Assertions

Reading assignment


Homework #1, on the class web pages later today
Today's Reading Assignment

• JML home page:  
  http://www.eecs.ucf.edu/~leavens/JML/

• Overview paper:  
  Gary T. Leavens Yoonsik Cheon, "Design by Contract with JML," draft paper,  

• Experimental evaluation:  

Assertions

• Sometimes called  
  Self-checking software or  
  Design by Contract

• insert specifications about the intent of a system  
  • violation means there is a fault in the system

• during execution, monitor if the assertion is violated  

• if violated then report the violation
History of Assertions

- Alan Turing discussed using “assert statements” in algorithms, ~1947
- Assert statements used in formal verification to indicate what should be true at points in a program, ~1967
- Assertions advocated for finding faults during execution, ~1972
  - Based on preprocessors
- Assertions introduced as part of programming and specification languages, ~1975
  - Euclid, Alphard, …

History of Assertions

- Bertrand Meyer popularizes Design by Contract and included assertions as an integral part of Eiffel, an OO language

- Assertion capabilities for common programming languages, available but limited
  - C and Java have very limited assertion capabilities

- Sophisticated assertion tools available in the market or public domain
  - E.g., Parasoft, JML

- Assertions widely used in Industry (e.g., Microsoft and Google)
  - Experimental evidence of effectiveness
An Assertion Mechanism

- High-level language constructs for
  - representing logical expressions (typically Boolean-valued expressions) for characterizing invalid program execution states
  - associating logical expressions with well-defined states of the program (scope of applicability)
- Predefined (and usually limited) user-defined runtime response that is invoked if the logical expression is violated
- Automatic translation of the logical expressions into executable statements that evaluate the expressions on the appropriate states (scope) of the associated program
Language for representing logical expressions

• Usually use a notation that can be “easily” translated into the programming language
• Boolean expressions
  • Use variables and operators defined in the program
  • Must adhere to programming languages scoping rules
  • ASSERT $X < Y + Z$;
    where $X$, $Y$, and $Z$ are variables in the program defined in the scope where the assert stmt appears

• Example quantification notations
  • ForAll or \forall or \all
  • ThereExists or \exists or \some

Example of Quantification

• --ASSERT for all $I$, $(1 \leq I < N)$, $A[I] \leq A[I + 1]$

• --ASSERT for some $I$, $(1 \leq I < N)$, $A[I] \leq A[I + 1]$

• quantification not always supported since it can result in expensive computation
Language for representing logical expressions

- Often want to reference original value and current value of a variable
  - Example notations
    - Pre(X)
    - Old(X)
    - X'
  - E.g., ASSERT (x==old(x) + 1)

- Sometimes can only reference previous values in post conditions (post conditions explained shortly)

Examples of using old and current values

- --ASSERT for all I, (1 ≤ I ≤ N),
  \[ \text{old}(A[I]) = A[I] \]
  - Value of the array has not changed

- --ASSERT for all J, (1 ≤ J ≤ N)
  (for some I, (1 ≤ I ≤ N), old(A[J]) = A[I])
  - Permutation of the array
**Scope of an assertion**

- **Local assertion**
  - checked at the definition site
  - `ASSERT X > 10`

- **Global assertion**
  - defined over a specific scope, usually using the scoping rules of the programming language
  - must determine the locations that need to be checked,
    - `Global ASSERT X > 10`
    - Compiler/preprocessor must determine all the locations where X is defined/assigned and check that X is greater than 10

**Scope of an assertion (continued)**

- **Loop assertion (Loop invariant)**
  - Checked at each iteration at the designated point in a loop
  - E.g., `\(\text{loop_invariant} (I < \text{Max})\)`

- **Pre (and Post conditions)**
  - Checked at the start (and end) of a method each time it is invoked
  - E.g., `Pre, assumes, requires`
  - E.g., `Post, ensures, provides`
  - E.g., `returns (returned value of a function) promise (impact on all other variables)"`
**Scope of an assertion (continued)**

- **Class assertion (Class invariant)**
  - Checked at the return of each method in a class
  - E.g., class_invariant, \texttt{\textbackslash invariant}

- All of the above are syntactic sugar
  - Could create the code to get the same results
  - But, assertion mechanism greatly simplifies writing assertions

**More Advanced Assertion Language Capabilities**

- may be able to introduce additional (hidden) operators, operands, and types
  - e.g., length or top operator for stack
  - must be able to define the hidden entities in terms of the provided/visible entities

- --Global ASSERT $Z < \text{Bound (Q)}$
  - means that whenever $Z$ is assigned a value, it must be less than $\text{Bound (Q)}$,
    - where $\text{Bound(Q)}$ is visible wherever $Z$ is visible and
    - either $\text{Bound(Q)}$ is already defined in the program or is defined to be a hidden operation

