Objects Analysis Threads Design 15-214

Principles of Software Construction: Objects, Design, and Concurrency

Formal Analysis of Software Artifacts

Charlie Garrod **Christian Kästner**

School of Computer Science
institute for SOFTWARE RESEARCH

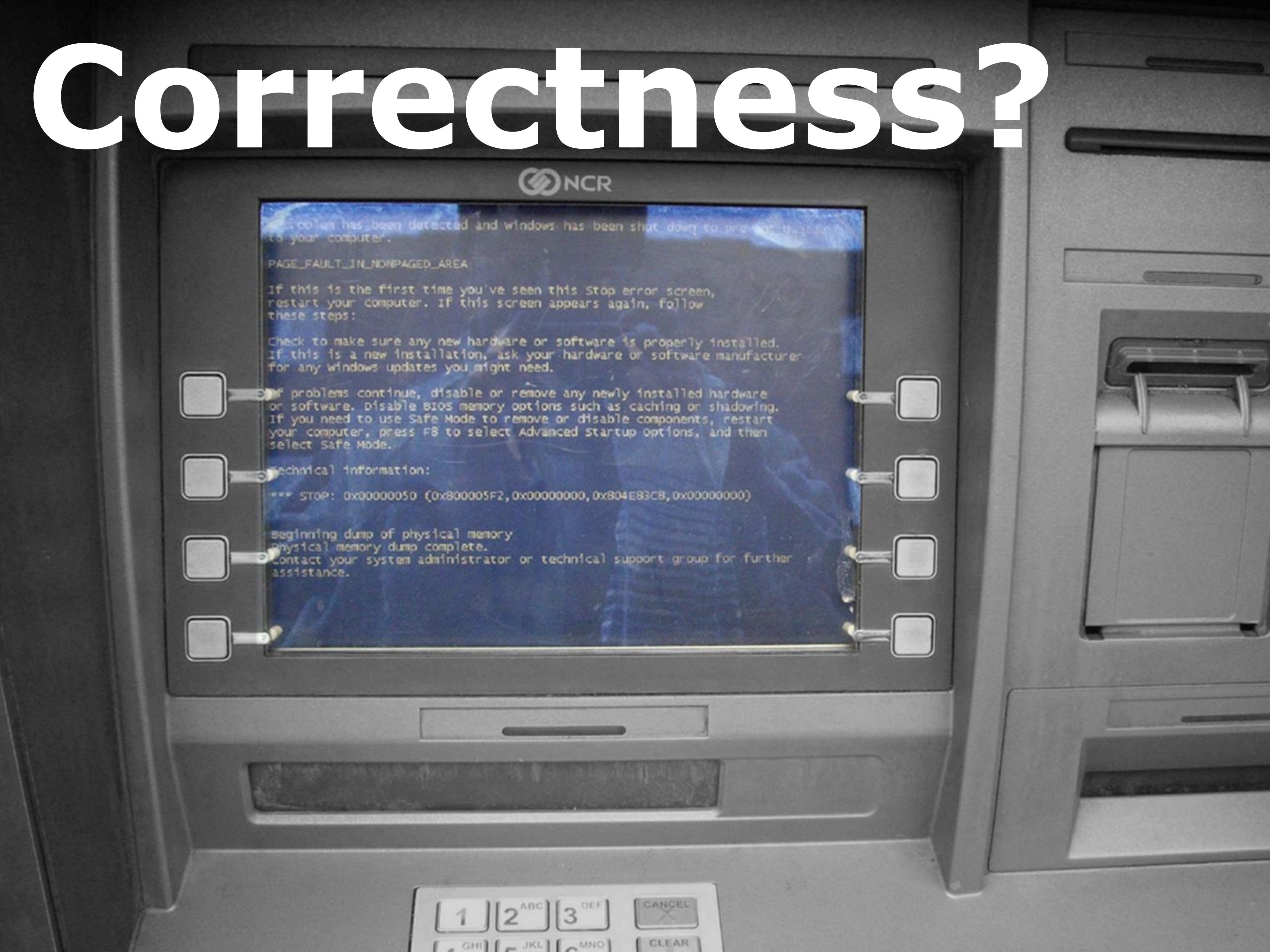
The four course themes

- Threads and Concurrency
  - Concurrency is a crucial system abstraction
  - E.g., background computing while responding to users
  - Concurrency is necessary for performance
  - Multicore processors and distributed computing
  - Our focus: application-level concurrency
  - Cf. functional parallelism (150, 210) and systems concurrency (213)
- Object-oriented programming
  - For flexible designs and reusable code
  - A primary paradigm in industry – basis for modern frameworks
  - Focus on Java – used in industry, some upper-division courses
- Analysis and Modeling
  - Practical specification techniques and verification tools
  - Address challenges of threading, correct library usage, etc.
- Design
  - Proposing and evaluating alternatives
  - Modularity, information hiding, and planning for change
  - Patterns: well-known solutions to design problems

Learning Goals

- Different strategies of quality assurance, different meanings of quality
- Importance of specifications (formal or informal)
- Formal verification vs. testing
  - Benefits and limits of formal verification
- Recalling formal reasoning with hoare logic, loop invariants and data structure invariants
- Reasoning in object-oriented programs: Class invariants and behavioral subtyping
- Demonstrate somewhat practical tools exists; rapid progress
  - Bridge to more lightweight mechanisms

Correctness?



Software Quality

Source: Braude, Bernstein, Software Engineering. Wiley 2011

- Sufficiency
  - Fails to implement the specifications … Satisfies all of the specifications
- Robustness
  - Will crash on any anomalous even … Recovers from all anomalous events
- Flexibility
  - Will have to be replaced entirely if specification changes … Easily adaptable to reasonable changes
- Reusability
  - Cannot be used in another application … Usable in all reasonably related applications without modification
- Efficiency
  - Fails to satisfy speed or data storage requirement … satisfies speed or data storage requirement with reasonable margin
- Reliability
  - Won't achieve required mean time between failure … will achieve the required mean time between failure
- Scalability
  - Cannot be used as the basis of a larger version … is an outstanding basis…
