2006 CCRTS

The State of the Art and the State of the Practice

Title of Paper:	New Directions in C2 Software Quality Assurance Automation Based on Executable Environment Models
Submission #:	C-125
Topics:	C2 Concepts and Organizations, C2 Analysis, C2 Modeling and Simulation

Authors: Mikhail Auguston, James Bret Michael, Man-Tak Shing,

Point of Contact: Mikhail Auguston

Name of Organization: Naval Postgraduate School

Complete Address: Department of Computer Science, Naval Postgraduate School, 833 Dyer Road, Monterey, California, 93943-5118, USA

Telephone/Fax: 831-656-2607 / 831-656-2814

Email: maugusto@nps.edu

Report Documentation Page		*Form Approved* *OMB No. 0704-0188*

1. REPORT DATE **JUN 2006**	2. REPORT TYPE	3. DATES COVERED **00-00-2006 to 00-00-2006**
4. TITLE AND SUBTITLE **New Directions in C2 Software Quality Assurance Automation Based on Executable Environment Models**		5a. CONTRACT NUMBER
		5b. GRANT NUMBER
		5c. PROGRAM ELEMENT NUMBER
6. AUTHOR(S)		5d. PROJECT NUMBER
		5e. TASK NUMBER
		5f. WORK UNIT NUMBER
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) **Naval Postgraduate School,Department of Computer Science,833 Dyer Road,Monterey,CA,93943-5118**		8. PERFORMING ORGANIZATION REPORT NUMBER
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)		10. SPONSOR/MONITOR'S ACRONYM(S)
		11. SPONSOR/MONITOR'S REPORT NUMBER(S)
12. DISTRIBUTION/AVAILABILITY STATEMENT **Approved for public release; distribution unlimited**		
13. SUPPLEMENTARY NOTES **The original document contains color images.**		
14. ABSTRACT		
15. SUBJECT TERMS		

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT **unclassified**	b. ABSTRACT **unclassified**	c. THIS PAGE **unclassified**		**40**	

Standard Form 298 (Rev. 8-98)
Prescribed by ANSI Std Z39-18

New Directions in C2 Software Quality Assurance Automation Based on Executable Environment Models

Mikhail Auguston, James Bret Michael, Man-Tak Shing

Naval Postgraduate School, Department of Computer Science, Monterey, CA USA
{maugusto,bmichael,shing}@nps.edu

Abstract. This paper presents some concepts, principles, and techniques for automated testing of real-time reactive software systems based on attributed event grammar (AEG) modeling of the environment in which a system will operate. AEG provides a uniform approach for automatic test generation, execution, and analysis. Quantitative and qualitative assessment of the system comprised of the software under test and its interaction with the environment, can be performed based on statistics gathered during automatic test execution within an environment model.

1. Challenges to C4ISR system software testing

Modern C4ISR systems designed to support network-centric warfare are typically large, heterogeneous, distributed systems-of-systems (SoS). The individual systems making up a C4ISR system-of-systems (i.e., the component systems) may have each been developed for different contexts of use and subjected to different sets of constraints than those of the system-of-systems. When assembled together as a SoS, these component systems are expected to work together to provide emergent services, that is, services that cannot be achieved by any of the systems working in isolation of one another.

The need to support *real-time retargeting* and *remote-fire* has transformed modern C4ISR systems into real-time, reactive systems. Some of the component systems must continuously interact with their environment under tight time budgets. Both the inputs and outputs of these component systems must satisfy timing constraints imposed by the SoS requirements, which may not be present in the original functional and dependability requirements[1] of the component systems.

Testing of a SoS necessitates an understanding of the SoS's operating environment and the interactions between its component systems. Modern C4ISR systems require a new level of testing to discover emergent behaviors resulting from the composition of legacy and new elements or components, and to study the quality of the systems-of-systems under different environmental conditions. One needs to test the SoS for both desired behaviors and those behaviors that are undesired or unanticipated, all of which can only be assessed at the SoS level; that is, it is not possible to test each of the component systems in isolation of one another and then draw any conclusions about the emergent behaviors of the SoS.

[1] We use the term "dependability" in this paper to refer to the spectrum of nonfunctional requirements, such as those concerned with safety, reliability, security, performance, or quality of service.

In its purest form, black-box testing does not require any knowledge on the part of the tester of the design or implementation details of the software under test; test cases are derived from the system requirements specifications. The software is executed on test data and the output is compared with the expected output (via a test oracle). A discrepancy indicates the presence of a fault in the system under test (SUT). Testing is a resource-intensive activity. The key questions to be answered when planning to conduct black-box testing, while keeping resource constraints in mind, are the following:

- How will test cases be created?
- How will test cases be exercised (i.e., run)?
- How will the results of exercising the test cases be analyzed?

Creating and exercising test cases, in addition to analyzing the results of exercising the test cases, can to a great extent be automated: much of these tasks are mechanical in nature, the steps of which can be represented as sets of rules. We contend that testing automation is necessary in order to assess large complex C4ISR SoS that may also be evolving at a relatively rapid rate, but testing automation is of little value if the formal model of the SoS's operating environment[2] is incorrect.

Modeling is generally understood as crucial in the development of high-quality complex software systems. Models provide developers with a means to gain insight into problems and solutions, select and use tools as appropriate, and manage complexity. For example, a common approach to verifying safety requirements involves developing two separate models: one for the SUT and the other for the environment (or equipment) under its control. The two models are then exercised in tandem to see if the simulation ends up in unknown hazardous states under normal operating conditions and under various failure conditions [1]. However, models are often under-utilized because it is not always clear whether modeling constructs can capture useful abstraction of a system and "current model development mechanisms do not facilitate timely creation and evolution of models" [7].

