Part I
Assertions and Contracts

1 Types are Your Friends

A type checker is a theorem prover. As it crawls over your program, it confirms your claims about your functions and variables once and for all. Example:

```
...
int f(long x) {
 return ... x ...;
}
...
```

When the Java type checker checks the program that contains this code fragment, it confirms two of your claims:

1. when `f` receives a value for `x`, it is an
2. if `f` produces a value, it is an element of the class of `int` integers.

While these facts don't represent much of the knowledge that went into the production of the program, it is still more than testing could ever establish. After all, testing can only confirm a finite number of facts about `f`.

Knowledge like this comes in handy when you are looking for errors in your program. As you inspect your program for potential problems, you do not have to re-confirm that `f` maps `long`s to `int`s; you may assume it as a given.

WARNING: The above is only true for program languages with sound type systems. Examples of such languages are Java, ML, and Haskell. In contrast, C and C++ do not have sound type systems; a function such as `f` can always consume a bit pattern that corresponds to a (portion of a string) and produce a (portion of a) bit pattern from a structure. Hence, when a C/C++ program breaks, you may not assume anything is correct.

In the ideal world, we should therefore express everything we know about our program in the type system. Then, as the type checker blesses the program, it would confirm that the program runs correctly for ever. Unfortunately, this is impossible in the abstract (if we wish to have terminating

type checkers) and in the concrete; the type languages of programming languages are just not expressive enough.

Consider this example:

```
prime h(prime x) {
 return ... x ...
}
```

This code fragment specifies that `h` consumes and produces prime numbers. No current type language supports a `prime` type at the moment, and including it in a type language would soon run into problems with the undecidable theory of arithmetic. Thus, even though such function signatures would be highly useful in, say, the realm of cryptography, it is impossible to state such facts and to have them confirmed by the type checker.

2 Assertions

(Types are not friendly enough)

To overcome this deficiency, we need to generalize the language of types to a language of arbitrary assertions. Roughly speaking, an assertion is a claim about the values in our computer programs as the (abstract) computer executes them. Historically, people have associated stated assertions about program variables. Examples:

```
int p; // p is a prime number
```

or

```
int x;
...
x = x * x; // x is now positive
...
```

or

```
Process current;
Queue[Process] q;
...
q.enq(current);
// q is not empty
```

```
runProcess(q.deq());
// warning: q might be empty now
...
```

The last two examples also show that assertions are established as the result of executing a statement, and using assertions, we can argue that code fragments will execute properly. Due to this role, assertions are often referred to as pre-conditions and post-conditions (of statements).

Since English is ambiguous and since assertions (claims of truth) are the elements that logic deals with, computer scientists quickly adopted the language of logic to express facts. Thus, instead of the above, people write:

```
// forall i such that 2 $\leq$ i < p,
//       not[(p modulo i) = 0]
int p;
```

and

```
int x;
...
x = x * x; // x >= 0
...
```

or

```
Process current;
Queue[Process] q;
...
q.enq(current);
// not[q.empty()]
runProcess(q.deq());
// WARNING: q.empty() v not[q.empty()]
...
```

Next people formulated rules for reasoning about program statements, especially assignment statements, if statements, and loops.

3 Assertions Describe the Behavior of Programs

Take a look at this program fragment:

```
int x = 0;
// 1: x = 0
x = x + 1;
// 2: x = 1
```

The comments are logical assertions about the state of the program. Indeed, they actually describe the execution of the program. Specifically, the first assertion describes the entire state after initialization and the second one the state after the assignment statement.

For if-statements, we need to take into account a case split:

```
int x = ...;
int y = ...;
...
int max = 0;
// 1: max = 0
if (x > y) {
 // 2a: (x > y)
 max = x;
 // 3a: x > y => max = y
 }
else {
 // 2b: not(x > y)
 // 2b: x $\leq$ y
 max = y;
 // 3b: x $\leq$ y => max = y
 }
// 4: (x > y => max = x) ^ (x $\leq$ y => max = y)
```

In each branch, we can add the condition that succeeded (and calculate with it: see 2b). Then, at the end of the if-statement, we can combine the two branches with a logical conjunction. In this example, we may conclude that no matter what `x` and `y` are, `max` is equal to the larger of the two.