- Security
  - Security not accounted for at all … No manner of breaching security is known

Functional Correctness

- Specification

15-214

- Formal Verification
- Unit Testing
- Type Checking
- Statistic Analysis

15-313

- Requirements definition
- Inspections, Reviews
- Integration/System/Acceptance/Regression/GUI/Blackbox/ Model-Based/Random Testing
- Change/Release Management

Textual Specification

public int **read**(byte[] b, int off, int len) throws IOException

- Reads up to len bytes of data from the input stream into an array of bytes. An attempt is made to read as many as len bytes, but a smaller number may be read. The number of bytes actually read is returned as an integer. This method blocks until input data is available, end of file is detected, or an exception is thrown.
- If len is zero, then no bytes are read and 0 is returned; otherwise, there is an attempt to read at least one byte. If no byte is available because the stream is at end of file, the value -1 is returned; otherwise, at least one byte is read and stored into b.
- The first byte read is stored into element b[off], the next one into b[off+1], and so on. The number of bytes read is, at most, equal to len. Let *k* be the number of bytes actually read; these bytes will be stored in elements b[off] throughb[off+*k*-1], leaving elements b[off+*k*] through b[off+len-1] unaffected.
- In every case, elements b[0] through b[off] and elements b[off+len] through b[b.length-1] are unaffected.

- **Throws:**
  - IOException - If the first byte cannot be read for any reason other than end of file, or if the input stream has been closed, or if some other I/O error occurs.
  - NullPointerException - If b is null.
  - IndexOutOfBoundsException - If off is negative, len is negative, or len is greater than b.length - off

Textual Specifications

List:

boolean **addAll**(int index, Collection<? extends E> c)

Inserts all of the elements in the specified collection into this list at the specified position (optional operation). Shifts the element currently at that position (if any) and any subsequent elements to the right (increases their indices). The new elements will appear in this list in the order that they are returned by the specified collection's iterator. The behavior of this operation is undefined if the specified collection is modified while the operation is in progress. (Note that this will occur if the specified collection is this list, and it's nonempty.)

