Rapid Simulation Evaluation from Scenario Specifications for Command and Control Systems

Raymond A. Paul, W. T. Tsai*, John S. Mikell

Department of Defense
Washington, DC
Telephone: 703-607-0649
raymond.paul@osd.mil

*Department of Computer Science and Engineering
Arizona State University
Tempe AZ, 85287 USA
wtsai@asu.edu

Abstract

This paper presents a technique to simulate and evaluate a system once the system scenarios are available without any simulation programming. This is different from traditional simulation where simulation code and the system specification are separately developed by human engineer and potential gaps between them might be introduced. Another significant advantage of this approach is that the scenarios specified do not need to be complete or consistent. Inconsistency and incompleteness, as well as safety, performance, and behavior problems, can be detected by the simulation via various dynamic analyses. This technique is a part of Scenario-Driven System Engineering (SDSE) that is being developed for Command-and-Control systems.

Keywords: Scenarios, ACDATE, Simulation, Scenario-Driven System Engineering, Completeness and Consistency Analysis, Safety Analysis.

1. Introduction

Future Command and Control (C2) systems need to operate within an integrated grid-based network-centric environment that allows rapid decision development and evaluation to meet the challenges of modern agile warfighting. This paper presents a Scenario-Driven System Engineering (SDSE) approach to develop, evaluate, and test C2 systems. One key component is that once system scenarios are specified, the system can be simulated without any programming and thus saves significant effort and time.

SDSE is compatible with the modern Service-Oriented Architecture (SOA) approach to develop trustworthy systems. DoD is embracing SOA in numerous projects such as the Defense Information Systems Agency GIG Enterprise Services (GES) with its component core services. The SDSE can be used to specify and analyze system behaviors in an SOA.

The core of SDSE is scenario specification and analyses. A scenario is specified using the ACDATE model:

- Actors – An actor is either an external user, system or device, or an internal system, device, component or object;
- Conditions – A condition is a predicate used to trigger an action;
- Data – Attributes of actor, and presenting the semantic of condition, event and action
- Actions – Specified by the trigger event, guard condition, the way to change the status of actors, and sent event(s) to some actors
- Timing – A semantic statement about the relative or absolute value of time or duration
- Events – External/internal significant occurrences that may trigger action(s)

Once system behaviors are specified, various static and dynamic (via simulation) analyses can be performed on the model:

- Completeness and consistency analysis: The model can be used to identify incompleteness at compile time as well as during simulation.
- Performance evaluation: The model can be simulated to determine system performance including throughput and delay.
- Safety analysis: The model can be used to generate the event-tree model and effect-cause diagram commonly used in safety analysis;
- Behavior analysis: The model can be used to generate the state model of the system and various behavior analyses such as reachability analysis, which can be performed on the state model. Formal verification techniques such as temporal logic can be used to analyze the state model.

The SDSE is to be integrated and supported by an automated tool E2E that is currently being used in several experimental projects by US Navy.

2. ACDATE Model – An Example

This section presents an example of ACDATE modeling technique which usually contains two steps:

- Decompose the requirements into ACDATE model elements; and
- Develop system scenarios using the ACDATE model elements.

Taking a battlefield as an example, each warfighting vehicle can be treated as an *Actor*. Each actor (warfighting vehicle) may have its own *Data* such as "available fuel", and its own *Conditions* such as if there is enough fuel to continue moving for 20 miles. The given condition example is constructed on the Data 'available fuel'. An *Event* "not enough fuel" could be fired when there is not enough fuel support subsequent operations. An *Action* would be "to refill the vehicle".

A system scenario would then be specified using the ACDATE model elements: if the event "not enough fuel" occurs, the actor "warfighting vehicle" shall perform action "to refill" within the time specified by the *Timing Attribute* "within 15 minutes".

3. Scenario-Based Rapid Simulation

The key feature of the rapid simulation is automatic simulation code generation once the system scenarios are available. The simulation code has an embedded scheduler, an event queue,

a monitor, and a policy checker to track and verify the alternative system behaviors under different environments, as well as the impact from and to the environment. The simulation is discrete event simulation [1][3].

3.1 Simulation Engine Architecture

The simulation engine has two main parts: system simulator and the environment simulator. The environment simulator simulates the behavior of the environment. It can also simulate the impact of the system to its environment. Separating the system simulation and environment simulation has several significant advantages: It offers the opportunity to observe the behavior of the *system under testing* (SUT) under different loads by varying the environment simulator. Specifically, robustness, reliability and scalability of the system can be determined by generating various inputs to drive the system, e.g., generating an incorrect input can evaluate the system's robustness, and generating the input according to the operational profile will determine the system's reliability, and generating inputs of various sizes to determine the system scalability.

The simulation engine architecture is illustrated by **Figure 1**. The ACDATE model elements form an *entity pool*. The execution of each scenario, which is scheduled by the *scheduler*, will access the entity pool to read or update their internal status. Events may be emitted during the execution of a scenario, which may in turn invoke the execution of other scenarios. An *event queue* is maintained to process all the events emitted by scenarios or the environment. The scheduler will drive the *monitor* or the *policy checker* properly to track all the activities or do runtime policy verification respectively.