- Hidden entities are created just to support assertions
An Assertion Mechanism

• High-level language constructs for
  • representing logical expressions (typically Boolean-valued expressions) for characterizing invalid program execution states
  • associating the logical expressions with well-defined states of the program (scope of applicability)
  • Predefined (and usually limited) user-defined runtime response that is invoked if the logical expression is violated
  • Automatic translation of the logical expressions into executable statements that evaluate the expressions on the appropriate states (scope) of the associated program

Responses to assertions

• Termination model
  • When an assertion is violated, issue an error report and terminate

• Failure and Warning model
  • 3 level model: failure, warning, no problem
    • On failure, issue an error report and terminate
    • On warning, issue an error report and continue
    • Continue as long as there is no problem
An Assertion Mechanism

- a high-level language
  - for representing logical expressions (typically Boolean-valued expressions) for characterizing invalid program execution states
  - for associating the logical expressions with well-defined states of the program (scope of applicability)
- predefined or user-defined runtime response that is invoked if the logical expression is violated
- automatic translation of the logical expressions into executable statements that evaluate the expressions on the appropriate states of the associated program

Assertion preprocessor

source code with assertions

assertion checking off

preprocessor

assertion checking on

instrumented source code

Translator (compiler or interpreter)
Execution Models

- Suppress assertion checking
  - Binary -> on or off
    - Easy to support with a preprocessor

- Multi-level
  - Select severity level to support
  - E.g., Suppress all assertions except those at severity level 3 and higher

Example

```java
public int binarySearch(int data [], int key){
    int lower = 0;
    int upper = data.length - 1;
    int location = -1;
    while (true) {
        if(upper < lower)
            { return (location) ;}
        else {
            location = midpoint(lower, upper);
            if (data [location] == key)
                {return (location); }
            else if (data[location] < key)
                {lower = location +1; }
            else
                { upper = location -1; }
        }
    }
}
```
**Example: requires clause**

```java
public int binarySearch(int data [], int key){
    int lower = 0;
    int upper = data.length - 1;
    int location = -1;
    while (true) {
        if(upper < lower)
            { return (location) ;}
        else {
            location = midpoint(lower, upper);
            if (data [location] == key)
                {return (location); } 
            else if (data[location] < key)
                {lower = location +1; }
            else
                { upper = location -1; }
        }
    }
}
```

**requires**

`(data != null) && \for all (int i = 0; i < data.length - 2; i++)
data[i] <= data[i + 1];`

---

**Example: return clause**

```java
public int binarySearch(int data [], int key){
    int lower = 0;
    int upper = data.length - 1;
    int location = -1;
    while (true) {
        if(upper < lower)
            { return (location) ;}
        else {
            location = midpoint(lower, upper);
            if (data [location] == key)
                {return (location); } 
            else if (data[location] < key)
                {lower = location +1; }
            else
                { upper = location -1; }
        }
    }
}
```

**ensures**

`\result == location &&
((\for all(int i=0; i<data.length-1; i++)
data[i] <= key)&&(location == -1)) ||
((\for some(int i=0; i<data.length-1; i++)
data[i] == key)&&(location == i));`
Example: alternative return clause

public int binarySearch(int data [], int key) {
    int lower = 0;
    int upper = data.length - 1;
    int location = -1;
    while (true) {
        if (upper < lower) {
            return (location);
        } else {
            location = midpoint(lower, upper);
            if (data [location] == key) {
                return (location);
            } else if (data[location] < key) {
                lower = location + 1;
            } else {
                upper = location - 1;
            }
        }
    }
}

ensures \result == location &&
((\forall i : 0 < i < data.length-1; i++) data[i] != key) &&
(location == -1)) || ((data[location] == key));

Example: promise clause

public int binarySearch(int data [], int key) {
    int lower = 0;
    int upper = data.length - 1;
    int location = -1;
    while (true) {
        if (upper < lower) {
            return (location);
        } else {
            location = midpoint(lower, upper);
            if (data [location] == key) {
                return (location);
            } else if (data[location] < key) {
                lower = location + 1;
            } else {
                upper = location - 1;
            }
        }
    }
}

promise (data != null) &&
data.length == \old data.length &&
\forall i : 0 < i < data.length-1; i++)
    data[i] == \old data[i];
**Note**

- Previous examples are not in any specific assertion notation

- JML has an assignable clause that indicates what can be assigned a value (and everything else can not be changed)
  - E.g., assignable location;
  - Can also indicate that nothing can be modified
  - E.g., assignable \nothing

**Example: internal assertions**

```java
public int binarySearch(int data[], int key) {
    int lower = 0;
    int upper = data.length - 1;
    int location = -1;
    while (true) {
        if(upper < lower) {
            return (location);
        }
        else {
            location = midpoint(lower, upper);
            /* location is the midpoint between upper and lower
            * assert location <= ((float)(lower + upper)/2.0) + 1.0
            * assert location >= ((float)(lower + upper)/2.0) - 1.0
            */
            if (data [location] == key) {
                return (location);
            }
            else if (data[location] < key) {
                lower = location +1;
            }
            else {
                upper = location -1;
            }
        }
    }
}
```
**Major objection to using assertions**