Parameters:

index - index at which to insert the first element from the specified collection

c - collection containing elements to be added to this list

Returns:

true if this list changed as a result of the call

Throws:

UnsupportedOperationException - if the addAll operation is not supported by this list

ClassCastException - if the class of an element of the specified collection prevents it from being added to this list

NullPointerException - if the specified collection contains one or more null elements and this list does not permit null elements, or if the

Specifications

- Contains
  - Functional behavior
  - Erroneous behavior
  - Quality attributes
- Desirable attributes
  - Complete
    - Does not leave out any desired behavior
  - Minimal
    - Does not require anything that the user does not care about
  - Unambiguous
    - Fully specifies what the system should do in every case the user cares about
  - Consistent
    - Does not have internal contradictions
  - Testable
    - Feasible to objectively evaluate
  - Correct
    - Represents what the end-user(s) need

Function Specifications

- A function's contract is a statement of the responsibilities of that function, and the responsibilities of the code that calls it.
  - Analogy: legal contracts
    - If you pay me $30,000
    - I will build a new room on your house
  - Helps to pinpoint responsibility
- Contract structure
  - Precondition: the condition the function relies on for correct operation
  - Postcondition: the condition the function establishes after correctly running
- (Functional) correctness with respect to the specification
  - If the client of a function fulfills the function's precondition, the function will execute to completion and when it terminates, the postcondition will be fulfilled
- What does the implementation have to fulfill if the client violates the precondition?

Formal Specifications

```
/*@ requires len >= 0 && array != null && array.length == len;
  @
  @ ensures \result ==
  @             (\sum int j;  0 <= j && j < len;  array[j]);
  @*/
int total(int array[], int len);
```

Java Modeling Language (JML)

- Language to specify hoare-style pre/post conditions in Java
- Written in Java comments, interpreted by separate tools
  - //@ <jml>
  - /*@ <jml> @*/
- Core syntax:
  - requires expr; ensures expr;
  - pure, non_null
  - invariant x; loop_invariant expr;
  - assert expr;
- Expressions:
  - \result, \old(expression)
  - (\forall <decl>; <range>; <body>), (\exists…)

http://www.jmlspecs.org/index.shtml

Quick Quiz

Assume the specification for sum given in the lecture slides:

requires array != null && len >= 0 && array.length == len

ensures \result == (\sum int j; 0 <= j && j < len; array[j])

Assume the following input and outputs for sum, where a 3 element array is written as [1, 2, 3]. For which of the inputs and outputs is the call and implementation of sum correct according to the specification given?

- Input: array = [1, 2, 3, 4], len = 4

  Output: 10
- Input: array = [0, 0, 3, -7], len = 4

  Output: *none (the program does not terminate)*
- Input: array = [1, 2, 3, 4], len = 3

  Output: 7
- Input: array = [1, 2, -3, 4], len = 4

  Output: 7

Quick Quiz

Assume the specification for sum given in the lecture slides:

requires array != null && len >= 0 && array.length == len

ensures \result == (\sum int j; 0 <= j && j < len; array[j])

Assume the following input and outputs for sum, where a 3 element array is written as [1, 2, 3]. For which of the inputs and outputs is the call and implementation of sum correct according to the specification given?

- Input: array = [1, 2, 3, 4], len = 4
  Output: 10
- Input: array = [0, 0, 3, -7], len = 4
  Output: *none (the program does not terminate)*
- Input: array = [1, 2, 3, 4], len = 3
  Output: 7
- Input: array = [1, 2, -3, 4], len = 4
  Output: 7

Call and implementation correc

Implementation incorrect, it should terminate

The call is incorrect (len should be 4)

Implementation incorrect, output should be 4

Erroneous Behavior Specifications

- A function can do anything at all if precondition is violated, BUT…
  - we may want the system to function even if one part fails
  - we may want to easily identify our mistakes
- Exceptional case specifications
  - Precondition: condition describing the input that leads to an error
  - Postcondition: condition established by the function under that erroneous input
- Example (BitSet.toArray() in JML)

```
/*@ public normal_behavior
    @   requires a!= null;
    @   requires (\forall Object o; containsObject(o);
    @                       \typeof(o) <: \elemtype(\typeof(a)));
    @ also
    @ public exceptional_behavior
    @   requires a == null;
    @   signals_only NullPointerException ;
    @ also
    @ public exceptional_behavior
    @   requires a != null;
    @   requires !(\forall Object o; containsObject(o);
    @                       \typeof(o) <: \elemtype(\typeof(a)));
    @   signals_only ArrayStoreException ;
    @*/
  Object[] toArray(Object[] a) throws NullPointerException, ArrayStoreException;
```