We suggest the following approach for creating and running test cases in automated black-box testing of real-time reactive systems (Figure 1). The model of the environment in the form of attributed event grammar (AEG) specifies the behavior of the environment in terms of events relevant from the point of view of the SUT; this grammar is used for random event trace generation. An event trace is a set of events with a certain structure. The generated event traces are not completely random since they fulfill constraints embedded in the environment model. Event attributes provide inputs to the SUT, and the event trace structure facilitates the necessary timing constraints. The test driver (e.g., a C program) can be derived from the given event trace.

[2] In this paper we refer to the environment as everything outside the engineering design space, that is, everything outside the sphere of control of the SUT.

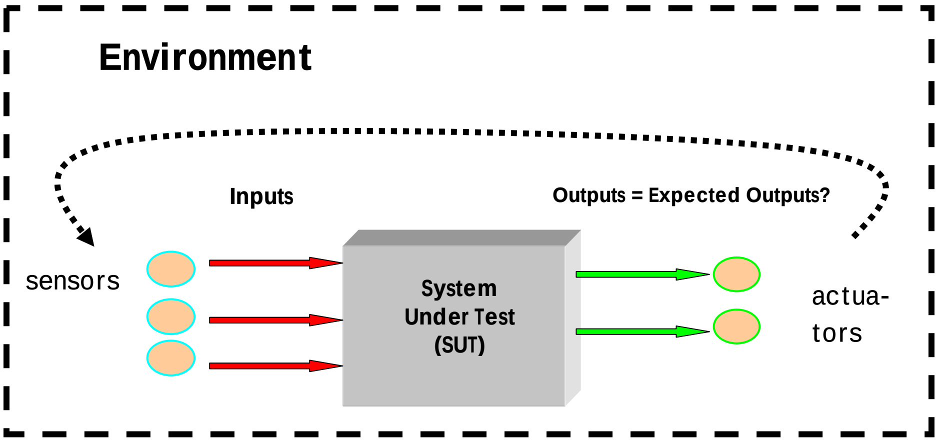


Figure 1. Black Box Testing Environment

A lot of research has been done regarding test oracles and behavior monitoring of the SUT. It is common practice for engineers to describe system behaviors from an external point of view using Unified Modeling Language (UML) use cases[3] [12]. Natural language specifications of use case scenarios focus on the events and responses between the actors and the system. Event grammars and state machines are two common means for formalizing the environment models based on these system events/responses. Moreover, 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. Behavior models based on event grammars can be designed for the SUT too, and used for runtime 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 [2], [3], and [4] and will not be discussed in this paper.

In this paper we demonstrate how expected SUT outputs could be incorporated into the environment model. This allows generated test cases to interact with the system and adjust the evolving event trace based on the results of that interaction. The environment model can contain descriptions of hazardous states in which SUT could arrive. Thus it becomes possible to conduct experiments with the SUT in the simulated environment and gather statistical data about the behavior of SUT in order to estimate operational effectiveness, safety and other dependability properties of the SUT. Furthermore, by changing the values of parameters of the environment model, for example adjusting frequencies of some events in the model, and running experiments with the adjusted model, we can identify dependencies between environment parameters and the behavior of the system. Our approach is to integrate the SUT into the environment model, and to use this model for both testing of the SUT in the simulated environment and assessing risks posed by the SUT.

The approach to testing automation described in this paper may be applied to a wide range of reactive systems, including C4ISR and other domains, where environment models can be defined to specify typical scenarios and functional profiles. The first prototype of the AEG test driver generator has been implemented at NPS and used for several case studies. Parameters of the AEG model, such as probabilities of alternative events and iterations can be adjusted in order to generate both load testing and stress scenarios. While the AEG formalism can be used for specifying environment

[3] According to [5], a UML *use case* is “a description of a set of sequences of actions, including variants, that a system performs that yields an observable result of value to an actor.” An *actor* is “a coherent set of roles that users of use cases play when interacting with the use cases.”

models for any SUT, it is most effective for specifying test scenarios and generating large numbers of pseudo-random test drivers for reactive and real-time system automated testing.

2. The environment model

The notion of event is central to our approach. An **event** 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. In our approach 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.

There are two basic relations defined for events: **precedence** (PRECEDES) and **inclusion** (IN). Two events may be ordered, 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**). Usually event traces have a certain structure (or constraints) in a given environment. The basic relations define a partial order of events. Two events are not necessarily ordered, that is, they can happen concurrently.

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, {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).

Example 1 (adapted from [16]).

OfficeAlarmSystem::= { DoorMonitoring, WindowMonitoring }
The *OfficeAlarmSystem* run is a set of two concurrent monitoring threads.

*DoorMonitoring::= DoorSensor **
The *DoorMonitoring* is a composite event, which contains a sequence of ordered events of the type *DoorSensor.*

*WindowMonitoring::= WindowSensor **

DoorSensor::= (DoorClosed | DoorAlarm)
The *DoorSensor* event may contain one of two possible alternatives.

WindowSensor::= (WindowClosed | WindowAlarm)

This 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.

3. Event attributes

An event may have attributes associated with it. Each event type may have a different attribute set. Event grammar rules can be decorated with attribute evaluation rules. This is similar to the notion of traditional attribute grammar [15]. 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.

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 a car-race monitoring system, we assume that SUT's sensors should receive a time of each car's *start*, *Lap*, *finish* events, and in the case when the car drops out from the race, the time of *drop_out* event from an appropriate test wrapper subroutines *send_start(),send_intermediate(), send_finished(),* and *send_drop_out (),* correspondingly.

Example 2.

CarRace::= { /Car.id := unique_id()/ Car } (<=10)*
/CarRace.final_count:= Number(Car, ***such that*** *Car.completed)/*

The attribute *id* of the event *Car* is assigned a value before the event appears on the trace, that is, it is propagated from the parent event *CarRace.* The *(<=10)* guard sets the upper limit for the number of events in the *{…}** iterator. The *Number* aggregate operation provides the number of *Car* events satisfying the condition within the scope of *{…}** iterator.

