

Applying Architecture Modeling Methodology to the Naval Gunship Software Safety Domain

Joey Rivera

US Naval Postgraduate School
1 University Circle
Monterey, California, USA
jrivera@nps.edu

Abstract. In this paper, we describe the results of using the Software Architecture modeling methodology in the Naval Gunship Software Safety domain. The paper describes the application of the Monterey Phoenix software architecture formal specification framework to the problem of testing gunship software interaction with environment that may result in software states that are considered unsafe.

Keywords: Software System Architecture, Modeling, Environmental Modeling, Assertion Checking

1. Problem

Naval Gunship software system architecture requires consideration for loss of life [6]. The US Navy's Software System Safety Technical Review Panel (SSSTRP) evaluates potential software systems for US Navy Gunships during the initial acquisition process. The SSSTRP organization has identified an unacceptable rate of software acquisition evaluation failures due to issues related to software safety [7]. The high failure rate results in the vendor being required to submit additional documentation and an acquisition process (and timeline) that is unacceptable to the US Navy Acquisition community.

2. Related Work

Conventional notations, such as UML, have long been the preferred tool of software architects, but the software architect needs a number of different views of the software architecture for the various uses and users [3] [4]. Architecture modeling provides a high-level abstraction for representing structure, behavior, and key properties of a software system [8] [9]. These abstractions are useful in describing complex problems to various stakeholders in an understandable manner. The use of the Monterey Phoenix (MP) methodology [1] enables software architects to model system's environments in such a way that enables the non-technical stakeholder to understand the potential effects when making changes to their environment.

3. Proposed Solution: Naval Gun System Domain Modeling Requirements

MP is a formal architectural modeling methodology that can be used to test software engineering architecture [1]. MP architecture describes events, event structure, and relationship between events. MP Environmental models provide high-level abstractions for representing the structure, behavior, and key properties of software systems relative to how the software system interacts with its environment.

The goal of MP design is to create a model with a set of architectural properties. The relative importance of the various event properties depends on the nature of the intended system. Attributes are properties of an event that can be used to further define domain model representations. Attributes are particularly valuable as they represent a more detailed (and measurable) application state.

3.1. Assertion Checking

MP constructs support evaluation of all possible scenarios within a limited scope and have the ability to find counter examples if the assertion is violated.

The below event grammar represents two systems that send/receive radar data. The GCC_activity sends Display Data to the Radar (ADS) Computer System. The ADS_activity event contains a sequence of zero or more ADS_receiveRequest_DisplayData_from_GCC events:

```
ROOT GCC_activity: {OperatorActions
  (* GCC_sendRequest_DisplayData_to_ADS
  ( GCC_timeoutWaiting_DisplayData_from_ADS_one
  | GCC_timeoutWaiting_DisplayData_from_ADS_two
  | GCC_receive_DisplayData_from_ADS *)});
ROOT ADS_activity: (*ADS_receiveRequest_DisplayData_from_GCC
  ADS_send_DisplayData_to_GCC *);
ROOT Connector_GCC_to_ADS_activity: (*
  GCC_sendRequest_DisplayData_to_ADS(
  ADS_receiveRequest_DisplayData_from_GCC
  | GCC_timeoutWaiting_DisplayData_from_ADS_one *) );
ROOT Connector_ADS_to_GCC_activity:
  (* ADS_send_DisplayData_to_GCC
  (GCC_receive_DisplayData_from_ADS
  | GCC_timeoutWaiting_DisplayData_from_ADS_two)*);
Connector_GCC_to_ADS_activity, GCC_activity
SHARE ALL GCC_sendRequest_DisplayData_to_ADS,
  GCC_timeoutWaiting_DisplayData_from_ADS_one;
Connector_GCC_to_ADS_activity, ADS_activity
SHARE ALL ADS_receiveRequest_DisplayData_from_GCC;
Connector_ADS_to_GCC_activity, GCC_activity
SHARE ALL GCC_receive_DisplayData_from_ADS,
```

GCC_timeoutWaiting_DisplayData_from_ADS_two;
 Connector_ADS_to_GCC_activity, ADS_activity
 SHARE ALL ADS_send_DisplayData_to_GCC;

Notation: {} = Unordered Events (no precedes relationship); {*R*} = Given an expansion scope of n (finite), event R has (n+1) scenarios.