Last but not least, we can also use assertions to describe the execution of loops:

```
int a[];
...
int i = 0;
int sum = 0;
// 1: i = 0 ^ sum = 0
```

```
...
// 2: sum = &Sigma; { a[j] | 0 $\leq$ j < i } ^ i = 0
for(i = 1; i < a.length; i++) {
 sum = sum + a[i];
 // 3: sum = &Sigma; { a[j] | 0 $\leq$ j < i }
 }
// 4: sum = &Sigma; { a[j] | 0 $\leq$ j < i = a.length }
```

Here we see that one and the same assertion holds before the loop starts (2), at the end of each loop body (3), and after the loop stops. The conclusion is then that `sum` is the sum of all numbers in the array `a`. Because this assertion holds at all these places, it is also called a loop invariant (i.e., an assertion that is true across several statements).

(Note: the assertion is false at the entry of the loop body. At that point `i` was just increased by 1, yet `sum` is still the old value.)

For years (1960's through the early 1980's), the holy grail of programming language research was the development of a calculus that would empower programmers to specify the initial state and the final state, and to create the program from these statements. Alternatively, the programmers would code something and an automatic theorem proofer would verify the result. (See Dijkstra's monograph.)

The effort collapsed when people couldn't figure out how to easily scale this kind of reasoning to programming languages with procedure definitions and procedure calls (and beyond). The idea of programming triples, however, survived. To this day, it is a good idea to describe small code fragments as Hoare triples:

```
PRE  programStatement  POST
```

where `PRE` and `POST` are logical statements describing the state of the abstract machine. Since people couldn't reason about several procedures, it is also natural that they started describing the behavior of procedures with triples.

4 Assertions for Method Boundaries

Although research on program correctness per se failed, the primary use of `PRE` and `POST` assertions is to enhance the type signature of a method. In other words, if these descriptions are added in an interface, they become a powerful tool to specify those obligations of callers and callees that go

beyond types. Even if such assertions no longer describe the entire behavior of a method's body, they can still notice basic mistakes that can pollute software for a long time.

Let us look at a canonical example:

```
interface Queue {
 // is this queue empty?
 boolean empty();

 // add Item x to the end of this queue
 Queue enq(Item x);

 // produce and remove the first Item from the queue
 Item deq();

 // how many items are in this queue
 int size();
}
```

This Java interface specifies a collection of data structures with four methods. At first a newly created queue is empty. With `enq`, we can add an item; with `deq` we can retrieve the item. The descriptions of the two actions imply that they increase and decrease the `size` of the queue. Their names and the name of the data structure finally suggest that `deq` retrieves the item that has been in the queue the longest. In short, our historical knowledge suggests a first-in/first-out data structure.

In principle, it is possible to supplement the signatures and purpose statements of this interface with assertions in an equational logic that describe the behavior we just described informally. Here are just some sample equations:

```
$\forall$ X: new Queue().enq(X).deq() = X
$\forall$ X, Y: new Queue().enq(X).enq(Y).deq() = X

new Queue().size() = 0
$\forall$ X: new Queue().enq(X).size() = 1
```

Given the stateful specification of the queue interface, however, it is impossible to formulate a simple set of equations or logical assertions that describe the complete behavior of `Queue`.

Still, many aspects of our informal descriptions can be translated into logical assertions, specifically into pre- and postconditions. Let's practice this idea with two questions:

1. Can the method be called in all situations? If not, describe when it can be called.
2. Is there anything special about the result value or the result state of each method?

Applied to our four methods we get these answers:

1. It is always possible to call `empty`. Its result is either true or false, and there is nothing more specific that we can say now.
2. It is always possible to call `enq`. Its result is a queue that is not empty. Furthermore, the size after the call is one more than the size before the call.
3. It is only possible to call `deq` when the queue contains at least one item. Otherwise it makes no sense to call it. Furthermore, at the end of the call the queue contains one item less than before the call.
4. Finally, it is always possible to call `size`. Its result is always an integer, though it is also true that this integer is always greater or equal than 0.