Car ::= start / send_start(Car.id, start.begin_time); Car.completed := True/
Lap (=5)*
When*(Car.completed) (finish /send_finished(Car.id, finish.end_time)/)*

The *When* guard terminates generation of the following event, *finish,* if the guard condition, *Car.completed,* becomes False. The *(=5)* guard sets the number of events in the iterator.

```
Lap ::= (    single_loop / send_intermediate(Car.id, single_loop.begin_time)/  |
             drop_out     / ENCLOSING Car.completed := False;
                            send_drop_out (Car.id, drop_out.end_time);
                            BREAK /
        )
```

The *ENCLOSING* construct provides access to the attributes of parent events. The *BREAK* action terminates the enclosing iteration. Attributes can be both inherited and synthesized [15]. We assume that all attribute evaluations are accomplished in a single pass. The AEG in Example 2 can be used in order to generate event traces with more constraints. In addition, some events in the generated trace will have attribute values obtained from the actions embedded in the grammar and *send* actions indicating that certain inputs should be fed into the SUT immediately after the preceding event.

4. Test generation

The purpose of the attribute event grammar discussed above is to provide a vehicle for generating event traces. Iterations can be constrained by use of guards. For alternatives we can provide the probability of selecting an alternative.

Example 3.

```
Lap ::=    (  p(0.8)  single_loop
                  / send_intermediate(Car.id, single_loop.begin_time)/    |
              p(0.2)  drop_out
                  / ENCLOSING Car.completed := False;
                    send_drop_out (Car.id, drop_out.end_time);
                    BREAK /
           )
```

Probabilities assigned to the alternatives determine the frequency of corresponding event generation.

Merging together Examples 2 and 3, we obtain a model of a *CarRace* and are in position to generate any number of scenarios. Each event trace will satisfy the constraints imposed by the event grammar. Some events are accompanied by the *send* action and have the timing attributes calculated during the trace generation. Such a trace could be transformed into a test driver (no pun intended) which will feed the SUT with the values according to the timing constraints. From the test generation point of view, the event trace is just a “scaffold” for determining the sequence and timing for the *send* actions which provide actual inputs for the SUT.

4.1. SUT outputs incorporated into the environment model

The behavior of the environment can be affected by the outputs from the SUT. The following (oversimplified) example of a missile defense system demonstrates how to incorporate an interaction with the SUT into AEG. We assume the SUT tracks the launched missile movement by receiving data from a radar sensor (*send_radar_signal()* action in the model simulates radar sensor inputs to the SUT), and at a certain moment makes a decision to fire an anti-missile (i.e., interceptor) by generating an output to a corresponding actuator (*SUT_launch_interception()*).

The *catch* construct represents an external event generated at runtime by SUT. The external event listener is active during the execution of a test driver obtained from the generated event trace. This particular external event is broadcast to all corresponding event listeners. The following event grammar specifies a particular set of scenarios for testing purposes.

Example 4.

```
Attack::= { Missile_launch } *
    The Attack event contains several parallel Missile_launch events.

Missile_launch::= Boost_stage
                       / Middle_stage.completed := True/  Middle_stage
                       When(Middle_stage.completed) Boom
```

The *Boom* event (which happens if the interception attempts have failed) represents an environment event, which the SUT in this case should try to avoid.

```
Middle_stage::= ( move |
                       catch   SUT_ launch_interception(hit_coordinates)
                      When(hit_coordinates == Middle_stage .coordinates )
                             [ p(0.1) interception
                                    / Middle_stage.completed := False;
                                      send_hit_input(Middle_stage .coordinates);
                                      Break /   ]
                      ) *
```

The sequence of *move* events within *Middle_stage* event may be interrupted by receiving an external event from the SUT. This will suspend the *move* event sequence and will either continue with event *interception* (with probability 0.1), which simulates the missile-interception event triggered by the SUT, followed by the *Break* command, which terminates the event iteration, or will resume the *move* sequence. This model allows several interception attempts through the same missile launch event. For simplicity it is assumed that there is no delay between receiving the external event *SUT_launch_interception(hit_coordinates)* and the possible *interception* event.

move ::= /adjust(ENCLOSING Middle_stage .coordinates) ;
send_radar_signal(ENCLOSING Middle_stage.coordinates);
move.duration:= 1 sec /

This rule provides attribute calculations and sends an input to the SUT. In general, external events (i.e., events generated by the SUT) may be broadcasted to several event listeners in the AEG, or may be declared as exclusive and will be consumed by just one of the listeners. It may happen that there is not a listener available when an external event arrives. This usually indicates presence of an error in the environment model and can be detected and reported at the test execution time. To alleviate this problem, AEG may contain a mechanism similar to an exception handler for processing external events which have missed regular event listeners.

5. Assessment of Operational Effectiveness and Safety

As mentioned in Section 1, many of the component systems of modern C4ISR systems have to continuously interact with their environment and control other weapon systems under tight timing constraints, making operational effectiveness and safety top requirements of modern C4ISR systems. Our approach to modeling operational effectiveness and system safety relies on modeling operational and safety hazards.[4] We use environment models to exercise the SUT so that we can assess the effects of each type of known operational or safety hazard will have on the operational effectiveness or safety of the SUT.

An environment model may contain events and attributes that could not be derived from the SUT model itself. In the previous example, the *Boom* event (in this example a mishap attributable to an operational hazard) occurs in certain scenarios depending on the SUT outputs received by the test driver and random choices determined by the given probabilities. From the point of view of the SUT, the *Boom* event is a highly undesirable.

The data gathered from running a statistically significant number of (automatically generated) tests provides some approximation for the risk of getting to this hazardous state and thus a measure of the operational effectiveness (e.g., hit-to-kill rate for ballistic missile interception) of the SUT. This becomes a constructive process of performing experiments with SUT behavior within the given environment model.

[4] Safety hazards are those conditions that can result in mishaps (i.e., losses) by actions of our systems. An example of a safety hazard is death or injury to shipboard or friendly personnel due to premature detonation of a missile warhead. In contrast, operational hazards are mishaps inflicted by outside systems, particularly hostile forces. For example, failure of the missile or other weapon to engage and destroy an incoming threat is an operational hazard.