Example Java I/O Library Specification (abridged)

public int **read**(byte[] b, int off, int len) throws IOException

- Reads up to len bytes of data from the input stream into an array of bytes. An attempt is made to read as many as len bytes, but a smaller number may be read. The number of bytes actually read is returned as an integer. This method blocks until input data is available, end of file is detected, or an exception is thrown.
- If len is zero, then no bytes are read and 0 is returned; otherwise, there is an attempt to read at least one byte. If no byte is available because the stream is at end of file, the value -1 is returned; otherwise, at least one byte is read and stored into b.
- The first byte read is stored into element b[off], the next one into b[off+1], and so on. The number of bytes read is, at most, equal to len. Let *k* be the number of bytes actually read; these bytes will be stored in elements b[off] throughb[off+*k*-1], leaving elements b[off+*k*] through b[off+len-1] unaffected.
- In every case, elements b[0] through b[off] and elements b[off+len] through b[b.length-1] are unaffected.

- **Throws:**
  - IOException - If the first byte cannot be read for any reason other than end of file, or if the input stream has been closed, or if some other I/O error occurs.
  - NullPointerException - If b is null.
  - IndexOutOfBoundsException - If off is negative, len is negative, or len is greater than b.length - off

Example Java I/O Library Specification (abridged)

public int **read**(byte[] b, int off, int len) throws IOException

- Reads up to len bytes of data f
  attempt is made to read as ma
  The number of bytes actually r
  until input data is available, en
- If len is zero, then no bytes ar
  attempt to read at least one by
  end of file, the value -1 is retu
  into b.
- The first byte read is stored int
  on. The number of bytes read
  bytes actually read; these byte
  1], leaving elements b[off+*k*]
- In every case, elements b[0] through b[off] and elements b[off+len] through b[b.length-1] are unaffected.

- **Throws:**
  - IOException - If the first byte
    or if the input stream has been
  - NullPointerException - If b is n
  - IndexOutOfBoundsException -
    than b.length - off

- **Specification of return**
- **Timing behavior (blocks)**
- **Case-by-case spec**
  - **len=0 → return 0**
  - **len>0 && eof → return -1**
  - **len>0 && !eof →return >0**
- **Exactly where the data is stored**
- **What parts of the array are not affected**

- **Multiple error cases, each with a precondition**
- **Includes "runtime exceptions" not in throws clause**

Quality Attribute Specifications: Discussion

- How would you specify…
  - Availability?
  - Modifiability?
  - Performance?
  - Security?
  - Usability?

Runtime Checking of Specifications

```
/*@ requires len >= 0 && array.length == len
  @ ensures \result ==
  @             (\sum int j;  0 <= j && j < len;  array[j])
  @*/
float sum(int array[], int len) {
    assert len >= 0;
    assert array.length == len;
    float sum = 0.0;
    int i = 0;
    while (i < len) {
        sum = sum + array[i]; i = i + 1;
    }
    return sum;
    assert ...;
}
```

```
java -ea Main
```

Notation from the Java Modeling Language (JML)

Runtime Checking of Specifications

```
/*@ requires len >= 0 && array.length == len
  @ ensures \result ==
  @              (\sum int j;  0 <= j && j < len;  array[j])
  @*/
float sum(int array[], int len) {
     if (len < 0 || array.length != len)
          throw IllegalArgumentException(...);
     float sum = 0.0;
     int i = 0;
     while (i < len) {
          sum = sum + array[i]; i = i + 1;
     }
     return sum;
     assert ...;
}
```

java -ea Main

Notation from the Java Modeling Language (JML)

Can we prove this method correct for all inputs?

```
/*@ requires len >= 0 && array.length == len
  @
  @ ensures \result ==
  @              (\sum int j;  0 <= j && j < len;  array[j])
  @*/
float sum(int array[], int len) {
        float sum = 0.0;
        int i = 0;
        while (i < len) {
               sum = sum + array[i];
                         i = i + 1;
        }
        return sum;
}
```

Notation from the Java Modeling Language (JML)

Testing and Proofs

- Testing
  - Observable properties
  - Verify program for one execution
  - Manual development with automated regression
  - Most practical approach now
- Proofs
  - Any program property
  - Verify program for all executions
  - Manual development with automated proof checkers
  - Practical for small programs, may scale up in the future

23

Testing and Proofs

- Testing
  - Observable properties
  - Verify program for one execution
  - Manual development with automated regression
  - Most practical approach now

- Proofs
  - Any program property
  - Verify program for all executions
  - Manual development with automated proof checkers
  - Practical for small programs, may scale up in the future

- So why study proofs if they aren't (yet) practical?
  - Proofs tell us how to think about program correctness
  - Important for development, inspection, dynamic assertions
  - Foundation for static analysis tools
  - These are just simple, automated theorem provers
  - Many are practical today!

