Environment Behavior Models for Scenario Generation and Testing Automation

Mikhail Auguston James Bret Michael Man-Tak Shing

Department of Computer Science
Naval Postgraduate School
833 Dyer Road, Monterey,
CA 93943-5118, USA

maugusto@nps.edu bmichael@nps.edu sshing@nps.edu

ABSTRACT

This paper suggests an approach to automatic scenario generation from environment models for testing of real-time reactive systems. The behavior of the system 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, executing, and analyzing 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 case studies of prototypes for the safety-critical computer-assisted resuscitation algorithm (CARA) software for a casualty intravenous fluid infusion pump and the Paderborn Shuttle System.

Categories and Subject Descriptors

D.2.5 Testing and Debugging: Testing tools

General Terms

Design, Reliability

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 [KS]. 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 [Bl, CL, DJ] and/or assertions describing the correct behavior of program code segments [BK, KA]. Software safety 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 [AL]. Hence, correct modeling of the environment is as important as the correct analysis of the system requirements.

It has become a common practice for engineers to analyze system behaviors from an external point of view using use cases. UML 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 events received by the system and the proper responses generated by the system.

The major paradigms for modeling system behavior are based on different variations of finite state machines such as statecharts and message sequence charts in UML. 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 [HL] or Petri nets. In [WP], Wang and Parnas 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. In [ABK], Alfonso et al. 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

A-MOST'05, May15–16, 2005, St. Louis, Missouri, USA.

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 [ABK] and ours, the [ABK] approach is effective for modeling static environments (with fixed scenarios) where as ours, which is based on event grammar, is more effective in specifying dynamic environments with an arbitrary number of actors (and concurrent events).

A major feature of our approach is the notion of 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 predetermined numbers of actors).

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. This technique may be used to automate test-result verification. Details can be found in previously published papers on event grammars for program testing, monitoring, and debugging automation [A1], [A2], and [AJ] 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 [MK]. [Ma] provides an outlook in the use of enhanced context-free grammars for generation of test data.

2. Event Traces and Event Grammars

Our approach focuses on the notion of **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. 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 repetition zero or more times of 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). Both relations imply partial order and are transitive, noncommutative, nonreflexive, and satisfy distributivity constraints, like

A IN B and B PRECEDES C implies A PRECEDES C

Attributed 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. 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. The purpose is to use it as a production grammar for random event trace generation by traversing grammar rules and making random selections of alternatives and numbers of repetitions. All generated concurrent events within sets start simultaneously. The **DELAY(t)** construct may be used to indicate delays in the event beginning. For example,

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

Each event type may have a different attribute set. An event grammar can contain attribute evaluation rules similar to the traditional attribute grammar [Pa]. Attribute values are evaluated during the AEG traversal. The */action/* is performed immediately after the preceding event is completed.

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: **enter_digit()**, **enter_operation()**, and **show_result(),** (Fig. 1).

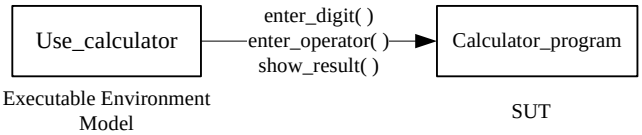

Fig. 1. The environment model for the calculator scenario

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

Perform_calculation	result
Enter_number	digit, value
Enter_operator	operation

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 ];
```

The **WHEN** clause provides for conditional action, **Enter_number[1]** refers to the first occurrence of event in the rule **Perform_calculation**, and correspondingly, **Enter_number[2]** refers to the second occurrence. In this example all event attribute evaluation can be accomplished at the generation time. The optional clause **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.

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

When traversing this rule, the choice of action sending the operator symbol to SUT is made based on the probability P(prob) assigned to the corresponding alternative.

```
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 event trace generated from the AEG traversal contains both events and actions that should be performed at corresponding time moments. The actions (wrapper subroutine calls in this example) can be extracted from the event trace and assembled into test-driver code which will perform those actions according to the timing attributes calculated during the trace generation. Thus, the event trace is used as a "scaffold" for test driver generation. 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, that is, the "scaffolding" is removed.

3. Case Study I: CARA Infusion Pump

CARA is a safety-critical software-intensive system system developed by the Walter Reed Army Institute of Research to improve life support for trauma cases and military casualties [W1]; it has been used as a case study by several software engineering research groups [AA]. The main responsibilities of the CARA system represented in this model include:

- To monitor a patient's blood pressure.
- To control a high-output patient resuscitation infusion pump.

