Rethinking the Formal Specification, Validation, and Verification Process: Making it an End-to-End Process that is Scalable

Bret Michael
Professor of Computer Science and Electrical Engineering
Naval Postgraduate School
National Capital Region Campus, Arlington, Virginia
bmichael@nps.edu
Disclaimer

The views and conclusions in this lecture are those of the presenter and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the U.S. Government.
The Perilous Journey of Software Development

Software errors introduced

- Client ideas
- Reqs Spec
- Design
- Code

< 85%

< 25%

June 20, 2013

SERE 2013
Validation and Verification (V&V)

Validation

Verification

Software errors introduced

June 20, 2013

SERE 2013
Inadequacy of Manual V&V Techniques

- Relies on
  - Manual examination of software requirements and design artifacts
  - Manual and tool-based analysis of design and code
- Ineffective for validating the correctness of the developer’s cognitive understanding of the requirements
  - See Example 1
- Inadequate for locating the subtle errors in the complex time-constrained software
  - See Example 2
Software Automation

Holds the key to the validation and verification of the behaviors of complex software-intensive systems

Relies on formal specification of system behaviors

- Specifications need to match the true intent of the customer’s requirements
Factors Contributing to Specification Errors

- Incorrect translation of the natural language to formal assertion
- Incorrect translation of the requirements, as understood by the modeler, to natural languages
- Incorrect cognitive understanding of the requirements
Iterative Process

The process for writing specification is **iterative** because of human nature

- Humans usually write natural language requirements with a **specific scenario in mind**
- They encounter **ambiguities** in the natural language requirement
  - Writing **formal specifications** can remove ambiguities
- Validate and generalize the requirements with a **plurality of scenarios**
Ideal Attributes for V&V Process

Early start, continuous, and proactive
Ideal Attributes for V&V Process

- Early start, continuous, and proactive
Formal V&V Techniques

There is no one-size-fits-all formal V&V techniques

- Need to select the right tool for the right job in the different phases of software development

Need a framework to understand the effectiveness of different formal methods in different phases of the software development process
The Role of Specification –

Have we built the right product?

E.g.,

“if pump pressure is turned Low then High and then Low again all within 10 milliseconds then pump should not be High for at least 20 additional milliseconds”
The Role of Verification –

Have we built the product right?

“if pump pressure is turned Low then High and then Low again all within 10 milliseconds then pump should not be High for at least 20 additional milliseconds”

Spec. = Formal representation

```
class PumpCtl {
    int x;
    void pumpOn() {
        ...
    }
}
```

Verification = The bridge between specification and implementation
The Role of Validation -

Customer cognitive requirements

E.g.,

"if pump pressure is turned Low then High and then Low again all within 10 milliseconds then pump should not be High for at least 20 additional milliseconds"

Are we really building the right system?

Spec. = Formal representation

June 20, 2013

SERE 2013
A Formal V&V Tradeoff Cuboid
The Specification/Validation Dimension

Represents the cost/effort and effectiveness/expressiveness associated with the specification language of a given formal method.

Deals with the ease and ability of writing formal specifications and getting them right.

That is, getting them to represent the cognitive intent the human owner has for this requirement.
The Implementation Dimension

Deals with the ease of adapting a given real-life complex program to a specific FV&V technique.

The Verification Dimension

Represents the cost/effort, and effectiveness/coverage of verification.
The Coverage Cube

- Validation related: How well are requirements covered?
- How well does formal specification match the actual code?
- To what extent can the formal specification be verified?

More is better.
The Cost Cube

Cost of writing specifications: how easy is it to write them and to get them right?

Cost of modeling: how easy is it to adapt the program in order for verification to take place

Less is better.
Specification Coverage

Measures the ability to express different classes of system behaviors

- **Logical behavior**
  - Describes the cause and effect of a computation, typically represented as functional requirements of a system
  - See Example 3

- **Sequencing behavior**
  - Describes the behaviors that consist of sequences of events, conditions and constraints on data values, and timing
  - See Example 4
Beyond Pure Sequencing

- **Timing constraints** - Describe the timely start and/or termination of successful computations at a specific point of time
  
  See Example 5

- **Time-series constraints** - Describe the timely execution of a sequence of data values within a specific duration of time
  
  See Example 6
Positive and Negative Behaviors

Positive behaviors – what you want the system to do
- For example, “Whenever stop command is received, the vehicle should reach complete stop within 30 seconds”

Negative behaviors – What you do not want the system to do
- For example, “Pump should never operate until at least two seconds after valve-shut”
Specification Coverage (cont’d)

Positive and Negative Behaviors (cont’d)

The key about negative behavior is not the way it is phrased, it is about a behavior that the system has to avoid.