Figure 1: Gunship_Sharing_Data MP Interface

Figure 2 represents an Assertion Check that evaluates a system state where the GCC system sends data and encounters a Timeout. In this scenario, the event "GCC_timeoutWaiting_DisplayData_from_ADS_one(4)" is considered an unsafe state.

Event grammar rules specify a set of possible event traces representing different scenarios for the GCC_activity. Note the "|" "Or" event grammar allows for the model to test for scenarios where the Send event ended in a timeout. This capability was particularly helpful when testing the software integration impact of a system that was highly dependent upon network communications.

3.2. Formal Verification of Model Properties

Due to the seriousness of the potential risks, the SSSTRP requires exhaustive testing before modifications are approved. Jackson's "Small Scope Hypothesis" [2] [5] argues that a high proportion of bugs can be found by testing the system within some

small scope. The SSSTRP evaluation process requires an ability to formally evaluate all possible scenarios within a finite scope. MP is able to generate all possible scenarios within scope while simultaneously evaluating model properties, or MP Assertions. In the below example we considered an unsafe state to be any scenario where maximum power output is above 100. Our research showed that there are two scenarios where the maximum power output is above 100:

MP Program

```

Plane Code
ROOT A_activity: (* <|> A_getDataFrom_B *);
A_getDataFrom_B: (
  A_prepareRequestDataFrom_B
  A_requestDataFrom_B
  A_waitDataFrom_B
  | A_failReceivingDataFrom_B
  | A_receiveDataFrom_B);
A_prepareRequestDataFrom_B: (duration=10, power=10);
A_requestDataFrom_B: (duration=1, power=1);
A_waitDataFrom_B: (power=1);
A_failReceivingDataFrom_B: (power=1);
A_receiveDataFrom_B: (duration=1, power=1);
ROOT B_activity: (* B_answerRequestDataFrom_A *);
B_activity: A_activity SHARE All A_requestDataFrom_B,
A_failReceivingDataFrom_B, A_receiveDataFrom_B;
B_answerRequestDataFrom_A: (
  A_requestDataFrom_B |
  | B_failProcessingRequestDataFrom_A
  A_failReceivingDataFrom_B);
| B_processRequestDataFrom_A
B_requestDataFrom_C
B_waitDataFrom_C | B_failReceivingDataFrom_C
A_failReceivingDataFrom_B
| | B_receiveDataFrom_C B_sendDataTo_A
A_receiveDataFrom_B );););

```

Simulation Options

General Options

Option	Value
Default Expansion Scope:	1
Display x scenarios:	100
Display starting scenario no.:	0
Use debug:	No
Scenario Expansion debug level:	-
Scenario Merging debug level:	-

Scenario Filter Conditions

Event Count			Maximum sum for parallel events			Maximum sum for consecutive events		
Count(event)	Condition	Count	Sum(attribute)	Condition	Value	Sum(attribute)	Condition	Value
-	-	-	power	At least	100	-	-	-
-	-	-	-	-	-	-	-	-
-	-	-	-	-	-	-	-	-
-	-	-	-	-	-	-	-	-
-	-	-	-	-	-	-	-	-
-	-	-	-	-	-	-	-	-

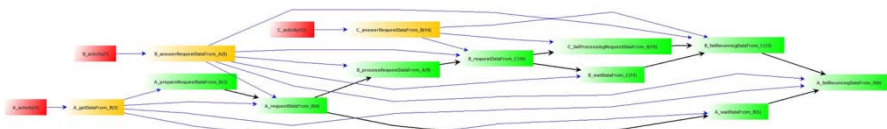


Figure 2: Assertion Checking: Maximum Sum of Power

4. Expected Contributions

Environment Behavior – Evaluating a gunship system requires the ability to model the system and its environment. Formally testing a system's functional profile while introducing external behavioral events is key to understanding how the system will react in a combat situation.

Performance Estimates – Early in the design process, non-functional requirements need to be formally tested in order to see how the integration of a system will impact the overall System of Systems (SoS). Creating a single SoS MP model prior to the acquisition process enables the acquisition community to standardize the vendor response questionnaire, thereby resulting in a formal method of evaluating potential software solutions. Furthermore, incorporating a technical questionnaire that elicits MP model input responses from potential vendors is expected to streamline the vendor's response efforts as well as the evaluation methodology.

Reuse - Naval gunship system models will be reusable in order to streamline future software acquisition and testing processes. Abstract versions of gunship systems can be reused as the specifics of the system components are not germane to the application of MP.