3.1 The Environment Model

Global parameters:

MINBP	minimal blood pressure
BR	patient's bleeding rate
RR	initial pump rotation rate
V	initial pump voltage
VRR	pump voltage to rotation rate coefficient
RRF	pump rotation rate to flow coefficient
REMF	pump rotation rate to EMF voltage coefficient
p1	probability of occlusion appearance
p2	probability of occlusion disappearance

Event attributes (all values are of integer type, constants True and False stand for 1 and 0, correspondingly):

Patient	blood_pressure, volume, bleeding_rate
Pump	rotation_rate, voltage, EMF_voltage, flow, occlusion_on

The environment model:

```
CARA_environment: { Patient, LSTAT, Pump };
```

The model is represented by three concurrent threads of events: **Patient**, **LSTAT** and **Pump** (Fig. 2). Since each of these events is an iteration with 1 sec 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 sec. Since the generated test driver is a sequential C program, this eliminates the data race concerns.

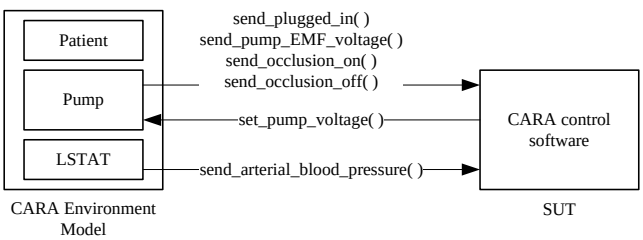

Fig. 2. The CARA environment model

```
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];
```

This simple model of Patient behavior sets dependencies on the Pump behavior, while allowing random fluctuation of the patient's bleeding rate between –9 and 9 ml/sec. The construct **ENCLOSING CARA_environment -> Pump** provides for the event **Patient** to refer to event **Pump**, which is not within scope of this rule, but is available via the parent event **CARA_environment**. This reference mechanism is convenient for event-attribute propagation over the derivation tree. The **[EVERY 1 sec]** clause guides the event trace generation with the desired event time stamps. For testing and safety assessment purposes, the event **Critical_condition** within the **Patient** event is of special interest.

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

LSTAT is a simple model of the part of environment (the stretcher) responsible for monitoring the patient's blood pressure measurements.

```
Pump:  Plugged_in
    / send_plugged_in(); Pump.rotation_rate = RR;
          Pump.voltage = V; /
    { Voltage_monitoring, Pumping };

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];
```

The **Pump** event in turn contains two independent concurrent threads. The **Voltage_monitoring** event thread is responsible for sending the pump back 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 the SUT. The **CATCH** construct in the **Pumping** event thread represents an external event of receiving an input from the SUT. It is implemented as a function **set_pump_voltage(ENCLOSING Pump.voltage)** call, which returns a True value and adjusts its parameter when SUT has issued corresponding output. 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** [HU] in which the input applied at a step depends upon the output sequence that has been observed. As mentioned above, the attribute **Pump.flow** value is shared with the **Patient** event thread. The synchronization is achieved by the identical periodicity of both iterations. This rule also simulates random occurrence of the **Occlusion** event.

NASA-STD-8719.13A [So] 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, and which could not be derived from the SUT model itself. For example, the **Critical_condition** 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, and a precise measurement of the time taken (on the average or worst-case) for the caregiver to intervene and correct the situation. This becomes a very constructive process of performing experiments with SUT behavior within the given environment model.

By experimenting with increasing or decreasing parameters such as BR, p1, and p2, we can conclude what impact those parameters have on the probability of hazardous outcome, and identify thresholds for SUT behavior in terms of those values.

4. Case Study II: Paderborn Shuttle System

The following model specifies part of the Paderborn Shuttle System behavior as presented in [PA] (Fig. 3).

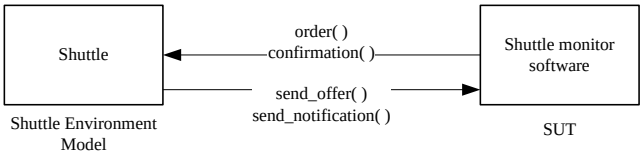

Fig. 3. The Shuttle environment model

Global parameters:

ShuttleNum	number of shuttles in the model
StationNum	number of stations
InitAccount	initial account
MaxLimit	the threshold requiring shuttle maintenance
TransitFee	the fee for a transit between stations
Wear	the wear increment after passing from station to station
Payment	transaction fee collected by the shuttle
MaintenanceFee	fee for performing shuttle maintenance

Event attributes:

Shuttle	id, at_station, account, start, destination, accepted, limit, retired

The environment model:

```
Shuttle_system:    {* Shuttle *} (1..ShuttleNum);
```

The behavior of the Shuttle System is represented as a set of parallel Shuttle threads. The SUT in this example is software that monitors the Shuttle System, sends orders and assigns tasks to shuttles.

```
Shuttle:
    / Shuttle.id = Unique_num();
      Shuttle.at_station = Rand(1..StationNum);
      Shuttle.account = InitAccount;
      Shuttle.limit = 0;
      Shuttle.retired = false; /
   (*  WAIT order(  Shuttle.start,
                    Shuttle.destination)
       WHEN (Shuttle.start == Shuttle.at_station)
           (   / send_offer(Shuttle.id,
                    calculate(Shuttle.start,
                    Shuttle.destination); /
             WAIT confirmation(Shuttle.accepted)
             WHEN (Shuttle.accepted) Move
           )
    *);
```

The Shuttle life cycle starts with setting some attribute values. After this it waits for an order from the SUT. If the shuttle is located on the start station for this order, the shuttle sends its bid to the SUT and waits for confirmation. If the bid is accepted, the shuttle proceeds with the movement. The **WAIT** construct represents an external event generated by the SUT. It halts the corresponding event thread until the expected input from the SUT is received, as opposed to the CATCH construct in the previous example, which just checks for the presence of input and if not yet received, proceeds with the next event or OTHERWISE clause.

```
Move:
    WHEN (ENCLOSING Shuttle.limit > MaxLimit)
        Maintenance
    / ENCLOSING Shuttle. at_station =
          next_station(ENCLOSING
                       Shuttle.at_station,
                ENCLOSINGShuttle.destination);
      ENCLOSING Shuttle.account -= TransitFee;
      ENCLOSING Shuttle.limit += Wear;
      send_notification(ENCLOSING Shuttle.id,
              ENCLOSING Shuttle.at_station); /
    CheckAccount
    WHEN (ENCLOSING Shuttle. at_station ==
            ENCLOSING Shuttle.Destination)
        /ENCLOSING Shuttle. account += Payment;/
      ELSE Move;
```

The **Move** event represents the activities of the shuttle during transportation. It checks to see if maintenance is needed, and updates its **at_station** attribute using the internal function **next_station**, as well as its **account** and **limit** attributes using the constants **TransitFee** and **Wear**. It sends a notification to the SUT each time it "arrives" at a station, and checks to see if its **account** is depleted. It then checks to see if it has reached its destination, and increases its **account** by the constant **Payment** if it reaches its destination. Otherwise, it will call **Move** recursively to continue its journey.

```
Maintenance:
    /ENCLOSING Shuttle.account -=
              MaintenanceFee;
     ENCLOSING Shuttle.limit = 0; /
    CheckAccount;

CheckAccount:
    WHEN (ENCLOSING Shuttle. Account <= 0)
        ( / ENCLOSING Shuttle.retired = True; /
          BREAK );
```

The **Maintenance** event decrements the shuttle's **account** by the constant **MaintenanceFee** and resets its **limit** attribute to zero. The **CheckAccount** event retires the shuttle by breaking the enclosing iteration in the Shuttle rule if the **account** is depleted.

5. Automatic Test Generation

For the purpose of scenario (and corresponding test case) generation, the AEG approach has several useful features, in particular:

- It is based on a precise and expressive behavior model in terms of an event trace with precedence and inclusion relations, well suited to capture hierarchical and concurrent behaviors. Since an event may be shared by other events, the model can represent synchronization events as well.
- The control structure suggested by the event grammar notation (sequence, alternative, iteration, concurrent event set), and the top-down, left-to-right order of traversal seems

to be intuitive and close to the traditional imperative programming style.

- Data flow of attributes is integrated with the control flow (i.e., event trace), and AEG notation provides for ease of navigation within the derivation tree (e.g., the ENCLOSING event construct, like in **ENCLOSING CARA_environment -> Patient.blood _pressure)**.

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

Some highlights:

- Parallel event threads (for sets, like {A, B}) are implemented by interleaving events/actions within them.
- All loops in AEG are unfolded either using explicit iteration guards, or by assuming a random number of iterations. Recursion, if used, can be dealt with in a similar fashion.
- Attributes are evaluated mostly at generation time, but those dependent on SUT outputs (on CATCH or WAIT clauses) are postponed until run time. Certain parts of the generated event trace may depend on those attribute values (e.g., 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.
- The generated driver contains only simple assignment statements and C subroutine calls for interfacing with the SUT, guarded by simple flags, making it efficient enough and usable for real-time SUT testing.

