Gordon J. Pace
(joint course design with Christian Colombo)
University of Malta

March 2018

RUNTIME VERIFICATION FROM THEORY TO PRACTICE AND BACK

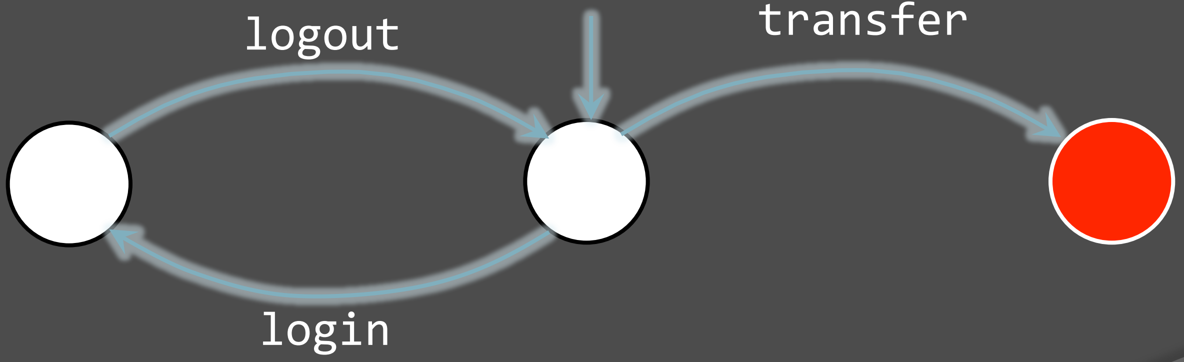
Implicit Monitor States

Implicit States

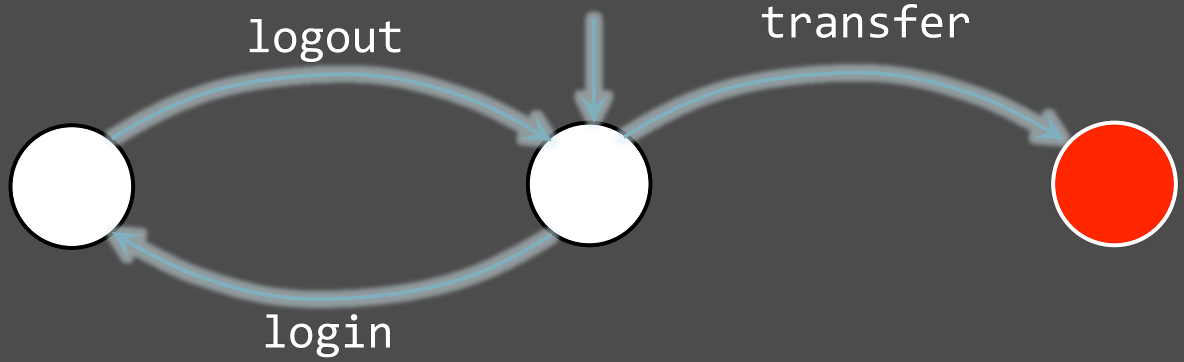
- GCL specifications are generally cleaner than writing AspectJ directly since (i) we need not concern ourselves with certain AOP issues and (ii) we avoid repeated conditional patterns making code more legible.
- However, we still replicate most of the AOP support code.
- In particular, the state of the specification at runtime has to be identically encoded in both approaches.
- Can we implicitly encode (at least part of) the state?

Finite State Automata (1)

- Finite state automata provide us with a way of implicitly encoding part of the monitor state.



Finite State Automata (1)



```
property starting LO {
  LO >>> login >>> LI;
  LI >>> logout >>> LO;
  LO >>> transfer >>> BAD["Cannot
transfer"];
}
```

Finite State Automata (1)

