Building Reliable Software with Applied Formal Methods

A Brief Overview

Daniel M. Zimmerman
Institute of Technology
University of Washington Tacoma

Outline

- *Applied* Formal Methods
- Correctness and the Java Modeling Language
- Unit Testing with JMLUnit
- Current Work

Applied Formal Methods

- **Formal methods** are mathematical techniques for building verifiably-correct software systems.
- **Applied formal methods** is the creation and evaluation of techniques *and tools* that make formal methods accessible *and useful* to developers who may not know all the mathematics involved.

Correctness

- A **correct** software system is one that does what it's supposed to.

Correctness

- A **correct** software system is one that does what it's supposed to.
- Correctness is always **relative**!

Correctness

- A **correct** software system is one that does what it's supposed to.
- Correctness is always **relative**!
- You need a **specification** of what a system is supposed to do before you can evaluate its correctness.

Specifications

- Specifications of software range in formality:
  - informal - English documentation (e.g., "normal" comments in code)
  - semi-formal - structured English documentation (e.g., **Javadoc**)
  - formal - annotations and assertions (e.g., `assert` statements and **contracts**)

Informal Specs

```
/* Deduct some cash from this account and
   return how much money is left. */

public int debit(int amount)
```

Informal Specs

```
/* Deduct some cash from this account and
   return how much money is left. */

public int debit(int amount)
```

- What happens when:

Informal Specs

```
/* Deduct some cash from this account and
   return how much money is left. */

public int debit(int amount)
```

- What happens when:
  - amount is negative?

Informal Specs

```
/* Deduct some cash from this account and
   return how much money is left. */

public int debit(int amount)
```

- What happens when:
  - amount is negative?
  - amount is bigger than the account balance?

Informal Specs

```
/* Deduct some cash from this account and
   return how much money is left. */

public int debit(int amount)
```

- What happens when:
  - amount is negative?
  - amount is bigger than the account balance?
- Is the balance changed when the call fails?

Semi-Formal Specs

```
/** Debit this account.
  * @param amount the amount to debit.
  *         <code>amount</code> must be
  *         non-negative.
  * @result the balance of this account
  * after the debit successfully occurs.
  */
  public int debit(int amount)
```

Semi-Formal Specs

```
/** Debit this account.
  * @param amount the amount to debit.
  *         <code>amount</code> must be
  *         non-negative.
  * @result the balance of this account
  * after the debit successfully occurs.
  */
  public int debit(int amount)
```

- Many of the same questions arise even though the documentation is much clearer.

Formal Specs

```
/** Debit this account.
  * @param amount the amount to debit.
  * @result the resulting balance.
  */
/*@ requires 0 <= amount && amount <= balance;
  @ ensures balance == \old(balance - amount) &&
  @         \result == balance;
  @*/
  public int debit(int amount)
```

Writing and Calling Methods Incorrectly

```
/* Deduct some cash from this account and
   return how much money is left. */
  public int debit(int amount) {
    if (amount < 0) throw NDE(amount);
    if (balance < amount)
      throw NBE(balance);
    ...
  }
```

Writing and Calling Methods Incorrectly

```
/* Deduct some cash from this account and
   return how much money is left. */
  public int debit(int amount) {
    if (amount < 0) throw NDE(amount);
    if (balance < amount)
      throw NBE(balance);
    ...
  }
```

```
try {
  b = debit(a);
  if (b < 0) throw NBE();
} catch (Exception e) {
  System.exit(-1);
}
```

Writing and Calling Methods Incorrectly

```
/* Deduct some cash from this account and
   return how much money is left. */
  public int debit(int amount) {
    if (amount < 0) throw NDE(amount);
    if (balance < amount)
      throw NBE(balance);
    ...
  }
```

HORRIBLE!

```
try {
  b = debit(a);
  if (b < 0) throw NBE();
} catch (Exception e) {
  System.exit(-1);
}
```

Calling Methods Correctly

```
/*@ requires 0 <= amount && amount <= balance;
  @ ensures balance == \old(balance - amount) &&
  @         \result == balance;
  @*/
  public int debit(int amount) {
    ...all conditionals are gone!
    ...
  }

if (debit_amount < 0 || balance < debit_amount)
  handle_bad_debit(debit_amount);
else
  resulting_balance = debit(debit_amount);
```

Design by Contract

- **Contracts** are a **key concept** in robust software design and construction.
  - **Precondition**: an assertion that must be true before a method can be called
  - **Postcondition**: an assertion that is guaranteed to be true when a method returns.
  - **Invariant**: an assertion that is true of an object at *observable states*.

