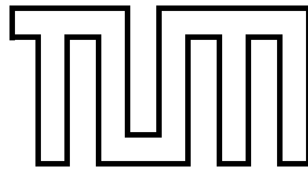


FAKULTÄT FÜR INFORMATIK

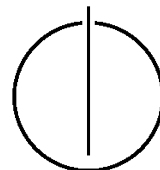
DER TECHNISCHEN UNIVERSITÄT MÜNCHEN

Master-Seminar Software Verification

Author: Lukas Erlacher

Advisor: Prof. Andrey Rybalchenko, Dr. Corneliu Popeea

Submission: April, 2013



Contents

1 Introduction **3**

2 ESC/JAVA checking pipeline **3**

3 Programmes **6**

3.1 Counter.java 6

3.2 Account.java 7

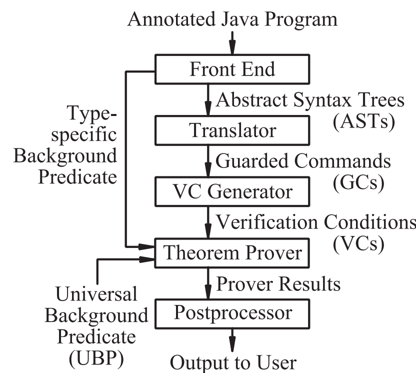
References **9**

1 Introduction

This[1] is the handout for my talk[2] for the Master Seminar Program Verification in the 2013 summer term at TUM.

It covers additional information about the ESC/JAVA static checker that goes beyond the scope of the talk, but is nevertheless interesting: We outline the proving pipeline in ESC/JAVA and show two example programmes demonstrating ESC/JAVA's concurrency features.

2 ESC/JAVA checking pipeline



The basic steps in ESC/Java's operation. (from [1])

ESC/JAVA uses a 5-stage pipeline to turn a java program into prover output. The "front end" is a parser very similar to java's that creates as AST. This abstract syntax tree is then translated into a guarded command (GC) language expression, essentially a dialect of Dijkstra's guarded command language [5]. This guarded command language has semantics defined by verification conditions. Let us first observe the syntax of ESC/JAVA's guarded commands (specified in [3]):

⟨*cmd*⟩ ::= ⟨*variable*⟩ = ⟨*expr*⟩
| 'skip'
| 'raise'
| 'assert' ⟨*expr*⟩
| 'assume' ⟨*expr*⟩
| 'var' ⟨*variable*⟩+ 'in' cmd 'end'
| ⟨*cmd*⟩ ; ⟨*cmd*⟩
| ⟨*cmd*⟩ ! ⟨*cmd*⟩
| ⟨*cmd*⟩ [] ⟨*cmd*⟩

[1] http://home.in.tum.de/~erlacher/sem/verification_2013_handout.pdf
[2] http://home.in.tum.de/~erlacher/sem/verification_2013_slides.pdf

The intuitive meaning of most of these constructs should be obvious; it remains to state that $C_1; C_2$ signifies sequential execution of C_2 after C_1, $C_1!C_2$ signifies the execution of C_2 if C_1 terminates exceptionally, and $C_1[]C_2$ is a nondeterministic choice between the execution of C_1 or C_2.

To derive how a GC program terminates - normally, exceptionally, or erroneously, we define the notion of *weakest liberal precondition*. This is a predicate $wlp.C.(N, X, W)$ that holds for exactly those initial states (state before execution of C) where normal execution of C terminates in state N, exceptional execution in X, and erroneous execution in W.

wlp is defined by the following equations:

$$\begin{array}{rcl}
wlp.(v = e).(N, X, W) & \equiv & N[v \leftarrow e] \\
wlp.\mathbf{skip}.(N, X, W) & \equiv & N \\
wlp.\mathbf{raise}.(N, X, W) & \equiv & X \\
wlp.(\mathbf{assert}\ e).(N, X, W) & \equiv & (e \wedge N) \vee (\neg e \wedge W) \\
wlp.(\mathbf{assume}\ e).(N, X, W) & \equiv & e \Rightarrow N \\
wlp.(\mathbf{var}\ v_1 \ldots v_n\ \mathbf{in}\ C\ \mathbf{end}).(N, X, W) & \equiv & \forall\ v_1 \ldots v_n \quad wlp.C.(N, X, W) \\
wlp.(C0\ ;\ C1).(N, X, W) & \equiv & wlp.C0.(wlp.C1.(N, X, W), X, W) \\
wlp.(C0\ !\ C1).(N, X, W) & \equiv & wlp.C0.(N, wlp.C1.(N, X, W), W) \\
wlp.(C0\ \Box\ C1).(N, X, W) & \equiv & wlp.C0.(N, X, W) \wedge wlp.C1.(N, X, W)
\end{array}$$

A GC and corresponding weakest precondition is derived for every routine in the java program.