Here is a rewrite of the same interface with `PRE` and `POST` conditions:

```
interface Queue {
 // is this queue empty?
 boolean empty();

 // add Item x to the end of this queue
 Queue enq(Item x);
 // POST: !(empty()) && size() = OLD[size()] + 1

 // produce and remove the first Item from the queue
 Item deq();
 // PRE: !(empty())
 // POST: size() = OLD[size()] - 1

 // how many items are in this queue
```

```
 int size();
 // POST: RESULT &ge; 0
}
```

The conditions contain two special notations. The first is `OLD[...]`. It denotes the value of the expression *before* the method call took place. The second one is `RESULT`. It is the value that the method produces. Given these explanations, the pre- and postconditions in the revised interface express all those conditions that we mentioned above, except for those that provide a full-fledged formal description of the methods' behavior. Still, experience shows that such additional conditions quickly catch mistakes in the general logic, especially in conjunction with other pre- and postconditions.

Let us look at a second example, the conversions of dollar amounts into strings and vice versa. Since money amounts require precise calculations, it is a good practice to develop a separate class of `Amount`s for business applications. Naturally, in addition to basic arithmetic operations on an `Amount` we also need an operation that converts an instance to a `String`:

```
interface IAmount {
  // convert the amount into a check-style string
  // with dollars and cents separated by a dot
  String print()
}
```

Furthermore, we may also want to provide a static factory method that reads a `String` into an `Amount:`

```
class Amount implements IAmount {
  ...
  // convert the given check-style string into an amount
  // assume the string separates the dollar amount from
  // the cents via a dot
  public static Amount read(String s) { ... }
  ...
}
```

In either case, it makes sense to describe the obligation of `print` or the obligation of `read`'s client with assertions that catch basic mistakes.

The informal purpose statement implies two major characteristics about the string so far:

1. the string consists of digits except ...

2. ... for the third spot from the right, which is a dot.

Using a forall quantifier, we can express these two constraints as follows:

```
RESULT[RESULT.length() - 3] = '.'
&&
$\forall$ i in [0..RESULT.length] : i != RESULT.length() - 3
 => RESULT[i] in ['0'..'9']
```

As before, the assertion uses both program variables and even programming notation to express basic ideas but also mathematical symbols, e.g., ∀. Furthermore, it does not express the complete behavior of either `print` or `read` but it expresses essential characteristics. In particular, it clarifies what it means to supply or receive a check-style string.

If this condition fails, something is seriously wrong with the method or its caller. Hence monitoring conditions such as these helps us defend ourselves against miscommunication among programmers. The key is then to not only add such statements but to turn them into a part of the running code.

5 From Assertions to Contracts

In the business world, a contract is a document that spells out the obligations of two (or more) participants in a joint activity. For example, a contract may specify that an oil company delivers oil to a residence and that the owner in turn pays for the oil. If either of the two parties doesn't live up to its obligations, the other one will go to a neutral arbiter who decides whether the claim is correct. If so, we say that one of the two parties got blamed.

This scenario suggests that turning assertions into useful program monitoring tools requires three things:

1. the assertions must become executable;

2. a neutral arbiter must monitor the assertions;

3. if an assertion goes wrong, the monitor must assign blame to the party that fails to live up to its obligations.

In short, assertions convey knowledge and you can be blamed for what you knowingly violate; converting assertions into contracts does just that.

We can go about making assertions executable in two ways. On one hand, we can create a notation for assertions, have programmers express their thoughts in this notation, and translate it into code. On the other hand, we can simply say that all boolean expressions of the underlying programming language are assertions and that the monitor just executes them at opportune moments.

The boolean-expression approach is dominant in industrial languages though only Eiffel treats these assertions as contracts. Meyer chose to equate boolean expressions with assertions because of pragmatic reasons, even though he was keenly aware of the problems. The major problem is of course that a call to a boolean-value function is an assertion but the execution of this assertion may have side-effects. That is, it may diverge, raise an exception, or change the state of a variable.