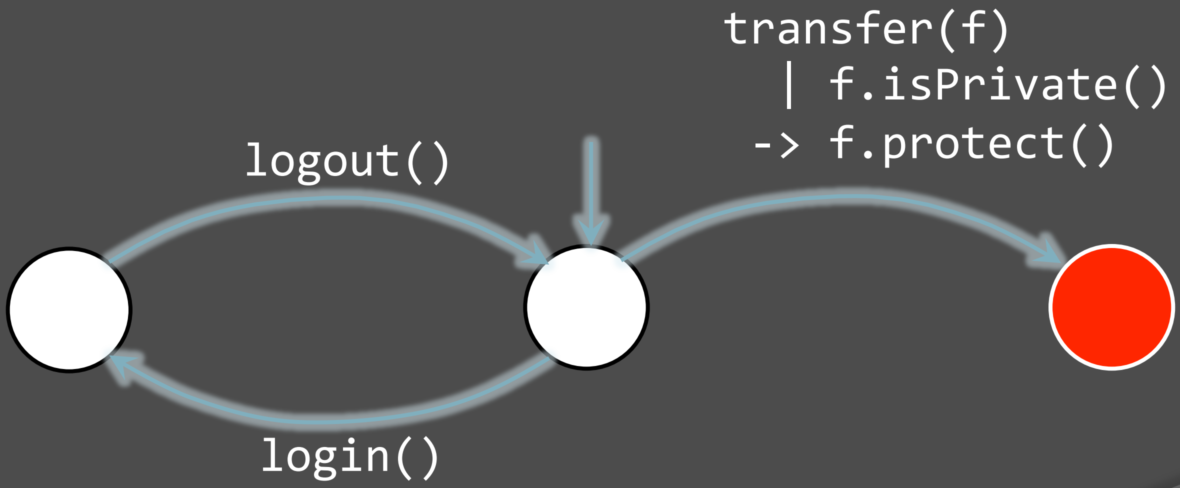
```
S1 >>> event >>> S2
```

becomes

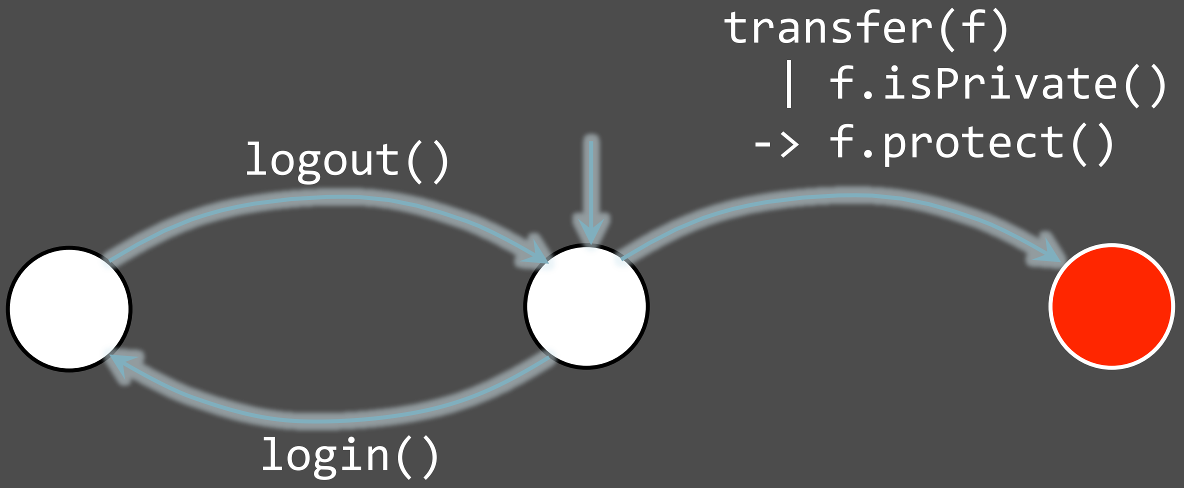
```
before (): call(* event(..)) {
    if (monitor.getState()==S1) {
        monitor.setState(S2);
    }
}
```

Finite State Automata (2)

- Labels can be made more informative:



Finite State Automata (2)



```
property starting LO {
  LO >>> login() >>> LI;
  LI >>> logout() >>> LO;
  LO
    >>> transfer(f) | f.isPrivate() -> f.protect() >>>
  BAD[“Cannot transfer”];
}
```

GCL-to-AspectJ Encoding

```
S1 >>> event | cond -> action >>> S2
```

becomes

```
before (): call(* event(..)) {
     if (state==S1 && cond) {
          state=S2;
          action
     }
}
```