The next stage, the verification condition generator, then creates the verification condition

$BP \Rightarrow wlp.C.(true, true, false)$

that can be intuitively understood as "Using the facts in the background predicate BP as axioms, we can prove that C always terminates either normally or with an exception, but does not provoke a runtime error". The background predicate is assembled from properties of the Java language, such as the type system, and declarations in the program, and filtered to only include facts that ESC/JAVA judges as relevant for the routine at hand (this is done to make the prover's work easier and avoid the exploration of useless theorem paths).
These verification conditions are then fed to Simplify, ESC/JAVA's theorem prover.
The last stage finally receives all the verification conditions Simplify was unable to prove and converts them into warning messages that are presented to the user.

Let us run through a small example to illustrate how this works in principle. Consider the following java method:

```java
public class Multiplier {
	//@pre a>=0 && b>=0;
	public static int mult(int a, int b)
	{
		int accum = 0;
		//@loop_invariant accum == i * b && i <= a;
		for(int i=0;i < a; i++)
		{
			accum += b;
		}
		//@assert accum == a*b;
		return accum;
	}

	public static void main(String[] args) {
		int m = mult(5,4);
		// prints "5 * 4 == 20"
		System.out.println("5 * 4 == " + m);
	}
}
```

We have to translate this method into a guarded command. The guarded command, with static loop unrolling, would look something like this:

```
var a b in
  assume a >= 0 && b >= 0
    ;
    var accum in
      accum = 0
      ;
      var i in
        i = 0
        ;
        assert accum == i * b && i <= a;
        accum = accum + b;
        i = i + 1;
        assume accum == i * b && i <= a
      end
      ;
      assert accum == a * b
    end
  end
```

The wlp derivation from this code is a purely mechanical task of applying the recursive definition of wlp, resulting in a predicate that can be checked as we explained before. In this case the predicate would look like this(s is the program state):

$\lambda s.0 \le s\ a \longrightarrow 0 \le s\ b \longrightarrow (\text{accum} = i * b)(s[i := 0]) \wedge (\text{accum} = i * b)(s[i := 1]) \longrightarrow 1 < s\ a \longrightarrow (\text{accum} = a * b)(s[i := 1])$

I.e. Given that a and b are non-negative, the theorem prover has to prove that the loop invariant holds in the first iteration of the loop and that the loop invariant holding in the second iteration of the loop, the post-condition (that the result is $a * b$) holds.

If we look closely at the translation of the for-loop, it doesn't really model a loop. It is simply an unrolling of the loop: `loop { invariant J } C end` is translated to
`assert J; C; assert J; assume false;`. (`assume false` means the checking stops)

This also means that the verification condition we just created cannot be proven correct because it doesn't hold. We need to generalize the wlp by removing the fixed assignments caused by the static unrolling:

$\lambda s.0 \leq s\ a \longrightarrow 0 \leq s\ b \longrightarrow (\text{accum} = i * b)(s[i := 0]) \wedge (0 < s\ i \wedge s\ i < a) \longrightarrow (\text{accum} = i * b)s \longrightarrow s\ a < s\ i \longrightarrow (\text{accum} = a * b)s$

This is one of the sources of unsoundness in ESC/JAVA, leading to the possibility that a program that does not generate any warnings (or cautions, where the prover ran out of resources trying to prove a verification condition) may nevertheless contain bugs.

Automatic reasoning about loops is difficult, because it is very hard to generate loop invariants. There are attempts to do this [4]; the version of ESC/JAVA used in the mobius checking framework[3] properly checks loop invariants if they are provided by the user, but cannot derive strong loop invariants on its own.