Hoare Triples

- Formal reasoning about program correctness using pre- and postconditions

- Syntax: {P} S {Q}
  - P and Q are predicates
  - S is a program

- Semantics
  - If we start in a state where P is true and execute S, then S will terminate in a state where Q is true

Hoare Triple Examples

- { true } x := 5 { }
- { } x := x + 3 { x = y + 3 }
- { } x := x * 2 + 3 { x > 1 }
- { x=a } if (x < 0) then x := -x { }
- { false } x := 3 { }
- { x < 0 } while (x!=0) x := x-1 { }

Hoare Triple Examples

- { true } x := 5 { x=5 }
- { x = y } x := x + 3 { x = y + 3 }
- { x > -1 } x := x * 2 + 3 { x > 1 }
- { x=a } if (x < 0) then x := -x { x=|a| }
- { false } x := 3 { x = 8 }
- { x < 0 } while (x!=0) x := x-1 { }
  - no such triple!

Hoare Logic Rules

- Assignments

  { P[E/x] } x:= E { P }

- Composition

  { P } S { Q }   { Q } T { R }

  { P } S; T { R }

- If statement

  { B & P } S { Q }   { !B & P } T { Q }

  { P } if (B) S else T { Q }

- While loop with loop invariant P

  { P & B } S { P }

  { P } while (B) S { !B & P }

- Consequence

  P -> P'   { P } S { Q }   Q -> Q'

  { P' } S { Q' }

122 Midterm

```
int find_peak_bin(int[] A, int n)
//@requires 0 < n && n <= \length(A);
//@requires is_peaked(A, 0, n);
//@ensures 0 <= \result && \result < n;
//@ensures gt_seg(A[\result], A, 0, \result);
//@ensures gt_seg(A[\result], A, \result+1, n);
{
int lower = 0;
int upper = n-1;
while (lower < upper)
      //@loop_invariant ______________________________ ;
      //@loop_invariant ______________________________ ;
{
      int mid = lower + (upper-lower)/2;
      //@assert _________________ ; /* optional */
      if (A[mid] < A[mid+1])
            lower = mid+1;
      else //@assert ______ ; /* optional */
            upper = mid;
}
//@assert ________________________ ; /* optional */
return lower;
}
```


institute for SOFTWARE RESEARCH

Quick Quiz

- Consider the following Hoare triples:
  - A) { z = y + 1 } x := z * 2 { x = 4 }
  - B) { y = 7 } x := y + 3 { x > 5 }
  - C) { false } x := 2 / y { true }
  - D) { y < 16 } x := 2 / y { x < 8 }
- Which of the Hoare triples above are invalid? What model witnesses the invalidity?
- Considering the valid Hoare triples, for which ones can you write a stronger postcondition? (Leave the precondition unchanged, and ensure the resulting triple is still valid)
- Considering the valid Hoare triples, for which ones can you write a weaker precondition? (Leave the postcondition unchanged, and ensure the resulting triple is still valid)

Quick Quiz

- Consider the following Hoare triples:
  - A) { z = y + 1 } x := z * 2 { x = 4 }
  - Invalid. A witness is [z=1, y=0]
  - B) { y = 7 } x := y + 3 { x > 5 }
  - Valid. A weaker precondition is { y > 2 }.
  - A stronger postcondition is { x == 10 }
  - C) { false } x := 2 / y { true }
  - Valid (any Hoare triple with a false precondition is valid)
  - A weaker precondition is { y != 0 }
  - We can choose any postcondition; the strongest is { false }
  - D) { y < 16 } x := 2 / y { x < 8 }
  - Invalid. A witness is [y=0]

31

Data Structure Invariants (rep. 122)

```
struct list {
	elem data;
	struct list* next;
};
struct queue {
	list front;
	list back;
};
```