Finite State Automata Encoding

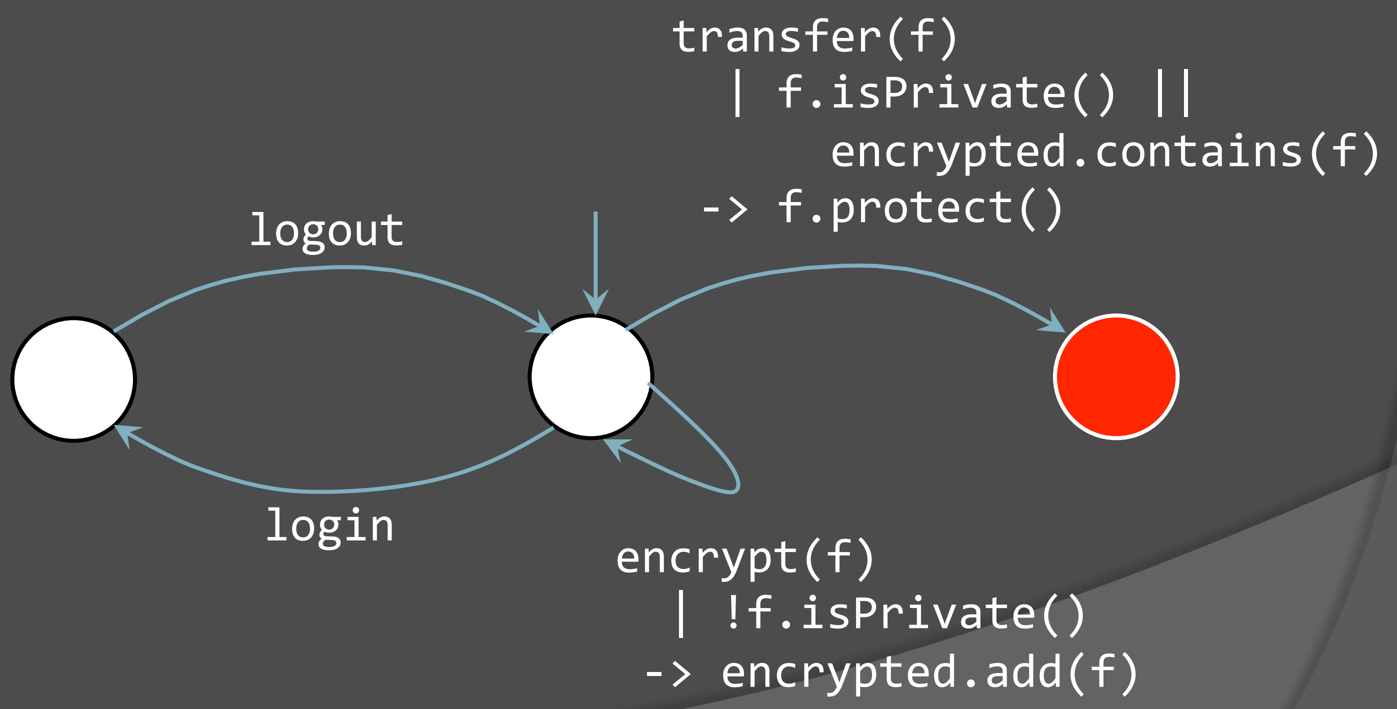
```
S1 >>> event | cond -> action >>> S2
```

becomes

```
before (): call(* event(..)) {
      if (monitor.getState()==S1 && cond) {
            monitor.setState(S2);
            action
      }
}
```

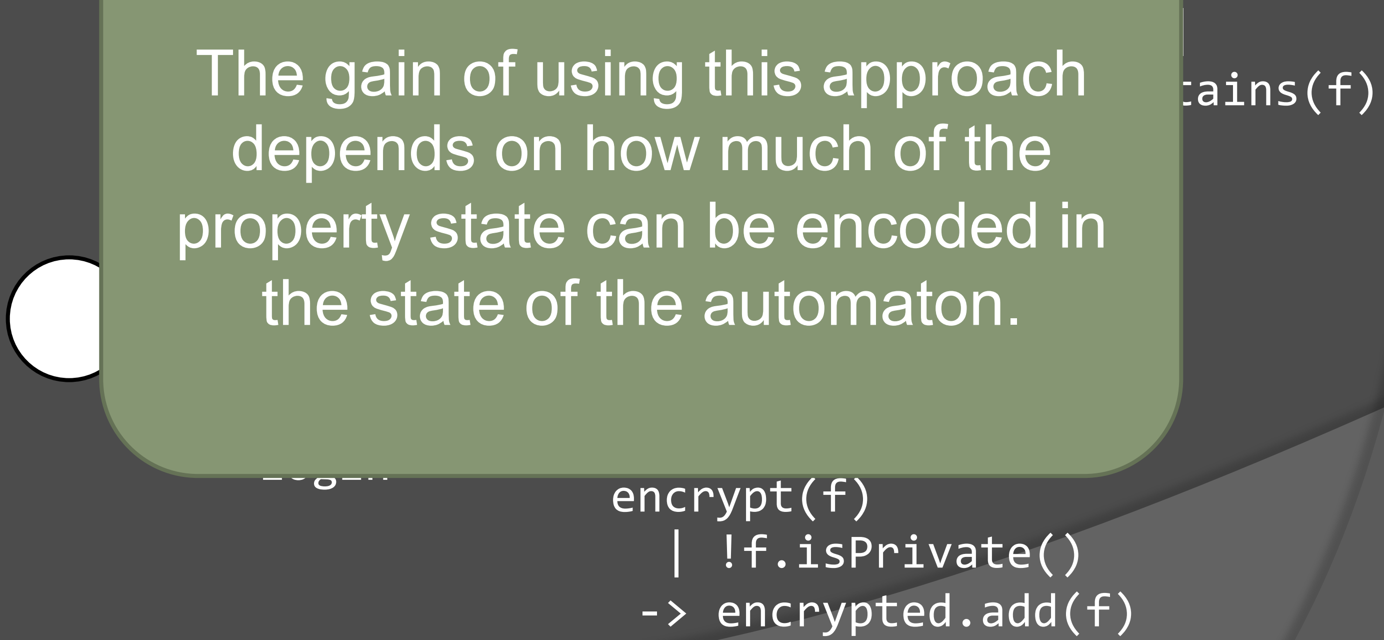
Finite State Automata (2)