We can do a qualitative analysis as well and ask questions like “what has contributed to this outcome?” We can change some probabilities in the environment model, or change some parameters in the SUT and repeat the whole set of tests. If the frequency of reaching a hazardous state changes, we can answer the question asked. These kinds of experiments with model parameters can be done automatically in a systematic way.

Example 5.

```
Attack::= { Missile_launch } * (<=N)

Missile_launch::= boost   Middle_stage   Boom

Middle_stage::= ( move |
                    catch SUT_output(hit_coordinates)
                    [ p(p1) interception
                            / send_hit_input(Middle_stage .coordinates);
                             Break  /  ]
                  )*
```

Experimenting with increasing or decreasing the number of missile launches *N* and probability of interception *p1*, we can determine what impact those parameters have on the probability of hazardous outcome, and find thresholds for SUT behavior in terms of *N* and *p1* values.

Environment models for realistic domains will have a significant number of parameters. Exhaustive testing of all combinations of parameter values is not pratical. Instead, we apply the methodology in a manner similar to that employed in combinatorial testing [5], [6]. Rather than examine all possible combinations of parameter values within certain intervals, our strategy is to use the test generation to try to cover all possible pairs of parameter values; by doing so we can significantly reduce the amount of required testing. Experience with combinatorial testing for data-driven applications demonstrates that such reduced test suites still provide a reasonable coverage in terms of error detection.

6. Test Code Generator

The environment model defined by AEG can be used to generate random event traces, where events will have attribute values attached, including time attributes. 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 functionality of this generated test driver is limited to feeding the SUT inputs and receiving outputs and may be implemented as an efficient C or even assembly language program that meets strict real-time requirements. Only *send* and *catch* actions obtained from the event trace are needed to construct the test driver; the rest of events in the event trace are used as “scaffolds” to obtain the ordering, timing and other attributes of these actions. The first prototype of the automatic test driver generator from AEG environment models has been implemented at NPS. The generator takes as input the AEG model and outputs random event traces. Necessary actions are then extracted from the trace and assembled into a test driver in the C programming language.

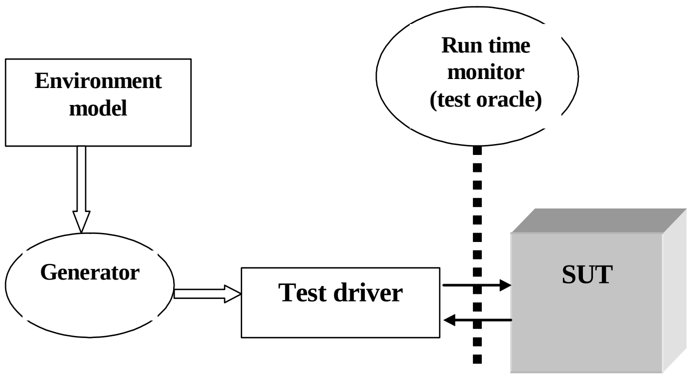


Figure 2. The architecture of automated test generator

The main advantages of the outlined approach are as follows:

- Environment model specified by AEG provides for automated generation of a large number of random (but satisfying the model constraints) test drivers.
- It addresses the regression testing problem: generated test drivers can be saved and reused.
- The testing tool can be adjusted to the changing requirements by adjusting the event grammar.
- The generated test driver contains only a sequence of calls to the SUT, external event listeners for receiving the outputs from SUT, and time delays where needed to fulfill timing constraints. Hence it is quite efficient and could be used for real-time test cases.
- Different environment models for different purposes can be designed, for example, for testing extreme scenarios by increasing probabilities of certain events.
- Experiments with the environment model running with the SUT provide a constructive method for quantitative and qualitative software risk assessment.
- Environment models can be designed in early stages, before the system design is complete and can be used as an environment simulation tool for tuning the requirements and prototyping efforts. The generated event traces can be considered as use cases that may be used for requirements specification on early stages of system design.

7. Related and future work

The use of context-free grammars for test generation has been discussed in the research literature for a long time, in particular to check compiler implementation (e.g., see [14]). [13] provides an outlook in the use of enhanced context-free grammars for test data generation.

Significant work has been done on automated test generation from the formal system specifications, such as in the form of finite state machines [19], [17], StateCharts [9], timed automata [11], hybrid automata [18], or UML design specifications [10]. This could be regarded as a white-box approach on different levels of abstraction for SUT specification, often targeting some kind of branch coverage criteria for the formal specification under consideration. Our approach supplements these efforts and differs in the emphasis on modeling the environment of the reactive SUT, treating the SUT as a black box, as opposed to modeling the SUT itself. We recommend the use of attribute event grammars as a framework for random test case generation. It may be worth mentioning that the AEG branch coverage criteria may be of interest as a metric of the suggested method.

Some directions for future work include the following topics:

- In order to feed the generated inputs from the test driver to the SUT and catch SUT outputs of interest for the model, a special set of wrappers or bridges should be provided.
- The test driver generator can be designed to enforce grammar branch coverage to ensure that all grammar alternatives have been traversed.
- The generated test driver can interact with the test oracle or the run-time monitor to support the integrity of the testing process.
- The suggested tool supports automated software risk assessment, both quantitative and qualitative, by automatically generating large numbers of randomly generated tests. It can gather the statistics of reaching hazardous states and can perform a series of targeted experiments to determine dependencies of test results on the model parameters, such as frequencies of specific types of events.

Acknowledgements

The research reported in this article was funded 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. The U.S. Government is authorized to reproduce and distribute reprints for Government purposes notwithstanding any copyright annotations thereon.

References

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

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

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

[4] Auguston, M., Jeffery, C., and Underwood, S. A framework for automatic debugging, in *Proc. of the Seventeenth Int. Conf. on Automated Software Engineering*, IEEE (Edinburgh, Scotland, Sept. 2002), pp.217-222.

