Assertions

Reading assignment

- A. J. Offutt, “A Practical System for Mutation Testing: Help for the Common Programmer,” Proceedings of the 12th International Conference on Testing Computer Software, Washington, D.C., June 1995, pp. 99-109.

Reading assignment

- L. A. Clarke, A. Podgurski, D. J. Richardson and Steven J. Zeil, "A Formal Evaluation of Data Flow Path Selection Criteria," IEEE Transactions on Software Engineering, 15 (11), November 1989, pp. 1318-1332.
- Background reading
  - S. Rapps and E. J. Weyuker, "Data Flow Analysis Techniques for Test Data Selection," Proceedings of the Sixth International Conference of Software Engineering, Tokyo, Japan, September 1982, pp. 272-277.

Assertions

- Self-checking software
- 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

History of Assertions

- Assertions introduced as part of programming and specification languages, 1975->
  - Euclid, Alphard, Clu, …
- Bertrand Meyer popularizes Design by Contract and includes assertions as an integral part of Eiffel, an OO language
- Assertion capabilities for common programming languages, available but limited

Parts of 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)
- automatic translation of the logical expressions into executable statements that evaluate the expressions on the appropriate states of the associated program
- predefined or user-defined runtime response that is invoked if the logical expression is violated

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

- Quantification
  - ForAll and ThereExists

Example

- --ASSERT for all I, $(1 \leq I < N)$,
  $A[I] \leq A[I + 1]$
- --ASSERT for some I, $(1 \leq I < N)$,
  $A[I] \leq A[I + 1]$
- not always supported since quantification can result in expensive computation

Language for representing logical expressions

- **Want to reference original value and current value of a variable**
  - **Pre(X) versus X**
  - **Old(X) versus X**
  - **X' versus X**

Example old and current values

- --ASSERT for all I, ($1 \leq I \leq N$),
  old(A[I]) = A[I]
  - Value of the array has not changed

- --ASSERT for all J, ($1 \leq J \leq N$)
  (for some I, ($1 \leq I \leq N$),old(A[J]) = A[I])
  - Permutation of the array

Scope of an assertion

- Local assertion
  - checked at the definition site
- 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
      must determine all the locations where X is defined and check that X is greater than 10

Scope of an assertion

- Loop assertion (Loop invariant)
  - Checked at each iteration at the designated point in a loop
- Class assertion (Class invariant)
  - Checked at the start and end of each method in a class
- Pre (and Post conditions)
  - Checked at the start (and end) of a method each time it is invoked
- All of the above are syntactic sugar

More Advanced Assertion Language Capabilities

- may be able to introduce additional (hidden) operators, operands, and types
  - e.g., length operator for stack
  - must be able to define the hidden entities in terms of the provided entities
  - --ASSERT Z < Bound (Q)
    - means that whenever Z is assigned a value, it must be less than Bound (Q),
      - where Bound(Q) is visible wherever Z is visible and
      - either Bound(Q) is already defined in the program or is defined to be a hidden operation