3 Programmes

In Java, there is a locking mechanism that allows the synchronization of parallel threads. Any object is lockable, as if every object had its own mutex built in. Acquiring and freeing those locks is done by wrapping code in **synchronized** blocks. ESC/JAVA supports the detection of data races and deadlocks in concurrent code. We will illustrate how the detection of data races works.

3.1 Counter.java

```
1  public class Counter {
2          int c = 0;
3
4          public static void main(String[] args)
5                  throws InterruptedException
6          {
7                  Counter c = new Counter();
8
9                  CountRunnable c0 = new CountRunnable(c, 1);
10                 CountRunnable c1 = new CountRunnable(c, -1);
11                 Thread t0 = new Thread(c0);
12                 Thread t1 = new Thread(c1);
13                 t0.start();
14                 t1.start();
15                 t0.join();
16                 t1.join();
17                 synchronized(c) {
18                         System.out.println(""+c.c);
19                 }
20         }
21 }
22
23 class CountRunnable implements Runnable {
24         /*@non_null*/ Counter c;
25         int d;
26
27         public CountRunnable(/*@non_null*/ Counter c, int d) {
28                 this.c = c;
29                 this.d = d;
```

[3] http://www.kindsoftware.com/products/opensource/ESCJava2/

```
30          }
31
32          public void run() {
33                  for (int i=0;i<10000;i++) {
34                                  c.c += d;
35                  }
36          }
37
38  }
```

Counter.java is a variation of one of the basic demonstrations of the danger of race hazards in concurrent programming. The two CountRunnable instances increment and decrement the variable c in the Counter instance. If the operations were executed sequentially, the variable should be 0 when the programme finishes. But due to the data race, it will show a completely different value.a

The `monitored` annotation instructs ESC/JAVA to check an object field for proper synchronization:

```
public class Counter {
        //@monitored
        int c = 0;
```

This creates a proof obligation in line 34 for the `Counter` instance to be locked before it is accessed.

ESC/JAVA then immediately detects the race:

```
Counter.java:34: Warning: Possible race condition (Race)
                                c.c += d;
                                  ^
MARKER Counter.java 34 Warning: Possible race condition (Race)
Associated declaration is "Counter.java", line 3, col 4:
        //@monitored
            ^
```

3.2 Account.java

ESC/JAVA also detects the race in a less contrived example. The race condition in the `Account` class in the following code is a common beginner mistake and was extracted from actual code [6]. Can you spot the data race?

```
1   public class Account {
2           int money;
3
4           public Account(int money)
5           {
6                   this.money = money;
7           }
8
9           public synchronized int GetMoney()
10          {
11                  return money;
12          }
13
14          public synchronized void deposit(int amount)
15          {
16                  this.money += amount;
```

```
17          }
18
19          public synchronized void withdraw(int amount)
20          {
21                  this.money -= amount;
22          }
23
24          public synchronized void transfer(int amount, /*@non_null*/ Account recipient)
25          {
26                  this.money -= amount;
27                  recipient.money += amount;
28          }
29  }
```

If you add the monitored annotation to the money field and run ESC/JAVA, the mistake will be obvious: ESC/JAVA will issue a warning on line 27 about a possible data race.

Fixing the bug is very easy here as well:

```
        public synchronized void transfer(int amount, /*@non_null*/ Account recipient)
        {
                this.money -= amount;
                synchronized{recipient} {
                        recipient.money += amount;
                }
        }
```

References

[1] Flanagan, Cormac, et al. *Extended static checking for Java.* ACM Sigplan Notices. Vol. 37. No. 5. ACM, 2002.

[2] Detlefs, David L. *Extended static checking.* Vol. 159. Compaq, Systems Research Center, 1998.

[3] Leino, K. Rustan M., James B. Saxe, and Raymie Stata. *Checking Java programs via guarded commands.* Formal Techniques for Java Programs, Technical Report 251 (1999): 1999-002.

[4] Janota, Mikolas. *Assertion-based loop invariant generation.* (2007).

[5] Dijkstra, Edsger Wybe, et al. *A discipline of programming*. Vol. 1. Englewood Cliffs: prentice-hall, 1976.

[6] Eytani, Yaniv, et al. *Towards a framework and a benchmark for testing tools for multi-threaded programs.* Concurrency and Computation: Practice and Experience 19.3 (2007): 267-279.