Design by Contract Example

CLASS	*CITIZEN*	**Part:** 1/1
TYPE OF OBJECT Person born or living in a country	**INDEXING** **cluster:** *CIVIL_STATUS* **created:** 1993-03-15 jmn **revised:** 1993-05-12 kw	

Queries	Name, Sex, Age, Single, Spouse, Children, Parents, Impediment to marriage
Commands	Marry. Divorce.
Constraints	Each citizen has two parents. At most one spouse allowed. May not marry children or parents. Spouse's spouse must be this person. All children, if any, must have this person among their parents.

Partial Class Features

- queries
  - spouse? single?
- commands
  - marry! divorce!
- constraints
  - at most one spouse is allowed
  - spouse’s spouse must be this person

Partial Class Sketch

```
Citizen my_spouse;
/*@ invariant (spouse() != null) ==>
  @           spouse().spouse() == this;
  @*/
Citizen spouse() { returns my_spouse; }
boolean single() { returns my_spouse == null; }

//@ requires single() && new_spouse != null;
//@ ensures !single() && spouse() == new_spouse;
//@ ensures spouse().spouse() == this;
void marry(Citizen new_spouse)
  { my_spouse = new_spouse;
    new_spouse.my_spouse = this; }

//@ requires !single();
//@ ensures single() && \old(spouse()).single();
void divorce()
  { my_spouse = null; my_spouse.my_spouse = null; }
```

Java Modeling Language

- The contracts we just saw were written in the Java Modeling Language (JML).
- JML is a notation for formally specifying the behavior and interface of Java classes and methods.
- Originally developed by Gary T. Leavens (Iowa State, now U. Central Florida) and others, now worked on by researchers worldwide (including me).

Java Modeling Language

- JML enables Design by Contract and **runtime assertion checking**, but also full **logical models** of Java classes.
- Why logical models? Often, class behavior can be specified in *one simple way*, which has *many possible implementations*.

Logical Models

- Consider a basic (unprioritized) queue data structure.
- *enqueue* and *dequeue* operations *mean the same thing*, regardless of the implementation of the queue - this is the logical model.
- **Model checking** compares a logical model to an implementation.
- JML enables the specification of logical models that can be used by model checkers.

Tools That Use JML

- Many tools understand JML.
- Obviously I can’t talk about them all here, but these are a few...
  - ESC/Java2 (University College Dublin)
  - Daikon (MIT)
  - Sireum/Kiasan (Kansas State)

ESC/Java2

- ESC/Java2 is a **static checker** - it performs analysis of source code *without running it*.
  - Other static checkers include FindBugs and CheckStyle, which check for common errors and style issues.
- ESC/Java2 uses an **automated theorem prover** to (try to) demonstrate that a particular piece of Java code is correct with respect to its JML specification.

ESC/Java2

- ESC/Java2 will typically say "this piece of code definitely fulfills its specification", or "this piece of code may violate its specification".
  - Occasionally, it will say "I don't know."
- ESC/Java2 also detects some common programming errors (null pointer exceptions, array indices out of bounds).

Daikon

- Daikon is an invariant detector.
- It runs a program, observes what the program does, and reports properties that were true throughout the execution.
- Helpful for adding specifications to legacy code that lacks them, or for discovering potentially overlooked invariants!

Sireum/Kiasan

- Part of the Sireum set of tools.
- Kiasan is a JML-based automatic verification and test case generation tool.
- It can detect various possible runtime problems, like ESC/Java2.
- It uses **symbolic execution** to analyze the possible behaviors of code and generate tests to exercise them.