[5] Booch, G., Rumbaugh, J., and Jacobson, I. The Unified Modeling Language User Guide.

[6] Burr, K. Combinatorial test techniques: Table-based automation, test Generation and code coverage, in *Proc. Int. Conf. on Software Testing, Analysis, and Review*, Orange Park, Fla.: Software Quality Engineering, Inc. (San Diego, Calif., Oct. 1998).

[7] Dalal, S.; Jain, A., Karunanithi, N., Leaton, J., and Lott, C. Model-based testing of a highly programmable system, in *Proc. Int. Symposium on Software Reliability Engineering*, IEEE (Paderborn, Ger., Nov. 1998), pp. 174-179.

[8] France R. B., Ghosh, S., and Turk, D. Supporting effective software modeling, *L'Objet Software, Databases, Networks J.* 9, 4 (2003): 11-30.

[9] Harel, D. and Gery, E. Executable object modeling with statecharts, in *Proc. Eighteenth Int. Conf. on Software Engineering*, IEEE (Berlin, Ger., Mar. 1996), pp. 246-257.

[10] Hartmann, J. Vieira, M., Foster, H., and Ruder, A. UML-based test generation and execution, white paper, Siemens Corporate Research, Princeton, N.J., June 4, 2004.

[11] Hessel, A., Larsen, K. G., Nielsen, B., Pettersson, P., and Skou, A. Time-optimal real-time test case generation using UPPAAL, in *Lecture Notes in Compute Science*, no. 2931 (Proc. Third Int. Workshop on Formal Approaches to Testing of Software), Berlin: Springer-Verlag, 2004, pp. 114-130.

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

[13] Maurer, P., Generating Test Data with Enhanced Context-free Grammars, *IEEE Software*, July 1990, pp.50-55

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

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

[16] Sommerville, I. *Software Engineering*, Seventh Ed., Harlow, England: Addison-Wesley, 2004

[17] Tahat, L. H., Vaysburg, B., Korel, B., and Bader, A. J., Requirement-based automated black-box test generation, in *Proc. 25th Annual Int. Conf. on Computer Software and Applications*, IEEE (Chicago, Ill., Oct. 2001), pp.489-495.

[18] Tan, L., Kim, J., and Lee, I. Testing and monitoring model-based generated program, *Electronic Notes in Theoretical Computer Science,* no. 89, issue 2, Berlin: Springer-Verlag, 2003.

[19] Tan, Q. M. and Petrenko, A., Test generation for specifications modeled by input/output automata, in Petrenko, A. and Yevtushenko, N., eds., *Testing of Communicating Systems, IFIP TC6 11th Int. Workshop on Testing Communicating Systems*, Boston: Kluwer Academic Publishers (Tomsk, Russia, Aug. 1998), pp. 83-100.

New Directions in C2 Software Quality Assurance Automation

M. Auguston, J.B. Michael and M.T. Shing

Naval Postgraduate School

Acknowledgement and Disclaimer

- The research was funded in part by a grant from the U.S. Missile Defense Agency.
- The views and conclusions in this talk 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.

Outline

- Challenges to C4ISR system software testing
- Automated test generation based on environment models
- Software safety assessment
- Conclusion

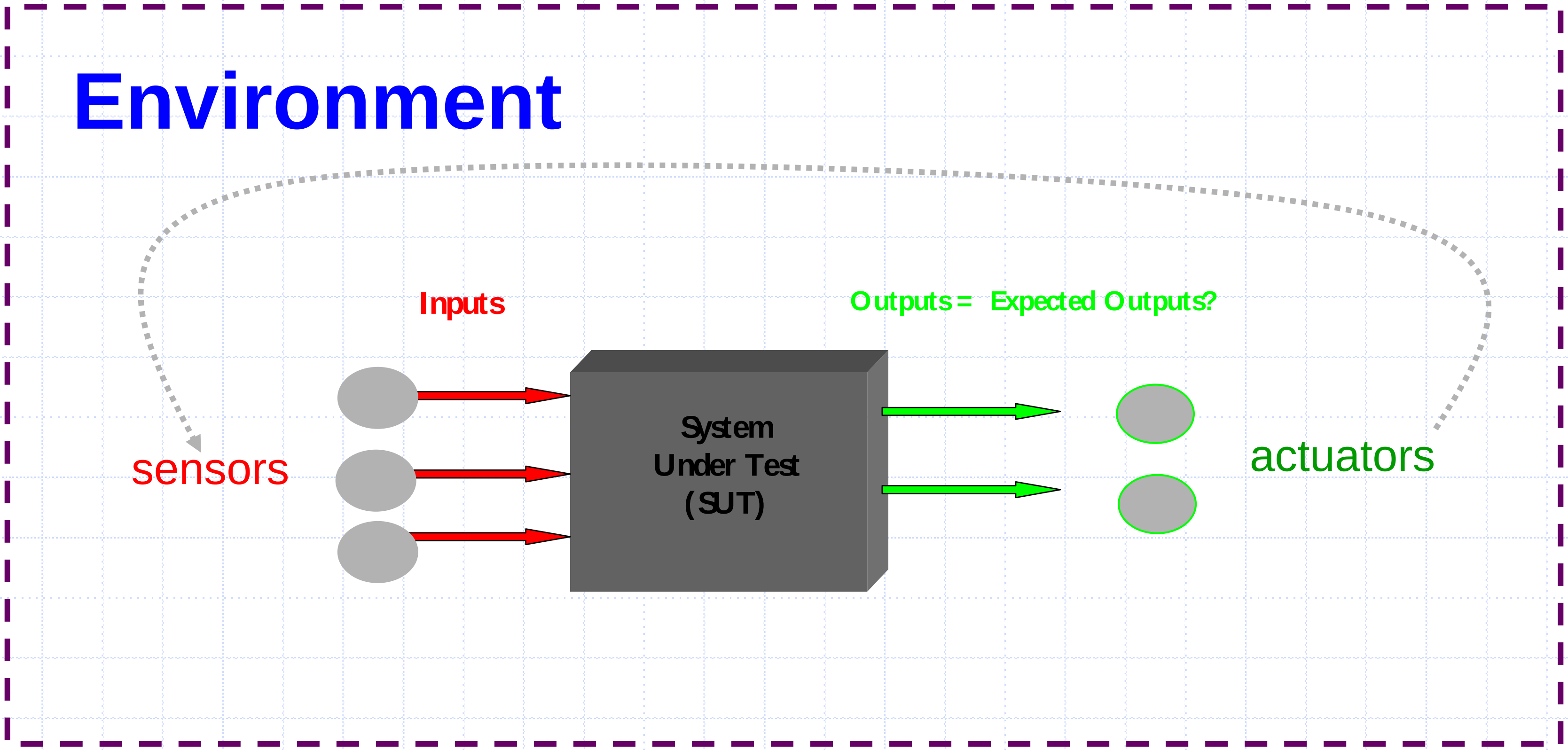
C4ISR Net-centric System-of-Systems (SoS) Characteristics

