N. Chaki, X. Liang, W. Ray, M. Brown, D. Floodeen, Comparative rapid prototyping: a case study, in: Proc. 14th Int. Workshop in Rapid Systems Prototyping, IEEE, San Diego, CA, June 2003, pp. 210–217.

[20] P. Maurer, Generating test data with enhanced context-free grammars, IEEE Software (1990) 50–55.

[21] W.M. McKeeman, Differential testing for software, Digital Tech. J. 1 (1) (1998) 100–107.

[22] OMNeT++ Discrete Event Simulation System <http://www.omnetpp.org/>.

[23] J. Paakki, Attribute grammar paradigms – A high-level methodology in language implementation, ACM Comput. Surveys 2 (2) (1995) 196–255.

[24] S.J. Prowell, C.J. Trammell, R.C. Linger, J.H. Poore, Cleanroom Software Engineering: Technology and Process, Addison-Wesley, Reading, MA, 1998.

[25] Software Safety, NASA Technical Standard. NASA-STD- 8719.13A, Sept. 1997, <http://satc.gsfc.nasa.gov/assure/nss871913.html/>.

[26] A. Varga, OMNeT++ Discrete Simulation System (Version 2.3) User Manual, Technical University of Budapest, Dept. of Telecommunications (BME-HIT), Hungary, Mar. 2002.

[27] WRAIR Dept. of Resuscitative Medicine, Narrative Description of the CARA software, proprietary document, WRAIR, Silver Spring, Md., Jan 2001.

[28] WRAIR Dept. of Resuscitative Medicine, CARA Pump Control Software Questions (Version 6.1), Proprietary Document, WRAIR, Silver Spring, MD, Jan 2001.

[29] WRAIR Dept. of Resuscitative Medicine, *CARA Tagged Requirements, Increment 3 (Version 1.2)*, Proprietary Document, WRAIR, Silver Spring, Md., March 2001.

[30] Y. Wang, D. Parnas, Simulating the behavior of software modules by trace rewriting, IEEE Trans. Software Eng. 20 (10) (1994) 750–759.