- It is worth noting that some of the state may still be explicitly encoded:



Finite State Automata (2)

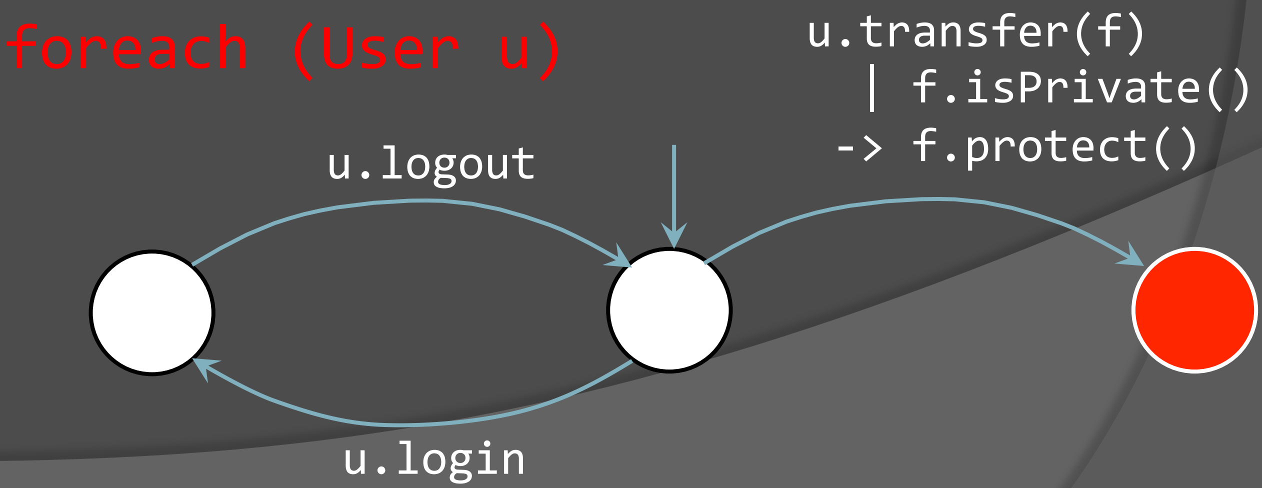
- It is worth noting that some of the state may still be explicitly encoded:



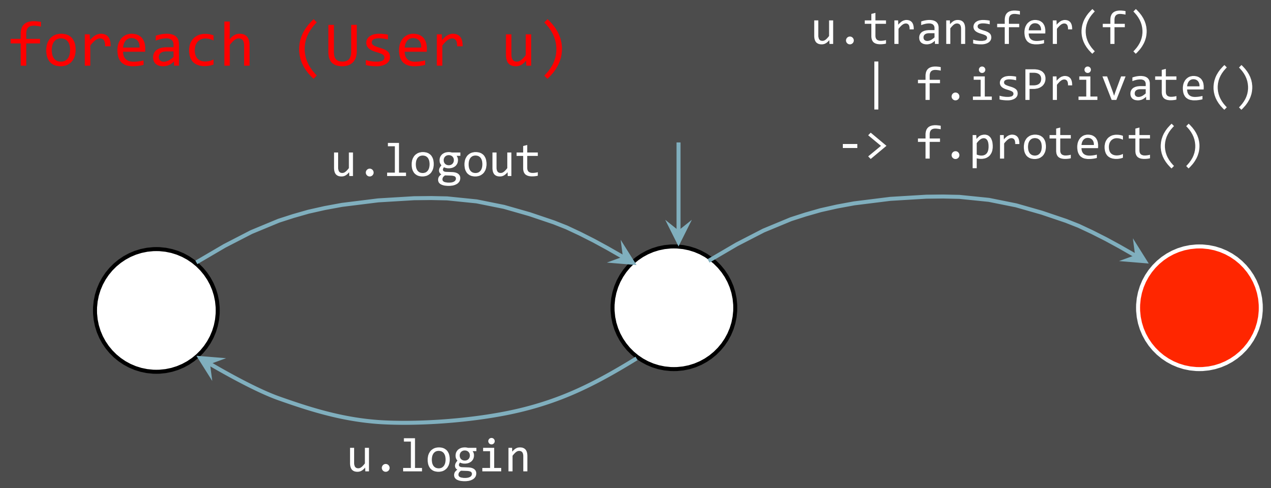
The gain of using this approach depends on how much of the property state can be encoded in the state of the automaton.