Research projects tend to use separate languages for contracts and assertions. Two prominent examples are the ESC/Java project at the former DEC SRC lab (short for extended static checking), which uses the Java Modeling Language (JML), and the SPEC# project at Microsoft research, which uses its own notation. The latter provides a programming language, a runtime monitoring system, and a theorem proofer. If the theorem proofer can verify that a party always meets its obligations, it can inform the compiler, which will then disable the corresponding monitoring code. Conversely, if the theorem prover cannot establish that the party lives up to the contract in all situations, the compiler will issue appropriate code.

No matter how contract violations are found, it is important that the guilty party is blamed and that the error message is a good explanation for the problem. After all, the purpose of contracts is to protect programmers against miscommunication and to help them eliminate problems.

Note 1: Unfortunately, most contract systems for Java and Eiffel lack a semantic foundation. As a result, they may not notice a contract violation at all, blame the wrong party, or blame the proper party with an incorrect explanation. Furthermore, the pun between assertion variables and program variables has prevented programmers from expressing and monitoring assertions about objects (rather than just plain values).(See Findler's dissertation and research for detailed explanations.)

Note 2: Monitoring contracts can get expensive. It is therefore important to think about what you want your contract system to check. The goal is to find the proper compromise between checking basic properties and checking them without much overhead in time, space, and energy. A basic

property of a method is an assertion whose violation suggests something fundamental went wrong. For example, if there is no dot at the proper position in the result of `print` in `Amount`, it is not the desired print representation of a dollar amount. Checking this one character in the string is relatively cheap; it adds a constant to the running time. Checking that every other character is a digit requires a loop. The cost is now linear in the length of the resulting string but that is acceptable because the routine itself is probably linear. Adding more than a linear cost would definitely be questionable.

6 Sequence Contracts

In addition to types (signatures) and basic conditions about the before and after state of the world, we often also want to ensure that methods are called in the proper order.

Take a look at this simple integer file interface:

```
interface IPort {
    // connect port to file
    void open();

    // read one integer from file, if the file isn't exhausted
    int read();
    // @pre: !out\_of\_intsP()

    // is the file content exhausted?
    boolean out\_of\_intsP();

    // disconnect the port from the file
    void close();

    // sequence contract: open . {read | out\_of\_intsP}* . close
}
```

Presumably, we create an IPort with a name and then work with it:

```
class PortTest {
    public static void main(String argv[]) {
int sum;
int silly[] = {1,2,3};
```

```
IPort ip = new PortState(silly);

// ip.open();
sum = 0;
while (!ip.out\_of\_intsP() || true)
    sum += ip.read();
ip.close();
System.out.println("6 == " + sum);
    }
}
```

Clearly, the intention is that consumers call the methods of IPort in a particular sequence, namely, one call to `open`, followed by calls to `read` and `out_of_intsP`, followed by one (optional) call to `close`.

We call this a sequence contract and write something like the above as

```
open . { read | out\_of\_intsP }* . close
```

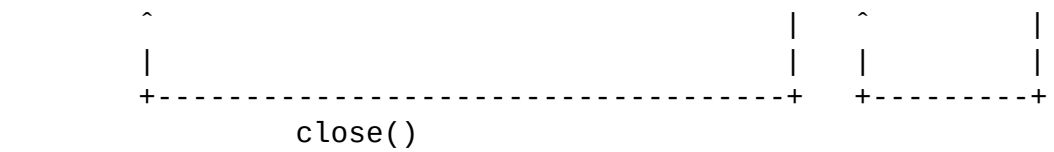
The notation:

- The braces group things.
- The brackets make things optional.
- a . b means a followed by b.
- a — b means a or b.
- a* means 0 or more actions a.
- a+ means 1 or more actions a.