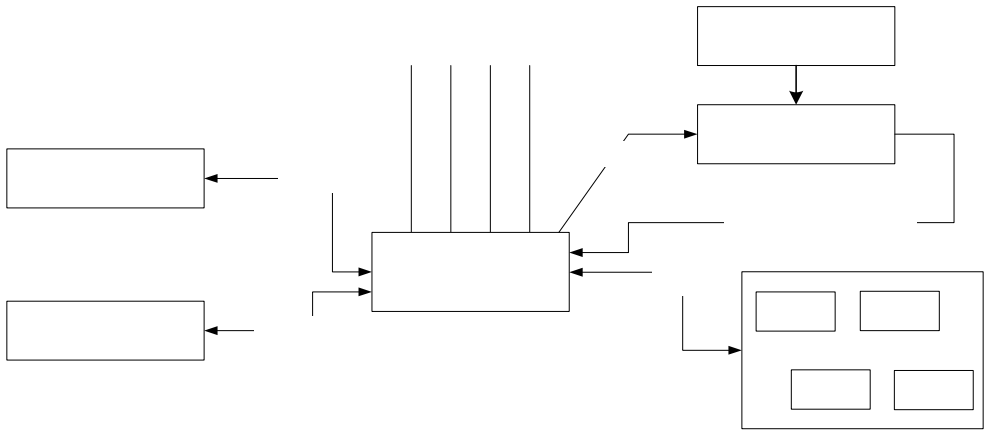

Figure 1: Scenario Based Simulation Architecture

The core of the scheduler is a virtual machine (VM) that enables fine-grained debugging capability. The execution of any scenario can be slow-played and the internal status of each entity can be set-up to any desired value to offer the opportunity to manipulate the simulation and observe rare occurrence or exceptional behaviors.

The monitor will track all the activities and the information will be used to generate the event-tree, the state diagram for each actor or the entire system. For any potential or real malfunction, the monitor or the policy checker will report warnings to indicate either there is a completeness and consistency breach or concurrency or security policy violation. The monitor will also record

the time elapsed that can be used to form the performance evaluation of each actor or the entire system.

The simulation works as follows. An event residing in the event queue, which might be emitted by the environment or the system, is picked up by the scheduler and processed. The event may trigger the execution of a scenario, which will be either concurrent with the execution of other scenarios, or occupy the VM before it finishes, due to the different scheduling policy. The execution course of the scenario may be determined by the internal status of some entities. The scheduler will read or update the corresponding internal status upon the request of the scenario. New events will be emitted on behalf of the scenario during execution, which will be appended to the end of the event queue. The new events might be communication among components inside the system or an outgoing event to the environment. Details of the execution, such as time information, action sequence, and event sequences will be recorded through the monitor, which is available to postmortem analyses. If a policy is registered into the scheduler, corresponding policy checking will be invoked by the policy checker at runtime.

3.2 Simulation Code Generation

The simulation code is generated based on the scenario specification, which includes the ACDATE definition and scenario description. Each ACDATE model element will be translated to an object with the attributes defined in the specification. Instrumentation code will be inserted to the objects to interface with the monitor and policy checker. Each scenario will be translated to a procedure that is basically a sequence of operations on the ACDATE objects or emitting events. Similarly, instrumentation code will be inserted to the procedure to interface with the scheduler, for scheduling the concurrent execution, and event queue for emitting new events. **Table 1** shows a sample simulation code that is automatically generated with instrumentation code that interfaces with the scheduler, event queue, monitor, or policy checker.

```
scenario_5 = function(co_routine_name, platform)  // a scenario
 coroutine.yield();                               // interface to scheduler
 ...
 event_4:AddDestination(1);                       // interface to monitor
 event_4:emit();                                  // interface to event queue
 ...
 data_3.value = 1000;                             // data_3:set() will be invoked and
                                                  // interface to monitor embedded there
 action_10:before_do(co_routine_name, platform);  // interface to policy checker embedded here
 action_10_dummy_func(co_routine_name, platform);
 action_10:after_do(co_routine_name, platform);
 timer[platform] = timer[platform] + unit;        // advance and record system time
 ...
```

Table 1 Sample Simulation Code

4. Simulation for Dynamic Analyses

Various dynamic analyses are enabled by simulation, e.g., completeness and consistency (C&C) analysis, performance analysis, safety analysis, and behavior analysis.

The dynamic C&C analysis complement static C&C checking [6] because some incompleteness or inconsistency can only be observed during runtime when concurrency comes

into play. The simulation will record all the potential inconsistency and incompleteness such as the simulation encounters a situation where there is no related instruction in the specification; or an action did not change the state of any actors, which may imply that the system is at an abnormal state that responses to no input. In a recent experiment with a C2 system that has around 1000 entities and 120 scenarios, it takes 3 minutes to generate and execute the simulation and it detected around 200 bugs related to incompleteness or inconsistency.

During the simulation, system time will be recorded for each action when it starts or ends, event when it is emitted or handled, and data when the value changes. The recorded time information can be used for performance analysis such as the throughput and average delay of the system.