- Typically Large, heterogeneous, distributed
  - Contains time-critical, safety-critical, reactive component systems
- Evolving
  - Includes legacy systems as well as systems under development
  - Integrate component systems work together to provide greater capability than that of component systems

Net-centric SoS Software Testing Challenges

- Emergent behaviors (both desirable and undesirable) can only be observed from the interactions between the SoS and its operating environment and the interactions between its component systems
- Good environment models are essential for testing SoS software

Black Box Testing



The SUT may be a complex reactive real-time C4ISR system

Black Box Testing (cont'd)

The main problems:

- How to create test cases
- How to run a test case
- How to verify the results of a test run

Testing methodology (How to create test cases)

Three possible approaches:

- Test cases should be carefully designed using “white box” (e.g., branch coverage) or “black box” (e.g., equivalence partition, boundary conditions) methods. This is like “sharp-shooting” for bugs…
- Test cases may be generated at random. This is like a “machine gun” approach…

Testing methodology (cont'd)

- We suggest an "intelligent" random generation based on the environment models.
  - It is best suited for a very special class of programs: reactive and real-time.

The Model of Environment

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

- An event is a time interval
- An event has attributes; e.g., type, timing attributes, etc.
- There are two basic relations for events:

  precedence and **inclusion**
- The behavior of environment can be represented as a set of events (event trace)

The Model of Environment (cont'd)

- Event traces are essentially use case scenarios
  - Examples of event traces can be useful for requirements engineering, prototyping, and system documentation
- Usually event traces have a certain structure (or constraints) in a given environment

Example: driving_a_car is an event that may be represented as a sequence of zero or more events of types

go_straight, turn_left, turn_right, or stop

The Model of Environment (cont'd)

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

Example:

- driving_a_car ::=
  go_straight
  (go_straight | turn_left | turn_right) *
  stop
- go_straight ::=
  (accelerate | decelerate | cruise)

Sequential and Parallel Events

- The precedence relation defines the partial order of events
  - Two events are not necessary ordered; i.e., they can happen concurrently

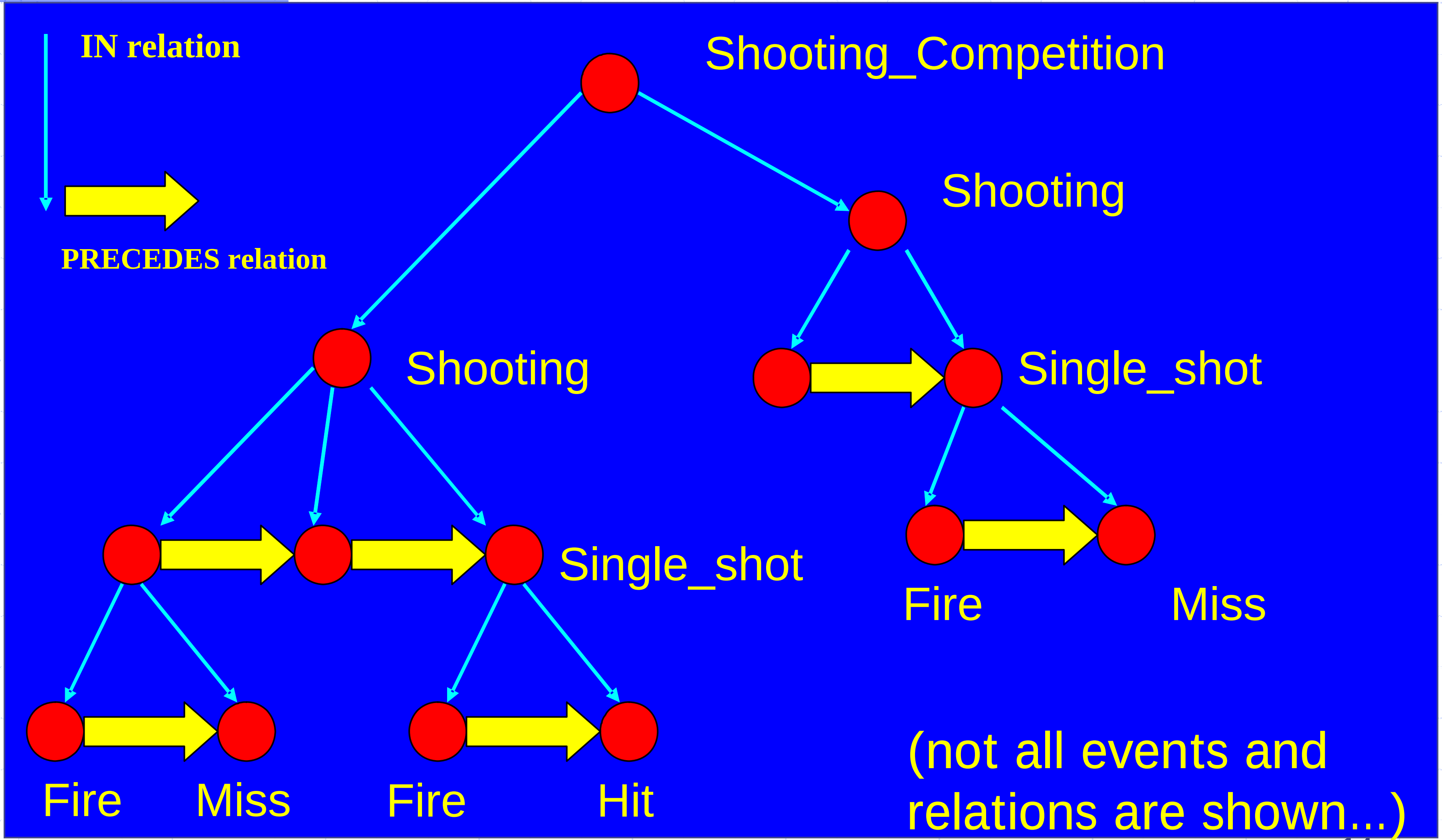
Example:

- Shooting_Competition ::= {* Shooting *}
- Shooting ::= (* Single_shot *)
- Single_shot ::= Fire (Hit | Miss)

Those events may be parallel

This is a sequence

Visual Representation of Event Trace



(not all events and relations are shown...)

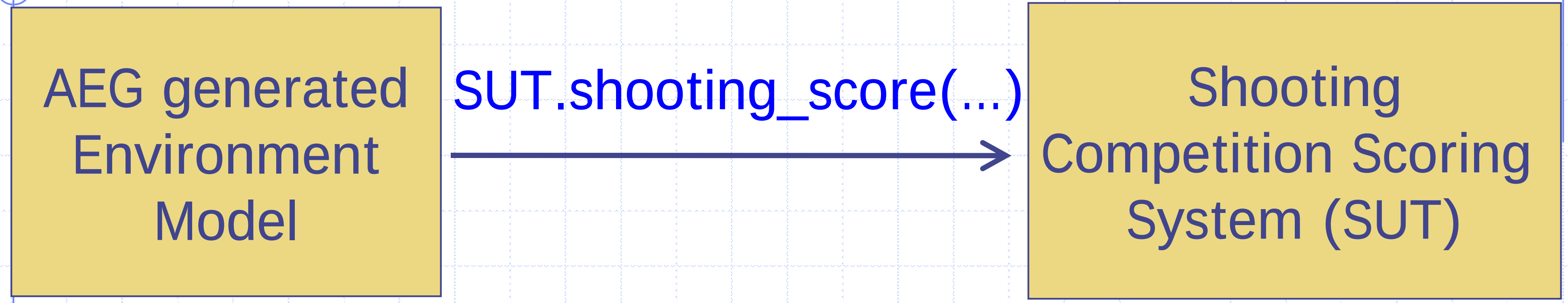
Event attributes

```
Shooting_Competition ::= /num = 0;/
{* /Shooting .id = num++; Shooting .ammo =10;/
    Shooting *} (Rand[2..100])

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

Single_shot ::=  Fire  (
    P(0.3) Hit  /Single_shot. points = Rand[1..10];
                  ENCLOSING Shooting .points
                       += Single_shot .points; /
|   P(0.7) Miss /Single_shot. points = 0;/ )
```

Attribute Event Grammar (AEG)

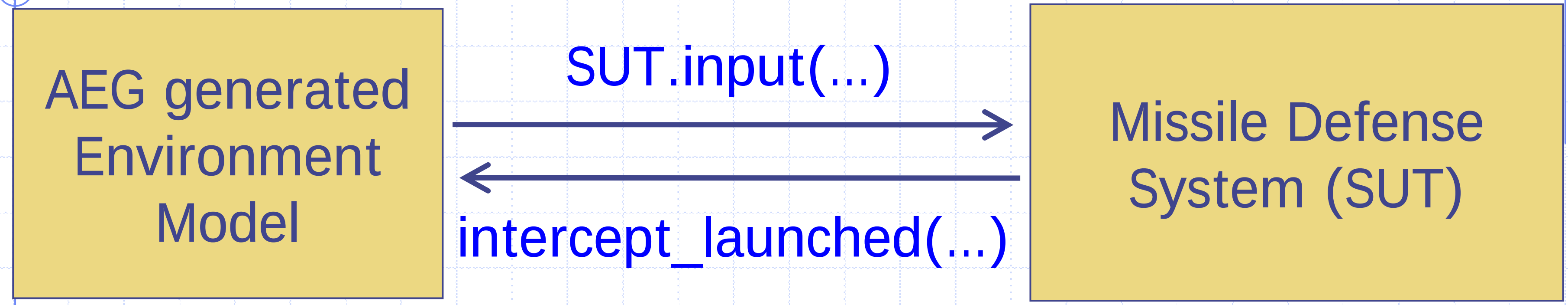
- Intended to be used as a vehicle for automated random event trace generation
  - The AEG is traversed top-down and left-to-right and only once to produce a particular event trace
  - Randomized decisions about what alternative to take and how many times to perform the iteration should be made during the trace generation
  - Attribute values are evaluated during this traversal

Sending Input to System-Under-Test (SUT)


AEG generated Environment Model
SUT.shooting_score(...)
Shooting Competition Scoring System (SUT)

```
Single_shot ::=  Fire  (
        Hit  /Single_shot. points = Rand[1..10];
              ENCLOSING Shooting .points
                                   += Single_shot .points;
              SUT.shooting_score(
                  ENCLOSING Shooting .id, Hit .time);/
    |   Miss /Single_shot. points = 0;/ )
```

Catching outputs from SUT



Attack ::= {* Missile_launch *} (<=N)

Missile_launch ::=

boost_stage / middle_stage.completed = true;/
middle_stage When(middle_stage.completed)
boom