An alternative (but equivalent) notation is that of finite state machines:

```
 port :

                open()                              read(),
      +-------------------------------------+     out\_of\_intsP()
      |                                     |   +---------+
      |                                     v   |         |
+------------+                          +----------+     |
| ClosedPort |                          | OpenPort |     |
+------------+                          +----------+     |
```



This notation suggests an implementation via the state pattern [GoF]:

```
{\tt
// Checking sequence contracts via the state pattern.

// Each class of states is represented with a single instance from a
// private class. Each state implements the full repertoire of method
// calls; the Port class itself just forwards each message to the current
// state.

class PortState implements IPort {

    // ------------------------------------------------------------------
    // fields:

    private int values[];

    private IPort state;

    private OpenPort theOpenPort = new OpenPort();

    private ClosedPort theClosedPort = new ClosedPort();

    PortState(int values[]) {
this.values = values;
state = theClosedPort;
    }

    // ------------------------------------------------------------------
    // the public interface

    public void open() {
state.open();
    }
```

```
    public int read() {
return state.read();
    }

    public boolean out\_of\_intsP() {
return state.out\_of\_intsP();
    }

    public void close() {
state.close();
    }

    // ------------------------------------------------------------------
    // the private classes

    private class OpenPort implements IPort {

private int ptr = 0;

public void open() {
    System.out.println("can't call open on open file");
    System.exit(-1);
}

public int read() {
    preRead();
    int r = values[ptr];
    ptr++;
    return r;
}

public boolean out\_of\_intsP() {
    return (ptr >= values.length);
}

public void close() {
    state = theClosedPort;
}

// checking the precondition for read
```

```
private boolean preRead() {
    if (ptr < values.length)
return true;
    System.out.println("precondition violation");
    System.exit(-1);
    return false;
}
    }

    private class ClosedPort implements IPort {

public void open() {
    state = theOpenPort;
}

public int read() {
    Ouch(); return -1;
}

public boolean out\_of\_intsP() {
    Ouch(); return false;
}

public void close() {
    Ouch();
}

private void Ouch() {
    System.out.println("sequence contract violation");
    System.exit(-1);
}
    }
}
}
```

7 Contract Violations and Fault-Tolerant Programming

When contracts go wrong, the system is in trouble. Contracts monitor basic properties that should never go wrong. At the same time, however, the

purpose of contracts is to check on the communication between disjoint parts of the system, and in common cases, the various parts of a system are not equals. If a lesser part fails, the overall system might just work at acceptable levels without the failing part.

This reasoning suggests a natural combination of contracts with techniques from fault-tolerant systems. Specifically, imagine that the composition of the system is flexible. While it runs, various components can join or leave the system. Naturally, the system contains essential components for which this is not true, but it might be true for peripheral or for redundant parts. Then, when one of those redundant or peripheral subsystems fails, the kernel of the system can "unplug" the failing part and inform the rest of the components to work without it. Indeed, depending on how the system is configured, the remaining components don't ever need to know.

Consider a system that is playing the games. The game administrator is obviously an essential component. If it fails, nothing else can replace it. In contrast, if one of the player fails—perhaps due to a bad connection or perhaps due to cheating—the game system can obviously do without the player.

The internet is another example of a fault-tolerant system. When computers notice that some other computer no longer responds with acknowledgments, they route network traffic around the failing computer. If the failing computer is a mail server, they may hold the mail for several days and attempt to redeliver it on a regular basis.

In general, the use of assertions (invariants) and contracts has led to a number of recent attempts to produce self-healing or self-repairing systems. While some ideas have already appeared in industrial contexts, for many it is too early to tell whether they will make it out of the research lab.

8 Bibliography

- Dijkstra. A Discipline of Programming. Prentice Hall. 1974.
- Findler. Behavioral Software Contracts. Dissertation, Rice University, 2001.
- Flanagan, Leino, Lillibridge, Nelson, Saxe, and Stata. Extended static checking for Java. PLDI 2002.
- Meyer. Applying "Design by Contract". IEEE Computer, Oct 1992.

- Barnett, Leino, Schulte. The Spec# programming system: An overview. CASSIS 2004.