Verification of Model Properties – The SSSTRP evaluation processes require a formal verification of a finite number of possible scenarios that may result in an unsafe software safety state. Since MP is capable of evaluating all scenarios within scope, the SSSTRP has the capability to test for unsafe system states after introducing a new environmental actor or event.

Model Views – MP has the ability to produce abstract model views of scenarios. The SSSTRP evaluation process requires a methodology that can extract representations of the model in visual and textual views in order to communicate the results to both technical and non-technical SSSTRP members.

5. Current Status

The initial MP framework was applied to the Navy Gunship Software Safety domain and tested using an MP compiler developed by this research team (www.riverainc.com/mp). The tool has created an executable code that bridged the gap between concept and design and proved to be critical during the testing process.

The initial implementation of the framework satisfied our objective of creating a framework that enables an ability to formally evaluate the integration risks associated with potential gunship software solutions. Furthermore, the most critical need for this research was to create an ability to identify unsafe states when testing environmental scenarios (Assertion Checking). We have modeled the entire Gunship using MP and are able to test for unsafe states using our MP compiler and graphical interface.

4. Plan for Evaluation

Evaluating the effectiveness of MP as applied to the Gunship Software Safety domain resulted in the evaluation of the following domain-specific solution properties:

Dynamic Reuse and Extensibility - Reuse is supported through the event grammar capability to add/edit/delete MP abstract components (Schemas). Since the interaction among systems and events is decoupled via connectors, a high degree of dynamism is provided with minimal disturbance to the rest of the model.

Efficiency - For the purposes of this research we used a custom event trace generator created specifically for MP to test our environmental models. Using the MP compiler enabled us to test proposed models in relatively short test cycles, thereby resulting in fewer errors in MP programming, standardized MP modeling methodology, and less time needed to change the MP test model to better reflect the Gun System and its environment.

Traceability – The MP framework supports a simple, faithful mapping between the architectural elements and their implementation. MP has the capability to map Use Cases to functional design requirements. This fidelity is necessary to ensure that the

conceptual integrity and key properties of an application's architecture are preserved by the architecture's implementation.

Extensibility – The MP framework supports design and implementation of new events. This allows one to customize the framework and create environmental scenarios that may or may not be supported by the existing system. Extensibility is a critical feature for MP as gunship systems are often deployed in extreme environments in which operational scenario testing is not economically or operationally feasible.

Assertion Checking - MP does an exhaustive search through all possible scenarios (up to the scope limit). MP provides a means to write assertions about the system behavior and tools to verify those assertions. This function is non-trivial since having the ability to write event grammar in a framework that outputs formal assertions allows the architect to test for unsafe states by evaluating assertion values.

References

1. M. Auguston, "Software Architecture Built from Behavior Models", ACM SIGSOFT Software Engineering Notes, September 2009 Volume 34 Number 5
2. D. Jackson and C. A. Damon. Elements of style: Analyzing a software design feature with a counterexample detector. IEEE Transactions on Software Engineering, 22(7), July 1996.
3. Philippe Kruchten: Architectural Blueprints - the 4+1 View Model of Software Architecture. In: IEEE Software. 12 (6) November 1995, pp. 42-5
4. Dewayne E. Perry and Alexander L. Wolf. "Foundations for the Study of Software Architecture", ACM SIGSOFT Software Engineering Notes, 17:4 (1992), pp. 40-52.
5. Jackson, Daniel. 2006. Software Abstractions: Logic, Language, and Analysis. Cambridge, Massachusetts: The MIT Press.
6. M.Auguston, B.Michael, M.Shing, Environment Behavior Models for Automation of Testing and Assessment of System Safety, Information and Software Technology, Elsevier, Vol. 48, Issue 10 , October 2006, pp. 971-980
7. Rivera, Joey, Luqi, and Valdis Berzins, Effective programmatic software safety strategy for US Navy Gun System acquisition programs. Proceedings of NPS 6th Annual Acquisition Research Symposium. 2009. 159-164, <http://edocs.nps.edu/npspubs/scholarly/TR/2010/NPS-GSBPP-10-003.pdf>
8. D.E. Perry, A.L. Wolf. Foundations for the Study of Software Architectures. ACM SIGSOFT Software Engineering Notes, vol. 17, no. 4, pp. 40-52, October 1992.
9. M. Shaw, D. Garlan. Software Architecture: Perspectives on an Emerging Discipline. Prentice-Hall, 1996.

Acknowledgement: The views and conclusions contained herein are those of the author and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the U.S. Government.