Event sequence can be useful for safety analysis. **Figure 2** shows a sample event sequence tree which is automatically generated by simulation. The effect-cause diagram [7] can be also automatically generated by examining all the event trees related to a system to link from an effect to its possible causes. Effect-cause diagram is similar to fault tree [5] except that the elements on the diagram are analytic entries. By combining event sequence tree and effect-cause diagram, one can pinpoint which system components that failed during failure analysis.

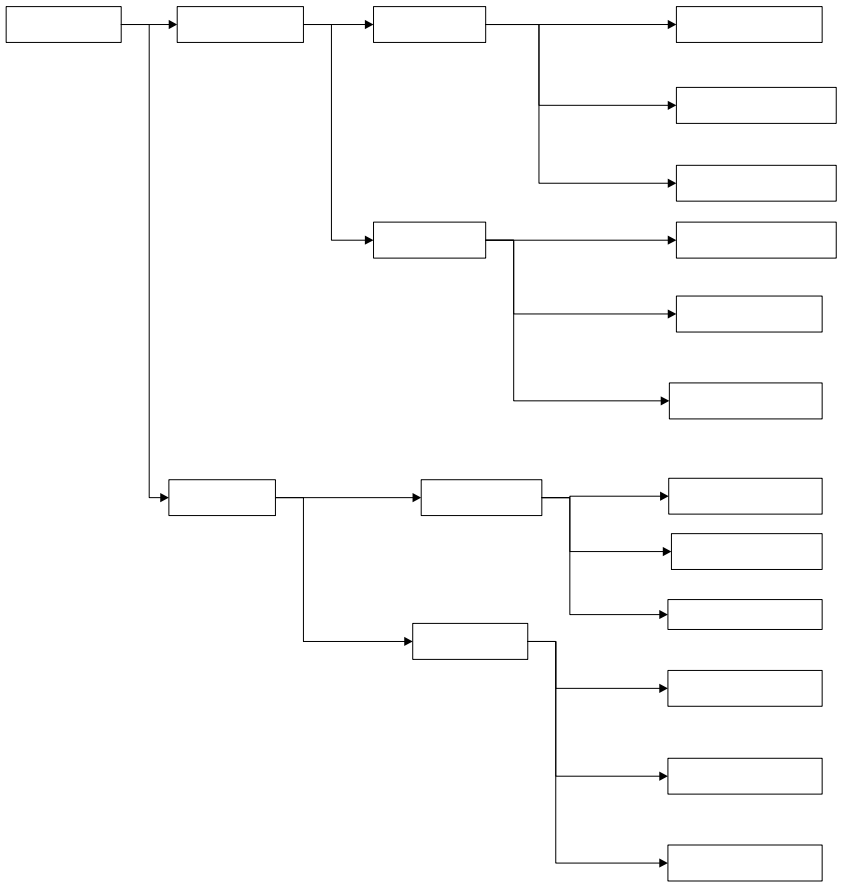

Figure 2 Sample Event Sequence Tree

Behavior analysis, such as reachability [8] analysis, is usually performed on state model. There is a natural mapping from the state model generated from the simulation (SMM) [7] to UML's Statechart [2], which can then be fed into various UML tools for further analyses. It is also possible to perform Linear Temporal Logic (LTL) analysis and model checking using SPIN [7] on the state model to detect deadlock or other malfunctions.

5. Conclusion

This paper proposes a systematic process to perform variety kinds of dynamic analyses based on scenario specification. Once system scenarios are specified, the simulation code can be automatically generated, and the system can be simulated without any additional programming. The simulation can be used to perform various dynamic analyses including C&C checking, safety analysis, and performance analysis. The SDSE is being integrated into an automated tool E2E.

References

[1] Jerry Banks, *Discrete-Event System Simulation*, Prentice Hall, 2001.

[2] B.P. Douglass, *Doing Hard Time: Develop Real-Time Systems with UML Objects*, Frameworks, and Patterns, Addison-Wesley, 1999.

[3] D. Luckham, *The Power of Events: An Introduction to Complex Event Processing in Distributed* Enterprise *Systems*, Addison-Wesley, 2002.

[4] SPIN: at http://spinroot.com/spin/whatispin.html.

[5] N. Storey, *Safety-Critical Computer Systems*, Addison Wesley, Reading, MA, 1996.

[6] W. T. Tsai, R. Paul, L. Yu, X. Wei, and F. Zhu, "Rapid Pattern-Oriented Scenario-Based Testing for Embedded Systems" to appear in *Software Evolution with UML and XML*, edited by H. Yang, 2004.

[7] W. T. Tsai, L. Yu, R. Paul, C. Fan, and X. Liu, "Rapid Scenario-Based Simulation and Model Checking for Embedded Systems", in Proc. of SEA, 2003, pp. 568-573.

[8] W. T. Tsai, L. Yu, A. Saimi, and R. Paul, "Scenario-based Object-Oriented Test Frameworks for Testing Distributed Systems", in Proc. of IEEE Future Trends of Distributed Computing Systems, 2003, pp.288-294.