Data Structure Invariants (rep. 122)

```
struct list {
    elem data;
    struct list* next;
};
struct queue {
    list front;
    list back;
};

bool is_queue(queue Q) {
    if (Q == NULL) return false;
    if (Q->front == NULL || Q->back == NULL) return false;
    return is_segment(Q->front, Q->back);
}
```

Data Structure Invariants (rep. 122)

```
struct list {
	elem data;
	struct list* next;
};
struct queue {
	list front;
	list back;
};

bool is_queue(queue Q) {
	if (Q == NULL) return false;
	if (Q->front == NULL || Q->back == NULL) return false;
	return is_segment(Q->front, Q->back);
}

void enq(queue Q, elem s)
//@requires is_queue(Q);
//@ensures is_queue(Q);
{
	list l = alloc(struct list);
	Q->back->data = s;
	Q->back->next = l;
	Q->back = l; }
```

Data Structure Invariants (rep. 122)

- Properties of the Data Structure
- Should always hold before and after method execution
- May be invalidated temporarily during method execution

```
void enq(queue Q, elem s)
//@requires is_queue(Q);
//@ensures is_queue(Q);
{ … }
```

Class Invariants

- Properties about the fields of an object
- Established by the constructor
- Should always hold before and after execution of public methods
- May be invalidated temporarily during method execution

Class Invariants

- Properties about the fields of an object
- Established by the constructor
- Should always hold before and after execution of public methods
- rily

```
public class SimpleSet {

    int contents[];
    int size;

    //@ ensures sorted(contents);
    SimpleSet(int capacity) { … }

    //@ requires sorted(contents);
    //@ ensures sorted(contents);
    boolean add(int i) { … }

    //@ requires sorted(contents);
    //@ ensures sorted(contents);
    boolean contains(int i) { … }
}
```

```
public class SimpleSet {

    int contents[];
    int size;

    //@invariant sorted(contents);

    SimpleSet(int capacity) { … }

    boolean add(int i) { … }

    boolean contains(int i) { … }
}
```

Behavioral Subtyping (Liskov Substitution Principle)

> Let q(x) be a property provable about objects x of type T. Then q(y) should be provable for objects y of type S where S is a subtype of T.
>
> Barbara Liskov

- An object of a subclass should be substitutable for an object of its superclass
- Known already from types:
  - May use subclass instead of superclass
  - Subclass can add, but not remove methods
  - Overridden method must return same or supertype
  - Overridden method may not throw additional exceptions
- Applies more generally to behavior:
  - A subclass must fulfill all contracts, the superclass does
  - Same or stronger invariants
  - Same or stronger postconditions for all methods
  - Same or weaker preconditions for all methods

Behavioral Subtyping (Liskov Substitution Principle)

```
abstract class Vehicle {
	int speed, limit;
	//@ invariant speed < limit;

	//@ requires speed != 0;
	//@ ensures |speed| < |\old{speed}|
	void break();
}
```

```
class Car extends Vehicle {
	int fuel;
	boolean engineOn;
	//@ invariant fuel >= 0;

	//@ requires fuel > 0 && ! engineOn;
	//@ ensures engineOn;
	void start() { … }

	void accelerate() { … }

	//@ requires speed != 0;
	//@ ensures |speed| < |\old{speed}|
	void break() { … }
}
```

Subclass fulfills the same invariants (and additional ones)
Overridden method has the same pre and postconditions

institute for SOFTWARE RESEARCH

Behavioral Subtyping (Liskov Substitution Principle)

```
class Car extends Vehicle {
        int fuel;
        boolean engineOn;
        //@ invariant fuel >= 0;

        //@ requires fuel > 0 && ! engineOn;
        //@ ensures engineOn;
        void start() { … }

        void accelerate() { … }

        //@ requires speed != 0;
        //@ ensures |speed| < |\old{speed}|
        void break() { … }
}
```

```
class Hybrid extends Car {
        int charge;
        //@ invariant charge >= 0;

        //@ requires (charge > 0 || fuel > 0)
                    && ! engineOn;
        //@ ensures engineOn;
        void start() { … }

        void accelerate() { … }

        //@ requires speed != 0;
        //@ ensures |speed| < |\old{speed}|
        //@ ensures charge > \old{charge}
        void break() { … }
}
```

Subclass fulfills the same invariants (and additional ones)
Overridden method start has weaker precondition
Overridden method break has stronger postcondition

Behavioral Subtyping (Liskov Substitution Principle)

```
class Rectangle {
        int h, w;

        Rectangle(int h, int w) {
          this.h=h; this.w=w;
        }

        //methods
}
```

```
class Square extends Rectangle {
        Square(int w) {
          super(w, w);
        }
}
```

Is Square a behavior subtype of Rectangle?