- Any behavior can be phrased as a positive or negative statement.

**Negative statement:**

*Pump should never operate until at least two seconds after valve-shut*  
(a safety requirement)

**Positive statement:**

*Pump should remain inactive until at least two seconds after valve-shut*  
(a design decision)
Implementation Coverage

Measures the extent a target system can be verified by a formal method

- For example, the special programming languages tailored specifically for the Theorem Proving process does not cover all aspects of the original C, C++ program, and hence has a low implementation coverage.
Verification Coverage

- Measures the thoroughness of verification by a formal method

- For example, whenever a theorem proving process does complete, it provides 100% coverage and hence has a high verification coverage
Specification Cost

Measures the amount of effort required to

- Express informal human expectations as formal specifications
- Validate the resultant formal specifications for correctness
- Maintaining the resultant formal specifications as the system evolves
Specification Cost (cont’d)

 Assertion-oriented versus model-oriented specifications

- Assertion-oriented specification
  - High-level requirements are decomposed into more precise requirements that are mapped one-to-one to formal assertions

- Model-oriented specifications
  - A single monolithic formal model (either as a state- or an algebraic-based system) captures the combined expected behavior described by the lower level specifications of behavior
  - Describes the expected behavior of a conceptualized system from the analyst’s understanding of the problem space
Advantages of Using an Assertion-Oriented Specification Approach

- Requirements are traceable because they are represented, one-to-one, by assertions (acting as watchdogs for the requirements)
  - A monolithic model is the sum of all concerns: on detecting a violation of the formal specification, it is difficult to map that violation to a specific human-driven requirement

- Assertion-oriented specifications have a lower maintenance cost than the model-oriented counterpart when requirements change (i.e., ability to adjust the model)
Continuation of Advantages

Assertions can be constructed to represent illegal behaviors, whereas the monolithic model typically only represents “good behavior.”

It is much easier to trace the expected and actual behaviors of the target system to the required behaviors in the requirements space.

- Formal assertions can be used directly as input to the verifiers in the verification dimension
Continuation of Advantages

Conjunction of all the assertions becomes a “single” formal model of a conceptualized system from the requirement space.

- Can be used to check for inconsistencies and other gaps in the specifications with the help of computer-aided tools.
Implementation Cost

Measures the amount of effort required to instrument the target code for verification

- For example, we must create an abstract model from a C++ program before it can be model-checked, and hence has a high implementation cost
Verification Cost

Measures the amount of effort required to carry out the verification

- For example, model-checking is an automatic, “push-button” process and has a very low verification cost
Application of the FV&V Tradeoff Cuboid

We shall illustrate the use of the tradeoff space with a qualitative comparison of three common categories of FV&V techniques:

- Theorem Proving
- Classical Model Checking
- Execution-based Model Checking
Application of the FV&V Tradeoff

Cuboid (cont’d)

Theorem Proving

- Examples: ACL2/PL; STeP/PLTL; PVS/HOL
- Existing theorem provers have rather weak specification languages
  - The more automated the theorem prover, the more restrictive is its specification language
- The behavior expressed in the specification not easily visualized
- Need to create a model to express behavior of a given program using the specification language
  - The new model will not cover all aspects of the original program
- Need human driver to guide the verification process
Application of the FV&V Tradeoff Cuboid (cont’d)

Theorem Proving (cont’d)
Application of the FV&WV Tradeoff - Cuboid (cont’d)

Model Checking

- Examples: SPIN/PLTL or Büchi-automata; UPPAAL/CTL
- Similar to TP in terms of the expressive power of their specification languages
- The behavior expressed in the specification not easily visualized
- Need to create abstract model from large programs to avoid state-space explosion
- 100% automatic model checking process
Application of the FV&V Tradeoff Cuboid (cont’d)

Model Checking (cont’d)
Application of the FV&V Tradeoff: Cuboid (cont’d)

Execution-based Model Checking
- Combination of Runtime Verification (RV) and Automatic Test Generation (ATG)
- Examples: StateRover, Java Path Finder (JPF)
- StateRover
  - Specification language is Turing equivalent
  - The UML-like statechart assertions are easier to create and understand than the text-based specifications
  - Need to insert “probes” in target code
  - Coverage depends on the ATG, usually not be 100%
- Java Path Finder
  - Instrument Java code with assertions
  - Use symbolic execution, could be 100% if enough space
Application of the FV&V Tradeoff Cuboid (cont’d)

Execution-based Model Checking (cont’d)
Computer-Aided V&V Process

Given scalability, cost, and coverage considerations, we advocate the use of a computer-aided V&V process that uses:

- Statechart assertions
- Runtime execution monitoring
- Scenario-based testing

We have explored this in terms of independent V&V (IV&V)
Our IV&V Framework

- Incorporates advanced computer-aided validation techniques to the IV&V of software systems

