2005

Test Automation and Safety Assessment in Rapid Systems Prototyping

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

M. Auguston, J. B. Michael and M. -. Shing, "Test automation and safety assessment in rapid systems prototyping," 16th IEEE International Workshop on Rapid System Prototyping (RSP'05), Montreal, Que., 2005, pp. 188-194.
http://hdl.handle.net/10945/59428

Test Automation and Safety Assessment in Rapid Systems Prototyping

Mikhail Auguston, James Bret Michael, Man-Tak Shing
Department of Computer Science
Naval Postgraduate School
833 Dyer Road, Monterey, CA 93943-5118, USA
{maugusto, bmichael, shing}@nps.edu

Abstract

This paper addresses the need for automatic generation of executable environment models to facilitate the testing of real-time reactive systems under development (SUD) in rapid system prototyping. We present an approach that allows users to model the environment in which the SUD will operate in the terms of attributed event grammar (AEG). The AEG provides a uniform approach for automatically generating, executing, and analyzing tests. The approach is supported by a generator that creates test cases from the AEG models. We demonstrate the effectiveness of the proposed approach with using as a case study a prototype of the safety-critical computer-assisted resuscitation algorithm (CARA) software for a casualty intravenous fluid infusion pump.

1 Introduction

The value of computer-aided prototyping in software development is clearly recognized. Feasible timing and safety requirements for the real-time safety-critical reactive systems are difficult to formulate, understand, and meet without extensive prototyping. Bernstein estimated that for every dollar invested in prototyping, one could expect a $1.40 return over the life cycle of the system development [Br]. Many time-series temporal behaviors (e.g., the maximum duration between consecutive missing deadlines must be greater than 5 seconds) and safety requirements (i.e., what the system should not do under abnormal or erroneous conditions) can only be evaluated through the execution of the real-time systems or their prototypes. In [DS1], we showed that the use of run-time monitoring and verification of temporal logic assertions, in tandem with rapid prototyping, aids debugging the requirements and identifying errors early on in the design process.

While a lot of work has been published on languages, methods, techniques and tools to support the rapid creation and evolution of system prototypes, not much work have been done on the rapid development of executable environment models to automate the systematic testing and evaluation of these prototypes.

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].

Most works on test automation are 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]. These approaches are only effective if one has correct requirements and system specifications from the start, which is often not the case in rapid system prototyping. Moreover, 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. Rapid prototyping can be used to gain a better understanding of the environment. To be effective, we need tools to construct and modify these executable environment models rapidly, accurately, and at a reasonable cost.

It has become a common practice for engineers to analyze system behaviors from an external point of view using use cases [JB]. UML use case scenarios, which are written in natural language, 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. We can also derive the environment functional model from the same set

of use cases, where the model receives responses from the system and generates events to the system in response. Event grammars and state machines are means for formalizing the environment models based on system events or responses. Event grammars, which are text based, have a smaller semantic distance from the use case scenarios than the state machines and hence are better suited to model environments described via use case scenarios. Moreover, event grammars are more effective in specifying dynamic environments where there are arbitrary numbers of actors (and concurrent events), whereas state machines are only effective for modeling static environments (with predetermined numbers of actors). Behavior models based on event grammars can be designed for the SUT as well, and used for run-time verification and monitoring. This technique may be used to automate the 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.

The use of context-free grammars for test generation has been discussed in research literature for a long time, 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.

The rest of the paper is organized as follows. Section 2 presents the attributed event grammar notation for specifying the environment models, and Section 3 presents a case study of environment model for the safety-critical computer assisted resuscitation algorithm (CARA) software for a casualty intravenous fluid infusion pump. Section 4 introduces the approach to system under development (SUD) safety assessment based on automated experiments with the environment models. Section 5 provides an implementation outlook for automatic generation of test cases from environment models to facilitate the testing of prototype real-time reactive systems. Section 6 presents a discussion of the approach and draws conclusions.

2 Attributed Grammar Based Environment Models

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 SUD. 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.

Example 1.

```
driving_a_car:
    go_straight
    (* ( go_straight | turn_left | turn_right )  *)
    stop;
```

The composite event **driving_a_car** contains event **go_straight** followed by zero or more events of types **go_straight, turn_left,** or **turn_right**. This sequence is always completed with the event **stop**.

```
go_straight:  ( accelerate | decelerate | cruise );
```

The event **go_straight** contains one of three possible inner events.

This simple event grammar defines a set of possible event traces–a model of 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.