Behavioral Subtyping (Liskov Substitution Principle)

```
class Rectangle {
        //@ invariant h>0 && w>0;
        int h, w;

        Rectangle(int h, int w) {
          this.h=h; this.w=w;
        }

        //methods
}
```

```
class Square extends Rectangle {
        //@ invariant h==w;
        Square(int w) {
          super(w, w);
        }
}
```

Is Square a behavior subtype of Rectangle?

Behavioral Subtyping (Liskov Substitution Principle)

```
class Rectangle {
        //@ invariant h>0 && w>0;
        int h, w;

        Rectangle(int h, int w) {
          this.h=h; this.w=w;
        }

        void scale(int factor) {
          w=w*factor;
          h=h*factor;
        }
}
```

```
class Square extends Rectangle {
        //@ invariant h==w;
        Square(int w) {
          super(w, w);
        }
}
```

Is Square a behavior subtype of Rectangle?

Behavioral Subtyping (Liskov Substitution Principle)

```
class Rectangle {
        //@ invariant h>0 && w>0;
        int h, w;

        Rectangle(int h, int w) {
          this.h=h; this.w=w;
        }

        void scale(int factor) {
          w=w*factor;
          h=h*factor;
        }

        void setWidth(int neww) {
          w=neww;
        }
}
```

```
class Square extends Rectangle {
        //@ invariant h==w;
        Square(int w) {
          super(w, w);
        }
}
```

Is Square a behavior subtype of Rectangle?

Behavioral Subtyping (Liskov Substitution Principle)

```
class Rectangle {
        //@ invariant h>0 && w>0;
        int h, w;

        Rectangle(int h, int w) {
          this.h=h; this.w=w;
        }

        void scale(int factor) {
          w=w*factor;
          h=h*factor;
        }

        void setWidth(int neww) {
          w=neww;
        }
}
```

```
class Square extends Rectangle {
        //@ invariant h==w;
        Square(int w) {
          super(w, w);
        }
}
```

← Invalidates stronger invariant (w==h) in subclass

With these methods, Square is not a behavior subtype of Rectangle

ESC/Java

- "Extended Static Checker for Java"
- Originated from Compaq Systems Research Center, later open sourced
- Analysis as Compile Time
- Hoare-Logic-Like Proofs with automated theorem prover
- ESC/Java2 uses JML specifications
- Requires Java 1.6 or earlier(!)
- Neither sound nor complete; usability tradeoff
- Further Reading: Cormac Flanagan, K. Runstan M. Leino, Mark Lillibridge, Greg Nelson, James B. Saxe, and Raymie Stata. Extended Static Checking for Java. Proc. Programming Language Design and Implementation, June 2002.

ESC/Java: Global Specifications

- Checks several global specifications in addition to user-specified specifictions
    - a.b <= a shall not be null
    - a[b] <= b shall be within the bounds of a

```
void getF(FHolder fHolder) {
    return fHolder.f;
}
```

ESC/Java will give an error of the form “Warning: Possible null dereference (Null).” What is the best way to eliminate this warning?

ESC/Java Limitations

- Incomplete: Does not check for some errors:
  - Infinite loops, arithmetic overflow
  - Functional properties not stated by user
  - Non-functional properties
- Unsound: may miss some errors
  - Only checks one iteration of loops
  - @modifies is unchecked
  - Assumptions about invariants in referred-to objects
  - Several others as well!