Finite State Automata (3)

- Many properties are universally quantified over instances of a class.
- The automaton would have to be replicated for each instance of an object:



Finite State Automata (3)



```
foreach target (User u) {
  LO >>> u.login() >>> LI;
  LI >>> u.logout() >>> LO;
  LO >>> ... >>> BAD[“Cannot transfer”];
}
```

Finite State Automata

```
foreach target (Type t) {
  S1 >>> event | cond -> action >>> S2
  ...
}
```

becomes

```
before (Type t): call(* event(..)) && target(t) {
  if (!monitor.containsTarget(t)) {
    monitor.addTarget(new Monitor());
  }
  if (monitor.getTarget(t).getState()==S1 && cond) {
    monitor.getTarget(t).putState(S2);
    action
  }
}
```

Exercises

1. Add a method `toAspectJ` to the classes `Transition` and `Automaton`, which return AspectJ source code (as a string) implementing the code.
2. Rewrite the specification of FiTS using Automata.
3. Run your translator on the specification to obtain an AspectJ script which you can compile with FiTS.
4. Rerun the scenarios to check that you get the same results as before.

Example – Source

```
VERIFICATIONCODE
public class Verification {
         public static Boolean a_happened;
         public static void fail(String s)
         {        System.out.println(“ERROR: ”+s);
         }
         public static void initialiseVerification()
         {        a_happened = false;
         }}

PRELUDE
package fits;

AUTOMATA
//P1
property starting Start {
  Start >>>
    *.makeGoldUser(..) target (UserInfo u)
      |  !(u.getCountry().equals("Argentina"))
    >>> Bad [P1 violated] }
```

Example – Verification.java

```
public class Verification {
        public static Boolean a_happened;

        public static void initialiseVerification()
        {
                a_happened = false;
        }
}
```

Example – Properties.aj

```
static HashMap<UserInfo, String> monitors504c2683 = null;
static void init504c2683(){ monitors504c2683= new
HashMap<UserInfo, String>(); }

static boolean checkMonitor504c2683(UserInfo target, String
state) {
 if (monitors504c2683.containsKey(target))
  return state.equals(monitors504c2683.get(target));
 else {
  monitors504c2683.put(target, state);
  return state.equals(state); }  }

static boolean setMonitor504c2683(UserInfo target, String
state) {
 if (monitors504c2683.containsKey(target))
 { monitors504c2683.put(target,state);
   return true;  }
 else {
  return false; }  }
```

Example – Properties.aj

```
before (UserInfo u): call(* UserInfo.makeDisabled(..)) && target(u) {
    if ( checkMonitor504c2683(u, "Enabled") && true
        && (setMonitor504c2683(u, "Disabled"))) {{}}
  }
before (UserInfo u): call(* UserInfo.makeActive(..)) && target(u) {
    if ( checkMonitor504c2683(u, "Disabled") && true
        && (setMonitor504c2683(u, "Enabled"))) {{}}
  }
before (UserInfo u): call(* UserInfo.withdrawFrom(..)) && target(u) {
    if ( checkMonitor504c2683(u, "Disabled") && true
        && (setMonitor504c2683(u, "Bad"))) {
                System.out.println("P3 violated");{}}
  }
```

Eclipse and the Automaton Tool

For the sake of this assignment:

1. Import the `MyRVTool` project.
2. Import `FinancialTransactions-04-Automata` – a clean FiTS project, with:
   a. A text file `specification.rules`
   b. A Java class `Verification.java`
   c. An AspectJ file `Properties.aj`
3. Right-clicking and choosing `Properties` on these files will give you their filename with the full path. Copy and paste them into the `Main.java` file in the `MyRVTool` project.
4. The workflow is to: (i) edit the the `MyRVTool` specification; (ii) running the `MyRVTool` project to generate the code in the `FinancialTransactions-04-Automata` project, which (assuming correct syntax is generated); (iii) compile and run the `FinancialTransactions-04-Automata` project.