As shown in Example 2, an event may have attributes associated with it. 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.

Events usually have timing attributes like *begin_time*, *end_time*, and *duration*. Some of those attributes can be defined in the grammar by appropriate actions, while others may be calculated by appropriate default rules. For example, for a sequence of two events, the begin time of the second event should be generated larger than the end time of the preceding event.

Example 2.

The interface with the SUD can be specified by an action that sends input values to the SUD. This may be a subroutine in a common programming language like C that hides the necessary wrapping code. In the following example of a simple calculator use case scenario, we suppose that SUD should receive a message about the button pressed by the user from an appropriate wrapper subroutine **enter_digit()**, **enter_operation()**, and **show_result()** correspondingly.

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 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 when 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 SUD.

```
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 SUD is made based on the probability P(prob) assigned to the corresponding alternative.

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

The event **Show_result**, when generated, will cause call to the wrapper subroutine that sends a message to the SUD.

According to [La] a scenario is a specific sequence of actions and interactions between actors and the system under discussion. We can generate large number of **Use_calculator** scenarios (or 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 a 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 "scaffolds" for test driver generation. Separation of generation phase from test execution is essential for the performance of the generated test driver, since the event selection and attribute evaluation can be performed at the generation time the test driver contains only wrapper calls to interact with the SUD.

3 Environment Model for the CARA Software

CARA is a safety-critical software-intensive system developed by the Walter Reed Army Institute of Research to improve life support for trauma cases and military casualties [W1, W2, W3]; it has been used as a case study by several software engineering research groups [AA, LS, DS].

CARA's mission is to monitor a patient's blood pressure and to automatically administer intravenous (IV) fluids via a computer-controlled pump at levels required to restore intravascular volume and blood pressure.

The main responsibilities of the CARA system include:

1. To monitor a patient's blood pressure.
2. To control a high-output patient resuscitation infusion pump.
3. To display (to a human caregiver) vital information about the patient and the system.
4. To log all data.
5. To alert the caregiver to hazardous conditions, such as fluid free-flow.

3.1 The Environment Model

The AEG for CARA's environment model will consist of the following global parameters and event threads.

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**. 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.

```
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 re-

sponsible for sending the pump back EMF voltage measurements to the SUD 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 SUD. The **CATCH** construct in the **Pumping** event thread represents an external event of receiving an input from the SUD. It is implemented as a function **set_pump_voltage(ENCLOSING Pump.voltage)** call, which returns a True value and adjusts its parameter when SUD has issued corresponding output. 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.

3.2 Integrating the Environment Model with the Prototype

Figure 1 shows the top level design of a CARA prototype developed using a PC-based computer-aided environment (called SEATools) to facilitate the analysis and understanding of the CARA requirements [LS]. The SEATools is based on the Prototyping System Description Language (PSDL) [LB, Lu], which is a high-level language designed specifically to support the conceptual modeling of real-time embedded systems.

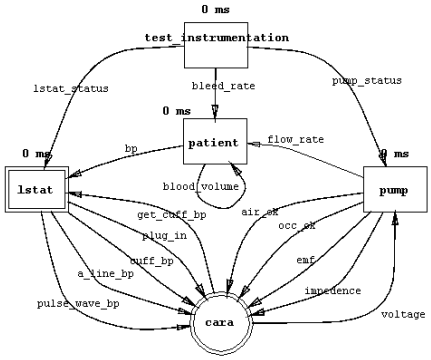

Figure 1. The top level design of CARA prototype

The top level of the CARA prototype consists of five modules, the CARA software and four external components (*lstat*, *patient*, *pump* and *test_instrumentation*).

We can easily convert the manual testing environment into an automated testing environment shown in Figure 2, by replacing the four external components in the top level of the CARA prototype by a single component called the *CARA_environment*, which contains the test driver generated from the AEG model described in Section 3.1.

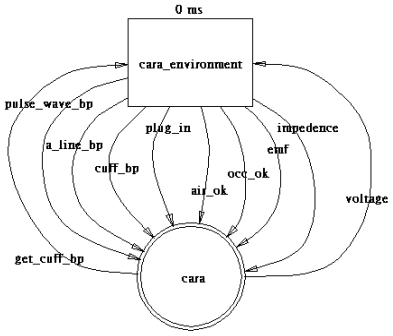

Figure 2. The top level design of revised CARA prototype

4 Safety Assurance

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 safe operation of CARA depends on the timely detection of abnormal situations in the pump lines and the patient's blood pressure. In addition to deciding on automatic or manual pump control according to the run time changing of the patient's blood pressure value, CARA must alert the caregivers to intervene in emergency situations via a set of alarms.