More Tools

- There are many more tools out there that understand JML, and even more under development.
- Many of these tools are used in developing real-world systems.
- A new standard for a JML intermediate representation to make tool development easier in the future is also in the works.

Unit Testing

- **Unit testing** has been an important validation technique in software development for many years.
- A developer designs a set, or **suite**, of unit tests.
- Each test gives some input to the system and checks to see if it gets the correct output from the system.

Unit Testing Issues

- Devising good tests is hard.
- It’s easy for developers to miss things that need testing.
- Handwritten tests can also have bugs, so if a test fails, it’s not necessarily telling you what you think it is!

JMLUnit

- JMLUnit is a **unit test generator** for code specified with JML, based on the JUnit framework.
- Uses the preconditions and postconditions of methods as **test oracles**.
- Requires the developer to come up with a set of test *data*, but not to write any test *code*.

JML and JMLUnit Demo

JMLUnit

- JMLUnit is an excellent tool, but has several shortcomings:

JMLUnit

- JMLUnit is an excellent tool, but has several shortcomings:
  - Generated tests are only as good as the specs. (not much to do about this one)

JMLUnit

- JMLUnit is an excellent tool, but has several shortcomings:
  - Generated tests are only as good as the specs. (not much to do about this one)
  - Developer needs to create the test data, and sometimes extra generator code—not always in a straightforward way.

JMLUnit

- JMLUnit is an excellent tool, but has several shortcomings:
  - Generated tests are only as good as the specs. (not much to do about this one)
  - Developer needs to create the test data, and sometimes extra generator code—not always in a straightforward way.
  - Can use *extreme* amounts of memory.

JMLUnit

- JMLUnit is an excellent tool, but has several shortcomings:
  - Generated tests are only as good as the specs. (not much to do about this one)
  - Developer needs to create the test data, and sometimes extra generator code—not always in a straightforward way.
  - Can use *extreme* amounts of memory.
  - Tests only single method calls, not method call sequences

Current Projects

- *Semantics- and Specification-Aware Unit Testing*
- *Distributed Unit Testing*
- *OpenJML*
- *Verified Gaming*

Semantics- and Specification-Aware Unit Testing

- Rewriting/extending JMLUnit to address the shortcomings noted previously (test data generation, memory usage, method call sequences).
- Using the semantics of Java and the JML specifications of the system under test to determine test data and the parts of the system to test with them.

JMLUnitNG

- The first step in the process toward a robust successor to JMLUnit is *JMLUnitNG*, based on TestNG.
- Addresses the data issue by using (very basic) reflective test data generation.
- Addresses the memory issue by using a lazy evaluation method to generate tests rather than generating entire test suites in memory.
- Does not address the other issues.

JMLUnitNG

- Initial results are promising:
  - JMLUnitNG can run millions of tests in under 1 GB of Java heap space, in minutes to hours.
  - Significant progress over JMLUnit taking weeks to attempt to run the same tests, in 16GB of Java heap space, and failing to generate any results.

Distributed Unit Testing

- Comprehensive unit testing takes time, especially if one is generating huge numbers of tests (as might arise from the previous project).
- Automatically distributing the unit tests across multiple, networked machines allows them to be run more efficiently.
- Currently, a number of machines at UWT, Kansas State University, and University College Dublin form such a network.

OpenJML

- Helping to develop the *next generation* of JML tools - because the current generation only handles the Java language as it existed up to late 2004.
- OpenJML is a new JML compiler and associated tool set built atop OpenJDK, Sun's open-source version of Java.
- There will of course be an *OpenJMLUnit* as well (JMLUnitNG is only an intermediate step).

Verified Gaming

- A teaching-related project, in conjunction with University College Dublin.
- Developing Java versions of classic games with verification-centric software engineering methods and tools, as a way of teaching formal methods.
- UWT undergrads have worked on Space Invaders, Frogger, and Pac-Man; I have worked on Tetris.

Questions?