Catching outputs from SUT (cont'd)

```
middle_stage ::=
 (* CATCH intercept_launched (hit_coordinates)
          -- this external event intercepts SUT output
    When (hit_coordinates == middle_stage .coordinates )
       [  P(p1) hard_hit
                  / middle_stage.completed= false;
                    SUT.input(middle_stage .coordinates);
                    -- this simulates SUT sensor input /
          Break; -- breaks the iteration ]
    OTHERWISE  move *)
```

Catching outputs from SUT (cont'd)

```
move ::=
   /adjust (ENCLOSING middle_stage .coordinates) ;
    SUT.input(
               ENCLOSING middle_stage .coordinates);
               -- this simulates SUT sensor input
    DELAY(50 msec); /
```

Software Safety Assessment

The environment model can contain description of hazardous states in which system could arrive, and which can not be easily retrieved from SUT requirements specifications

- For example, the boom event will occur in certain scenarios depending on the SUT outputs received by the test driver and random choices determined by the given probabilities

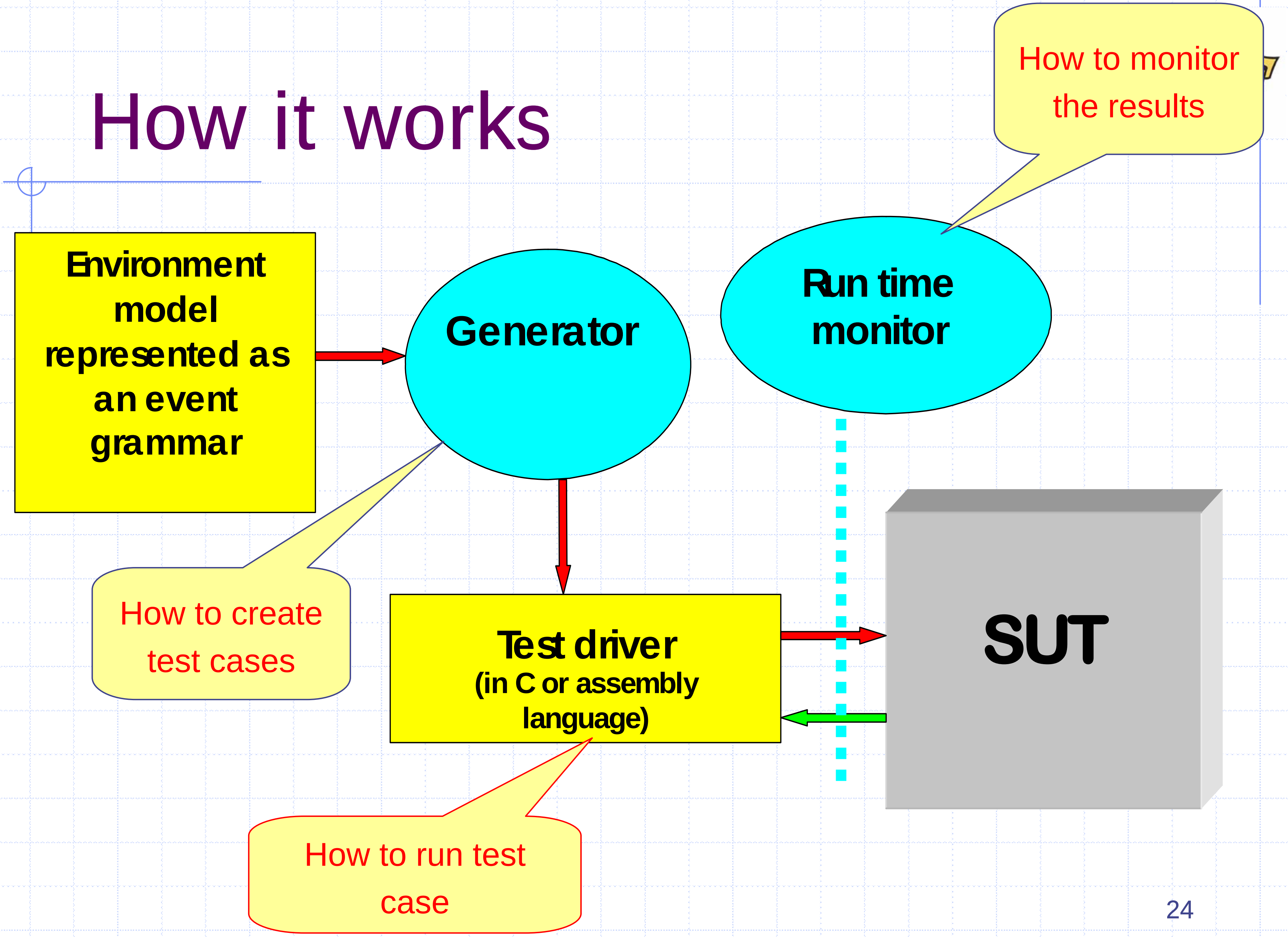
Software Safety Assessment (cont'd)

- If we run large enough number of (automatically generated) tests, the statistics gathered gives some approximation for the risk of getting to the hazardous state.
- By varying the probabilities in the environment model, or changing some parameters in the SUT and repeating the whole set of tests in a systematic way, it is possible to answer questions, such as "what has contributed to this outcome?"

Software Safety Assessment (cont'd)

- This becomes a very constructive process of performing experiments with SUT behavior within the given environment model
- The process is supported by automated test case generation and runtime monitoring of test output

How it works


How to monitor the results
Environment model represented as an event grammar
Generator
Run time monitor
How to create test cases
Test driver (in C or assembly language)
SUT
How to run test case

Conclusion

- The main advantage of the proposed approach
  - Whole testing process can be automated
  - The AEG formalism provides powerful high-level abstractions for environment modeling
  - AEG is well structured, hierarchical, and scalable

Conclusion (cont'd)

- It is possible to run many more test cases with better chances to succeed in exposing an error
- It addresses the regression testing problem – generated test drivers can be saved and reused.
- The environment model itself is an asset and could be reused