The environment model can contain a description of hazardous states in which the system could arrive, and which could not be derived from the SUD model itself. For example, the **Critical_condition** event will occur in certain scenarios depending on the SUD 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 for the risk of getting to 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 SUD behavior within the given environment model.

We can do qualitative analysis as well. It is possible to ask questions such as "what has contributed to this outcome?" We can change some probabilities in the environment model, or change some parameters in the SUD and repeat the whole set of tests. If the frequency of reaching a hazardous state changes, we can answer the question asked. The changes in the model could be done automatically in some systematic way.

For instance, 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.

5 Implementation Issues for Automatic Test Generator

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 in a similar fashion.
- Attributes are evaluated mostly at the generation time, but those dependent on SUD 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.
- The generated driver contains only simple assignment statements and C subroutine calls for interface with the SUT, guarded by simple flags, hence is very efficient and can be used for real time SUD testing.

6 Conclusions

Traditionally reactive systems and their environments are modeled with some kind of finite state machine, like statecharts or timing automata. For the purpose of scenario (and corresponding test case) generation, the AEG approach may have several useful features, in particular:

- It is based on a sound and powerful 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, hence facilitating the design large models.
- Data flow of attributes is integrated with the control flow (i.e., event trace), and AEG notation provides means for ease of navigation within the derivation tree (e.g., the ENCLOSING event construct, like in **ENCLOSING CARA_environment-> Patient.blood _pressure)**.
- The probabilities for alternatives or number of iterations are attached to meaningful events in the model and are more intuitive and less numerous than in Markov models based on finite state machines.

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 random (but satisfying the constraints) 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 serious requirement errors 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 SUD 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.
- Event traces generated from the AEG model represent examples of SUD interaction with the environment, and are in fact use cases, that could be useful for requirements specification and other prototyping tasks.
- Environment models can be designed early on before the system design is complete, can be run as environment simulation scenarios, and can be used for tuning the requirements and for prototyping efforts.

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.

[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*, IEEE (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] C. Boyapati, C., Khurshid, S. and Marinov, D. Korat: Automated testing based on Java predicates, in *Proc. Int. Symposium on Software Testing and Analysis*, ACM (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.

[Br] Bernstein, L. Forward: Importance of software prototyping, *J. Systems Integration* 6, 1 (1996), pp. 9-14.

[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.

[DS1] Drusinsky, D. and Shing, M. Verification of timing - properties in rapid system prototyping, in *Proc.14th Int. Workshop on Rapid System Prototyping*, IEEE (San Diego, Calif., June 2003), pp. 47-53.

[DS2] Drusinsky, D. and Shing, M. TLCharts: Armor-plating Harel statecharts with temporal logic conditions", *Proc.15th Int. Workshop on Rapid System Prototyping*, IEEE (Geneva, Switz., June 2004), pp. 29-36.

[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.

[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.

[JB] Jacobson, I., Booch, G., and Rumbaugh, J. *The Unified Software Development Process*, Reading, Mass.: Addison-Wesley, 1999.

[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.

[La] Larman, C. *Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and the Unified Process*, Upper Saddle River, N.J.: Prentice Hall, 2002.

[LB] Luqi, Berzins, V., and Yeh, R. A prototyping language for real-time software, *IEEE Trans. Software Engineering* 14, 10 (Oct. 1988), pp. 1409-1423.

[LS] Luqi, Shing, M., Puett, J., Berzins, V., Guan, Z., Qiao, Y., Zhang, L., Chaki, N., Liang, X., Ray, W., Brown, M., and Floodeen, D. Comparative rapid prototyping, A case study, in *Proc. 14th Int, Workshop on Rapid System Prototyping*, IEEE (San Diego, Calif., June 2003, pp. 210-217.

[Lu] Luqi. Real-time constraints in a rapid prototyping language, *J. Computer Languages* 18, 2 (Spring 1993), pp. 77-103.

[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.

[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.

[W2] WRAIR Dept. of Resuscitative Medicine, CARA Pump Control Software Questions (Version 6.1), proprietary document, WRAIR, Silver Spring, Md., Jan. 2001.

[W3] WRAIR Dept. of Resuscitative Medicine, CARA Tagged Requirements, Increment 3 (Version 1.2), proprietary document, WRAIR, Silver Spring, Md., Mar. 2001.