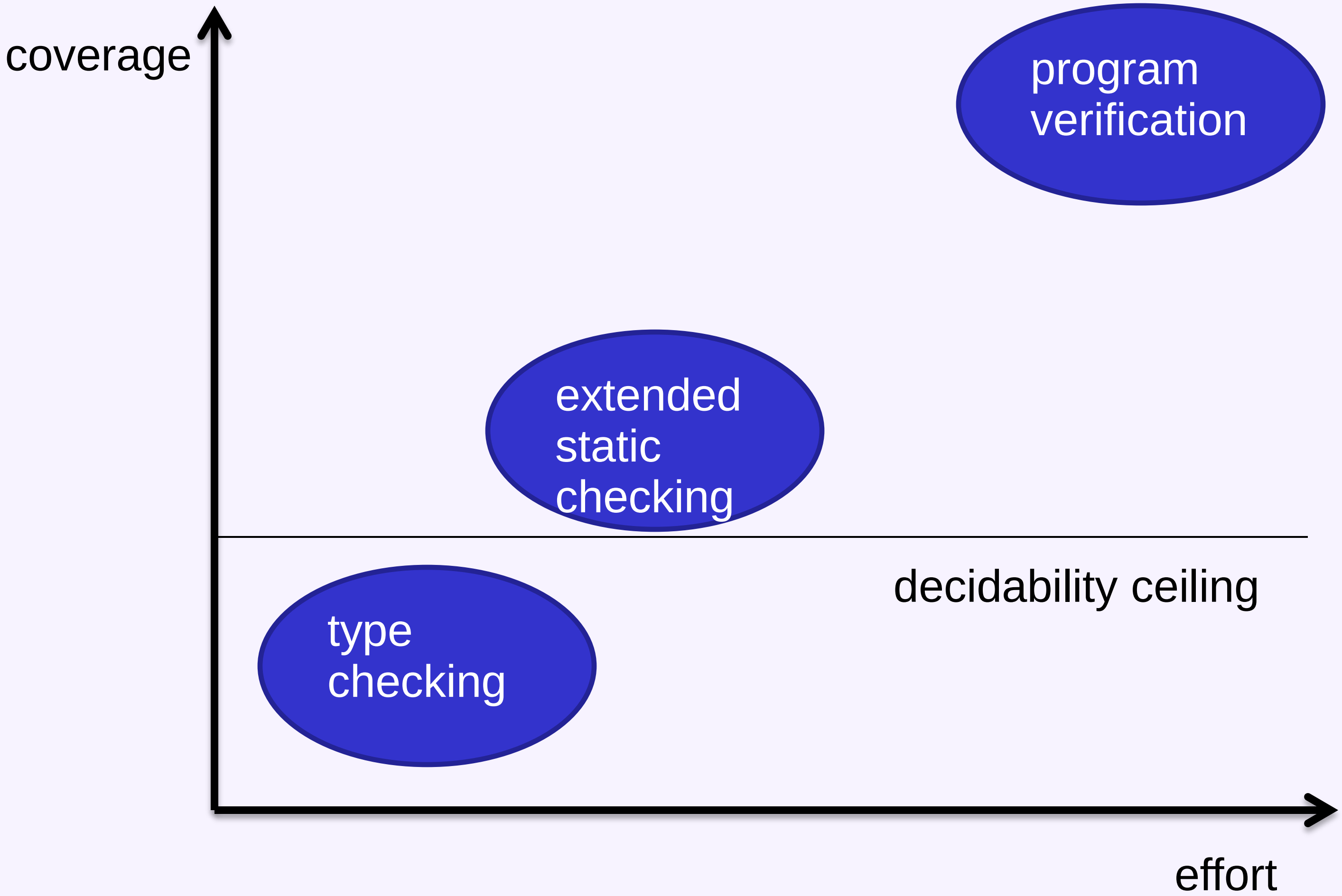
The loop:

```
//@ loop_invariant E;
while (B) {
  S
}
```

Is treated as:

```
//@ assert E;
if (B) {
  S
  //@ assert E;
  //@ assume !B;
}
```

ESC/Java Tradeoff



Source: Cormac Flanagan, K. Runstan M. Leino, Mark Lillibridge, Greg Nelson, James B. Saxe, and Raymie Stata. Extended Static Checking for Java. Proc. Programming Language Design and Implementation, June 2002.

Session Summary

- Specification between textual and formal specifictions
- Proving (Hoare Logic) vs Testing
- Class Invariants and Behavioral Subtyping
- Tools such as ESC/Java can make Hoare Logic-style checking much more practical
    - Reduces effort relative to proof by hand
    - Still considerable work in writing specifications and invariants
    - Can be useful in documenting code and finding errors
    - The current tool may miss some defects