- storage and runtime overhead
  - often shown not to be a problem
  - need more empirical data
- optimization techniques could remove many of the assertions
  - basically *proving* that the assertion is valid
  - would expect that many of the assertions could be eliminated
    - preconditions are often redundant checks on the validity of the parameters

**Assertions versus Exceptions**

- Assertion violation => *error*
  - Predefined response
    - Error report
    - Terminate or continue
  - More expressive notation (e.g. All, Some, old, class invariant)
  - Can be turned on and off during deployment
- Exception violation => *unusual case*
  - Style guideline
    - exceptions should be reserved for truly exceptional situations
    - Outer context knows how to deal with the situation
  - Program-defined response
    - Handler
    - Different choices for resuming execution
    - Complex exception flow
  - Always part of the deployed code
Correct by Design

- Design/Code by contract
- Development method that incorporates assertions early in the design and coding process
- Eiffel, Bertrand Meyer included assertions as part of the language
  - An important component of any library
- Assertions were the most requested feature in Java
  - Unfortunately, Java 1.4 introduced a very limited assertion capability

Design by Contract

- Recognizes the widespread use of library components
  - Precondition clearly states what a component expects to be true
    - Obligations on the client
  - Post condition clearly states what the component is going to provide to the client
- Bertrand Meyer claims that this alone greatly reduces the number of defects in software
Assertions are becoming widely used in industry

• Microsoft strongly encourages the use of assertions

Microsoft Assertion Effectiveness Study

• paper presents an empirical case study of two commercial software components at Microsoft Corporation
  • Applied to two development tools that are part of the Visual Studio™
    • Written in C and C++
    • For each component, analyzed two internal releases, thus providing four data sets
      • Releases: A-1, A-2, B-1, B-2.
  • The developers systematically employed assertions
**Methodology**

- For each component, measured the number of assertions at the time of release
  - Used a rank correlation technique, so removed from the analysis all the files that have no assertions and no faults
    - Such files skew the results because they cannot be used to evaluate the efficacy of assertions and inflate the size of the study
    - Included files that have faults but no assertions as well as files with assertions but no faults
  - **Component Size in KLOCs; Assertion density**
    - A-1 104.03 KLOC, 26.63 assertions/KLOC
    - A-2 105.63 KLOC, 33.09 assertions/KLOC
    - B-1 372.04 KLOC, 37.09 assertions/KLOC
    - B-2 365.43 KLOC, 39.49 assertions/KLOC

**Methodology continued**

- After release, measured the faults for the component post-release until the next version is released
  - Using the assertion and fault information along with the size of the system (in KLOCs), computed the assertion density (number of assertions/KLOC) and fault density (number of faults/KLOC)
    - The definition of fault was confirmed by verifying if the entry in the bug database has a corresponding source code change
Hypotheses

- With an increase in the assertion density in a file, there is a statistically significant decrease in fault density
  - measured by post-release fault density

- The usage of software assertions in these components found a large percentage of the faults in the bug database

Results comparing assertions to faults

- For files that have low assertion density, there is a higher fault density
- Files with a low fault density have a higher assertion density
- By analyzing the faults, observed that on average 14.5% of the faults for component A and 6.5% of the faults for component B were detected by assertions
Comparing effectiveness to static analysis tools

- Used static analysis tools within Microsoft (FxCop, PREfast, and PREfix) to obtain the faults from the bug database for components A and B
- “Significantly more faults found using assertions except in case B”
- Would like to see assertions taught and used in undergraduate and graduate courses

Common Mistakes Using Assertions

- Assertions too general
  - Assert that I is a positive integer when it should be between 1 and 10
- Assertion too specific
  - Assert that I is a positive integer when it can actually also be =0
- Assertion tied to the current implementation instead of the specification
  - Want class invariants be as general as possible
  - Often the assertions associated with a method must be implementation specific
- Distinguish between assertions and exceptions
**Summary about Assertions**

- Assertions are a relatively easy way to improve software reliability.
  - Developing effective assertions takes care and insight.
- Assertion languages are accessible to most programmers.
- Assertions document intent and thus are useful beyond just runtime checking.
- Overhead is usually small, especially if optimization techniques are applied.
- Need more experimental data.
  - Which kinds of assertions are most useful?
  - What is the expected overhead?

**Bottom line:** Appears to be an effective approach with little execution overhead.
Being increasingly used in industry.

**Homework assignment**

- Develop *meaningful* assertions for at least two interesting classes.
  - All classes should have class invariants.
  - All loops should have loop invariants.
  - All methods should have pre and post conditions.

- More information will be posted on the web.