- Allows the IV&V team to capture both
  - Its own understanding of the problem
  - The expected behavior of any proposed system for solving the problem via an executable system reference model
Some Definitions of Terms

Developer-generated requirements
- The requirements artifacts produced by the developer of a system

System Reference Model (SRM)
- The artifacts developed by the IV&V team’s own requirements effort
Contents of the SRM

- Use cases and UML artifacts
- Formal assertions to describe precisely the necessary behaviors to satisfy system goals (i.e., to solve the problem) with respect to:
  - What the system should do
  - What the should not do
  - How the system should respond under non-nominal circumstances
Starting Point

Development of formal, executable representations of a system’s properties, expressed as a set of desired system behaviors.
Use Cases and UML Artifacts of the SRM

Stakeholder's Input
(mission statements, operation concepts documents, user expectations, etc.)

Use Case Scenarios

Dynamic UML Models
(Message Sequence Charts, Activity Diagrams etc.)

Static UML Models
(Object Class Diagrams)
Use of Assertions

- One statechart assertion for each behavior of interest
- Can have nondeterminism in statechart assertions because we must address existential conditions (use of existential quantifier) instead of just the universal quantifier
Use of Statechart Assertions

- Start with high-level requirement
  - **R1.** The track processing system can only handle a workload not exceeding 80% of its maximum load capacity at runtime.

- Reify R1 into lower level requirement
  - **R1.1** Whenever the track count (cnt) Average Arrival Rate (ART) exceeds 80% of the MAX_COUNT_PER_MIN, cnt ART must be reduced back to 50% of the MAX_COUNT_PER_MIN within 2 minutes and cnt ART must remain below 60% of the MAX_COUNT_PER_MIN for at least 10 minutes.
On-Entry/timer120.restart(); cnt = 0;

On-Entry/timer600.restart(); cnt = 0;

onTime = primary.getTime();
cnt = 0;

state: final mt: MAX_COUNT_PER_MIN = 10;
long nTime = 0;
double cnt = 0;
double t;
TRTimeoutSimulatedTime timer120 = new TRTimeoutSimulatedTime(120, this);
TRTimeoutSimulatedTime timer600 = new TRTimeoutSimulatedTime(600, this);
Validation of Assertions

- Formal assertions must be executable to allow the modelers to visualize the true meaning of the assertions via scenario simulations.

- One way to do this is to use an iterative process that allows the modeler to:
  - Write formal specifications using Statechart assertions.
  - Validate the correctness of the assertions via simulated test scenarios within the JUnit test-framework.
Use of Scenarios

1. Identify typical scenarios
2. Capture Natural Language Requirements
3. Create scenario-based test cases
4. Write requirements as statechart assertions
5. Run test cases against the generated code
6. Create more test cases from alternative scenarios
7. Is there other alternative scenarios?
8. Assertion’s behavior satisfies the conceptual requirement as understood by the modeler?
9. Use test cases and assertions for test automation and runtime monitoring

Possible errors:
1. Incorrect translation of requirement to natural language
2. Incorrect translation of natural language specs to assertions
3. Incorrect translation of natural language specs to assertions
4. No cognitive understanding of requirements
End-to-end Validation Process

- Start by testing individual assertions using the scenario-based test cases to validate the correctness of the logical and temporal meaning of the assertions.

- Next test the assertions using the scenario-based test cases subjected to the constraints imposed by the objects in the SRM conceptual model.

- Then use an automated tool to exercise all assertions together to detect any conflicts in the formal specification.
Pictorial View of Validation

Stakeholder's Input
(mission statements, operation concepts documents, user expectations, etc.)

Use Case Scenarios

Dynamic UML Models
(Message Sequence Charts, Activity Diagrams etc.)

Static UML Models
(Object Class Diagrams)

Executable Assertions

JUnit Test Framework

(1) Tests driven by use case scenarios without the application context

(2) Tests driven by use case scenarios with the application context

(3) Tests driven by white-box tester for detecting assertion (and requirement) conflicts

White-box Automatic Tester
Runtime Verification

- Uses executable SRMs
- Monitors the runtime execution of a system and checks the observed runtime behavior against the system’s formal specification
  - It serves as an automated observer of the program’s behavior and compares it with the expected behavior per the formal specification
- Requires that the software artifacts produced by the developer be instrumented
Execution-based Model Checking

- Can be used if state-based design models are available
- A combination of RV and Automatic Test Generation (ATG)
  - Large volumes of automatically generated tests are used to exercise the program or system under test, using RV on the other end to check the SUT’s conformance to the formal specification
Pictorial View of EMC

isSuccess()