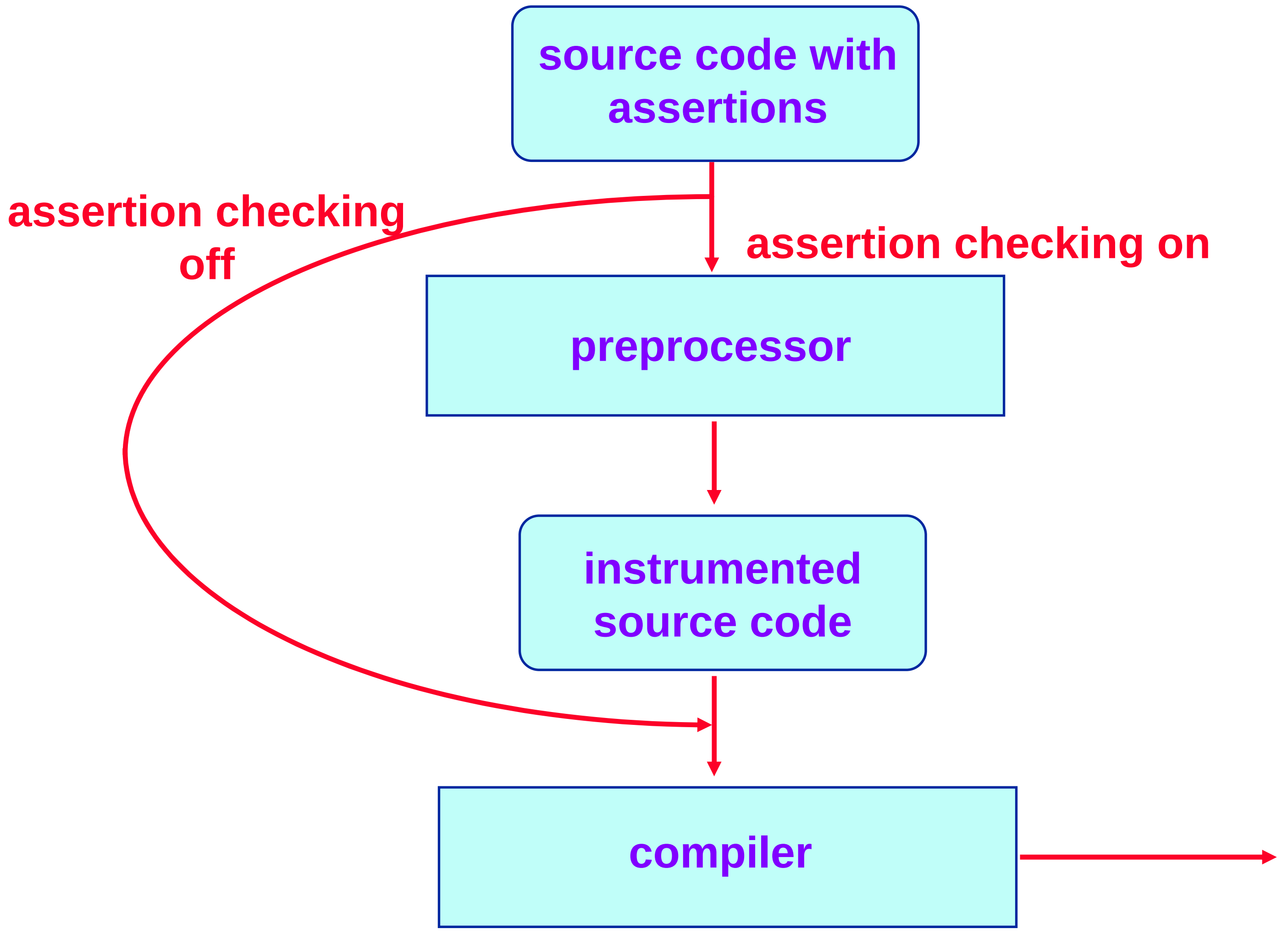
Parts of an Assertion Mechanism

- 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)
- automatic translation of the logical expressions into executable statements that evaluate the expressions on the appropriate states of the associated program
- predefined or user-defined runtime response that is invoked if the logical expression is violated

Execution Models

- Suppress assertion checking
  - Binary -> on or off
  - Multi-level
    - Select severity level to support
    - Suppress all assertions except those at severity level 3 and higher

Assertion preprocessor



Response model

- Termination model
  - When an assertion is violated, issue an error report and terminate
- Failure and Warning model
  - 2 (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

Annotation PreProcessor (APP)

- David Rosenblum
- APP supports 4 types of assertions
  - assume--specifies a precondition on a function
  - promise--specifies a postcondition on a function
  - return--specifies a constraint on the return value of a function
  - assert--specifies a constraint on an intermediate state of a function body
    - Where should these be placed?

APP

- provides quantification operations *all* and *some*
- default action gives an error message that prints information about the location of the violation and values of any variables involved
- users can define their own violation actions
- can associate severity levels with assertions
  - before processing indicate which severity levels should be checked

APP case study

- evaluated a program called Yeast
- 12,000 LOCs of C code
- half the program developed using APP
  - had specific rules for writing assertions
  - resulted in 116 assertions

Results of the study

- detected 19 known faults
  - 8 detected by APP
  - 6 would have been detected by APP, if it had been used
  - 2 detected by a dynamic storage certification routine
  - 3 required event sequence information
- Overhead of using assertions
  - 3.7% larger
  - no discernible difference in speed!

Example

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

Example: assume clause

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

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

Example: return clause

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

return location where
((all(int i = 0; i < data.length; i++) data[i] != key) && (location == -1)) ||
((some(int i = 0; i < data.length; i++) data[i] == key) && (location == i))

Example: promise clause

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

Example: internal assertions

```
public int binarySearch(int data [], int key){
  int lower = 0;
  int upper = data.length - 1;
  int location;
  while (true) {
     if(upper < lower)
        { return (-1) };
     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
  - Not apparent in APP study
  - 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)
- Exception violation => unusual case
  - Style guideline=> exceptions should be reserved for truly exceptional situations
  - Program defined response
    - Handler
    - Different choices for resuming execution
    - Complex exception flow

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
- Assertions are the most requested feature in Java
  - 1.4 introduced a very limited assertion capability

Summary about Assertions

- assertions are a relatively easy way to improve software reliability
- 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 overhead