6. Conclusions

This paper suggests an approach to automatic scenario generation from environment models for testing of real-time reactive systems based on attributed event grammar. The main advantages of the suggested 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.
- The generated test driver is efficient and could be used for real-time test drivers.
- It addresses the regression testing problem–generated test drivers can be saved and reused. We expect that environment models will be changed relatively seldom unless significant errors in the requirements are discovered during testing.
- AEG is well structured, hierarchical, and scalable.
- The environment model may contain events which represent hazardous states of the environment (e.g., patient's Critical_condition in CARA). Experiments with the SUT embedded in the environment model provide a constructive method for quantitative and qualitative assessment of software safety. Such an approach is needed for identifying, confirming, and mitigating hazards, such as those arising from software faults in the Abbott Lifecare PCA Plus II Infusion pump that resulted in loss of life [EC].
- 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. The environment model itself is an asset and could be reused.
- Environment models can be designed early on, before the system design is complete, and can be run as environment simulation scenarios. Event traces generated from the AEG model represent examples of SUT interaction with the environment, and are in fact use cases, that could be useful for requirements specification and other prototyping tasks.

7. Acknowledgements

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

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

[ABK] Alfonso, A., Braberman, V., Kicillof, N., and Olivero, A. Visual timed event scenarios, in *Proc. 26th Int. Conf. on Software Engineering*, ACM Press (Edinburgh, Scot., May 2004), pp. 168-177.

[A1] Auguston, M. A language for debugging automation, in Chang, S. K., ed., *Proc. Sixth Int. Conf. on Software Engineering & Knowledge Engineering*, Skokie, Ill., Knowledge Systems Inc., June 1994, pp. 108-115.

[A2] Auguston, M. Lightweight semantics models for program testing and debugging automation, in *Proc. 7th Monterey Workshop: Modeling Software System Structures in a Fastly Moving Scenario*, (Santa Margherita Ligure, Italy, June 2000), pp. 23-31.

[AJ] Auguston, M., Jeffery, C., and Underwood, S. A framework for automatic debugging, in *Proc. 17th Int. Conf. on Automated Software Engineering*, ACM Press (Edinburgh, Scot., Sept. 2002), pp. 217-222.

[AL] Atchison, B. M. and Lindsay, P. A safety validation of - embedded control software using Z animation, in *Proc. 5th Int. Symposium on High Assurance Systems Engineering*, IEEE (Albuquerque, N.M., Nov. 2000), pp. 228-237

[BK] Boyapati, C., Khurshid, S., and Marinov, D., 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.

[Bl] Blackburn, M. R. 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.

[CL] Crowley, J. L., Leathrum, J. F., and Liburdy, K. A. Issues in the full scale use of formal methods for automated testing, in *Proc. ACM SIGSOFT Int. Symposium on Software Testing and Analysis*, *ACM SIGSOFT Software Engineering Notes* 21, 3 (1996), pp. 71-77.

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

[EC] ECRI. Hazard Report. Abbott PCA Plus II - Patient controlled analgesic pumps prone to misprogramming, resulting in narcotic overinfusions, *J. Health Devices* 26 (1997), pp. 389-391.

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

[HU] Hierons, R. M. and Ural, H. 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.

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

[KS] Kreiner, C., Steger, C., and Weiss, R. 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.

[Ma] Maurer, P. Generating test data with enhanced context-free grammars, *IEEE Software*, July 1990, pp.50-55

[MK] McKeeman, W. M. Differential testing for software, *Digital Tech. J.* 10, 1 (1998), pp. 100-107.

[Pa] Paakki, J. Attribute grammar paradigms - A high-level methodology in language implementation, *ACM Computing Surveys* 27, 2 (June 1995), pp. 196-255.

[PA] Paderborn Shuttle System Case Study at

http://wwwcs.upb.de/cs/ag-schaefer/CaseStudies/ShuttleSystem/

[So] Software Safety, NASA Technical Standard. NASA-STD-8719.13A, Sept. 1997, http://satc.gsfc.nasa.gov/assure/nss8719_13.html.

[W1] WRAIR Dept. of Resuscitative Medicine, Narrative Description of the CARA software, proprietary document, WRAIR, Silver Spring, Md., Jan 2001.

[WP] Wang, Y. and Parnas, D. Simulating the behavior of software modules by trace rewriting**,** *IEEE Transactions on Software Engineering* 20, 10 (Oct. 1994), pp. 750-759.