JUnit test suite

Auto-generated (white box) test cases

Statechart model with embedded statechart assertions

StateRover Statechart Model

Primary Thread

Primary Statechart

Assertion Thread

Assertion
Some Ways to Use Auto-generated Tests

- To search for severe programming errors, of the kind that induces a JUnit error status, such as NullPointerException
- To identify test cases which violate temporal assertions
- To identify input sequences that lead the statechart under test to particular states of interest
An Example

StateRover generated WBTestCase creates sequences of events and conditions for the state chart under test

- Only sequences consisting of events that the SUT or some assertion is sensitive to, by repeatedly observing all events that potentially affect the SUT when it is in a given configuration state, selects one of those events and fires the SUT using this event
Hybrid Model- and Specification-based WBATG

StateRover’s WBTTestCase automatically generates:
- Events
- Time-advance increments, for the correct generation of timeoutFire events
- External data objects of the type that the statechart prototype refers to

WBATG observes all entities, namely, the SUT and all embedded assertions:
- It collects all possible events from all of those entities
Verification of Target Code

If only executable code is available, the IV&V team can use the StateRover white-box tester in tandem with the executable assertions of the SRM to automate the testing of the target code produced by the developer:

- Executable assertions of the SRM
  - Keep track of the set of possible next events to drive the SUT
  - Serve as the observer for the RV during the test
Automated Testing Using the SRM

SUT - model
(instance of class model)

1. Observe events, data, time delays
3. Dispatch input event and data
2. incrTime()
4. Output events
5. isSuccess()

ExternalAssertionChecker

Assertions

WBATG

Timer

Implement time delays

SUT - model (instance of class model)
Questions?
Backup Slides
Example 1

What does it mean by

“generate a report once every 30 days until the project is complete”? 

What happen if we complete the project in 15 days? Do we need to submit a generate?
Example 2

Sequencing behaviors like

“If pump pressure is turned Low then High and then Low again all within 10 milliseconds then pump should not be High for at least 20 additional milliseconds”

are only observable at runtime and at such a time scale that make human intervention at runtime impractical.
Example 3

Logical behavior:

Given two positive numbers $x$ and $e$, the square root function $sqrt(x)$ must satisfy the requirement:

$$|x - sqrt(x) \times sqrt(x)| < e.$$
Example 4

Sequencing Behavior:

Once engine is turned off, compartment lights must be on until driver door is opened.
Example 5

Timing constraint:

The `sqrt()` function must complete its computation and return an answer within 200 milliseconds from the time it is called.
Example 6

Time-series constraints:

Whenever the system load (L) exceeds 75% of the MaxLoad, L must be reduced back to 50% of the MaxLoad within 1 minute and must remain at or below 50% of the MaxLoad for at least 10 minutes.
Example of Conducting Assertion-oriented Specification

Start with high-level requirement

- **R1.** The system shall not exceed 75% of its maximum load capacity at runtime.

Reify R1 into lower level requirement

- **R1.1** Whenever the system load (L) exceeds 75% of the MaxLoad, L must be reduced back to 50% of the MaxLoad within 1 minute and must remain at or below 60% of the MaxLoad for at least 10 minutes.
Continuation of Example

Map R1.1 to a formal assertion expressed as a Statechart assertion

```java
// Statechart

// NormalWorkLoad
void completeTask(L > 0)
L--;

L > highLoad
L++;

newTask(L++)

timeoutFire(L > 0)

// WaitOneMinute
L <= normalLoad

// StartOneMinuteTimer
onEntry/onEntry

newTask(L++)

onEntry/onEntry

timeoutFire(L > 0)

L <= sustainingLoad

// Error
onEntry/sSuccess = false;
```

```java
// Class

public class ExampleClass {
    static final int MaxLoad = 20;
    double highLoad = 0.75 * MaxLoad;
    double normalLoad = 0.5 * MaxLoad;
    double sustainingLoad = 0.6 * MaxLoad;
    int L = 0;

    TRTimeoutSimulatedTime oneMinuteTimer = new TRTimeoutSimulatedTime(60, this);
    TRTimeoutSimulatedTime tenMinuteTimer = new TRTimeoutSimulatedTime(600, this);
}
```
static final int MaxLoad = 20;
double highLoad = 0.75 * MaxLoad;
double normalLoad = 0.5 * MaxLoad;
double sustainingLoad = 0.6 * MaxLoad;
int L = 0;
TRTimeoutSimulatedTime oneMinuteTimer = new TRTimeoutSimulatedTime(60, this);
TRTimeoutSimulatedTime tenMinuteTimer = new TRTimeoutSimulatedTime(600, this);

L > highLoad

L <= normalLoad

L <= sustainingLoad