

MIDlet Navigation Graphs in JML

Wojciech Mostowski and Erik Poll
Digital Security
Radboud University Nijmegen
The Netherlands

<http://www.cs.ru.nl/~{woj,erikpoll}/>

<http://mobius.inria.fr>

Overview

- Problem statement: **Expressing** navigation graphs in JML and **formal verification**
- **MIDlets**: Java programs for mobile phones (MIDP = Mobile Information Device Profile)
- MIDlet security policies: **navigation graphs**
- **JML**: Java Modelling Language for specifying properties of Java source code
- MIDP Java API architecture
- Program verification: **ESC/Java2**, later **KeY**
- Case study: Mobius quiz game midlet

MIDlets

- MIDP: Java for mobile phones
- Comparatively small API:
 - GUI: Screens, user actions (commands)
 - Connectivity: SMS and Internet connections
 - Access to other phone services, e.g. phone book
 - Includes part of the regular desktop Java API
- Full concurrency
- MIDlets small in size, large in quantity
- Security sensitive:
 - Air usage costs caused by midlets
 - May contain sensitive user data

MIDlet Security

- Different levels of trust:
 - Untrusted applet: no air time
 - Trusted applet: controlled air time
 - Fully Trusted applet: unlimited air time
- Midlets digitally signed: certificate defines trust level
- Midlets work in a sandbox:
 - Limited semantics: something is either forbidden or allowed
 - User confusion and annoyance: alert pop-up boxes

MIDlet Security

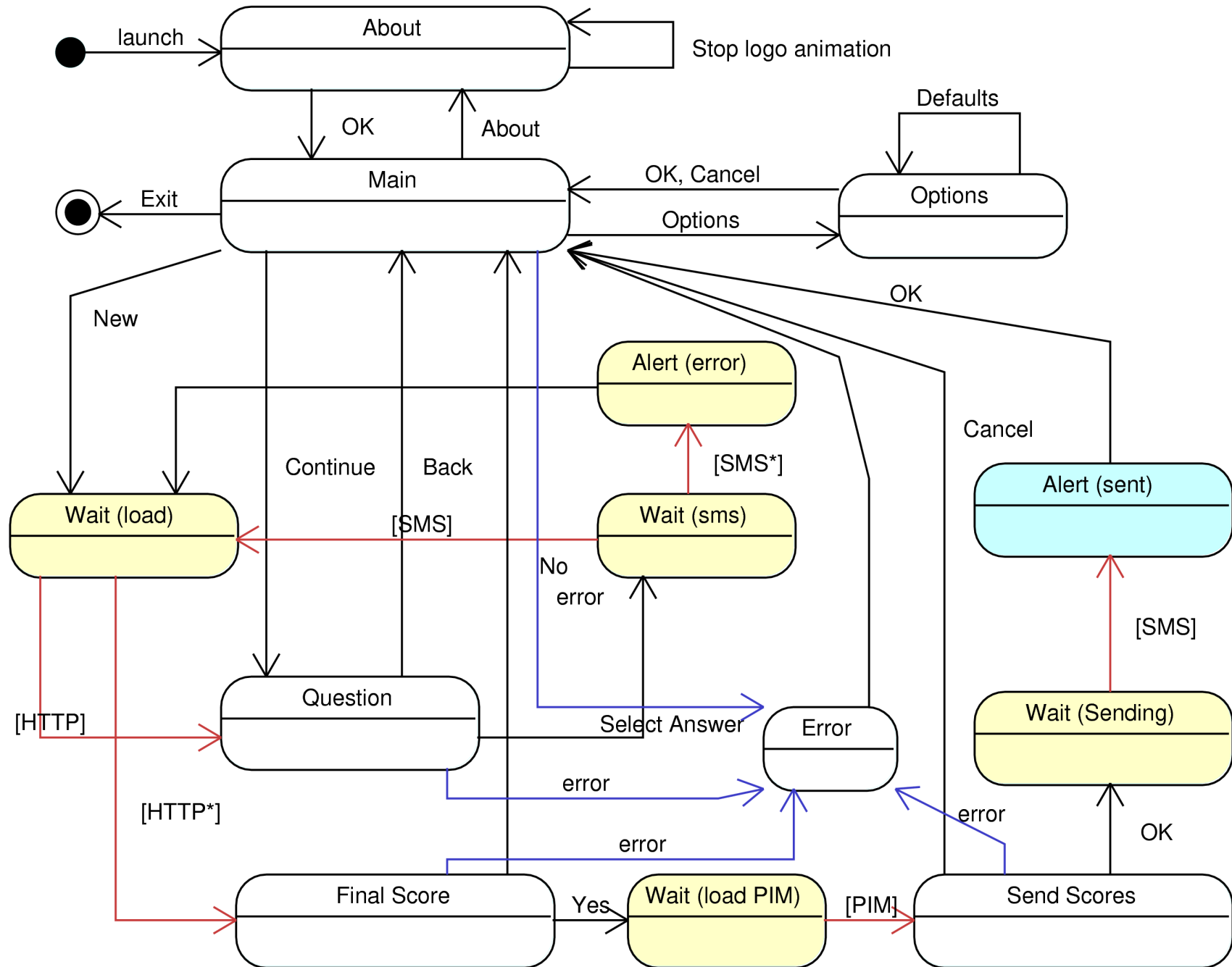
Desired MIDlet properties:

- No hidden functionality (secret screens)
- No **unwanted air time**
- No **premium cost air time**
- Threats taken seriously by telecom companies: high stakes from user (client) perspective

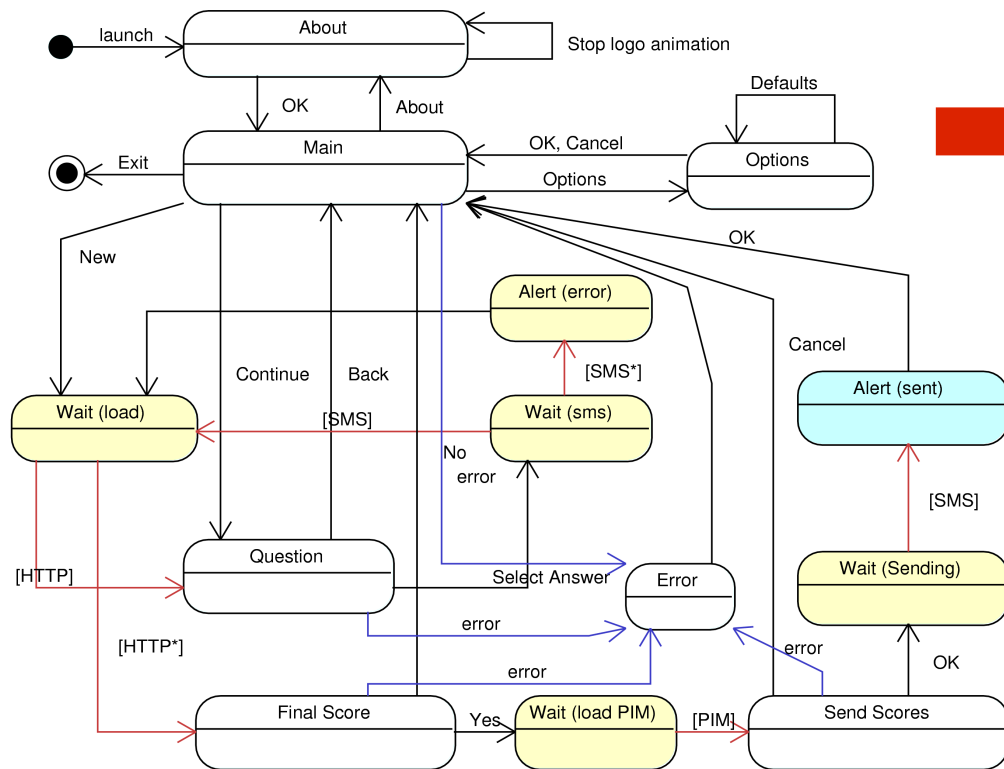
Large numbers:

- Thousands of applications, new releases every few months
- Tens of different hand sets, different application ports
- High testing requirements mandated by the Unified Testing Criteria, e.g. each application needs few minutes of interactive testing
- Practice: **manual certification** (UTC document **91 pages**)

MIDlet Security - Navigation Graph



Navigation Graph to JML



```
class X {
```

```
  //@ invariant ???;
```

```
  //@ requires ???;
```

```
  //@ ensures ???;
```

```
  void methodY(...) {...}
```

```
}
```

For this we need to know the MIDP API structure:

- How the screens are build and displayed
- How user commands are handled
- And then where to hook up our specifications exactly...

Relevant MIDP API: Screens, Commands

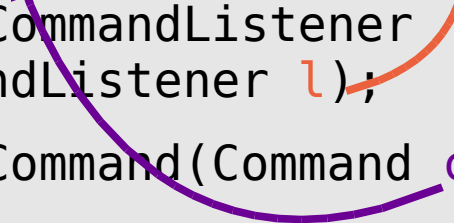
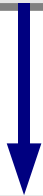
```
abstract class MIDlet {  
}
```

```
class Display {  
    Displayable current;  
    MIDlet midlet;  
    void setCurrent  
        (Displayable d);  
    static Display  
        getDisplay(MIDlet m);  
}
```

```
class Command {}
```

```
abstract class Displayable {  
    CommandListener listener;  
    Set commands;  
    void setCommandListener  
        (CommandListener l);  
    void addCommand(Command c);  
}
```

```
interface CommandListener {  
    void commandAction  
        (Displayable d, Command c);  
}
```



JML API Annotations

```
class Display {  
    // Current screen contents:  
    /*@ non_null @*/ Displayable current;  
  
    // The MIDlet this screen belongs to:  
    /*@ non_null @*/ MIDlet midlet;  
  
    /*@ ensures current == d;  
    /*@ modifies current;  
    void setCurrent (/*@ non_null @*/ Displayable d);  
  
    /*@ ensures \result.midlet == m;  
    /*@ modifies \nothing;  
    static /*@ non_null @*/ Display  
        getDisplay(/*@ non_null @*/ MIDlet m);  
}
```

JML API Annotations

```
class Command {  
    // The Displayable this command is attached to:  
    //@ ghost Displayable displayable;  
    ...  
}
```

```
abstract class Displayable {
```

```
    CommandListener listener;  
    //@ ensures listener == l; modifies listener;  
    void setCommandListener(CommandListener l);
```

```
    //@ ensures c.displayable == this;  
    //@ modifies c.displayable;  
    void addCommand(/*@ non_null */ Command c);
```

```
}
```

Specific MIDlet Specifications

- Reflect the graph structure in JML
 - Limit the **set of possible screens** (states)
 - Limit the **set of possible commands for each screen**
 - Describe **screen transitions**
- Difficulties:
 - No single point of entry into the midlet, commands handled by a dispatcher thread
 - Explicit concurrency
 - Direct access to screen object references
- Result:
 - No central formula describing the graph - specs spread over many MIDlet classes
 - Some cheating not to deploy full concurrent reasoning

Possible States

```
class MyMIDlet {  
  
    /*@ non_null @*/ Display display;  
    /*@ non_null @*/ MyMainScreen mainScreen;  
    /*@ non_null @*/ MyAbout aboutScreen;  
    //@ invariant display.current == mainScreen.mainGuiElement ||  
                 display.current == aboutScreen.mainGuiElement;  
  
    MyMIDlet methods  
  
}
```

Possible Transitions

```
class MyMainScreen implements CommandListener {  
  
    /*@ non_null @*/ Command cmdExit = new Command("Exit");  
    /*@ non_null @*/ Command cmdAbout = new Command("About");  
    /*@ non_null @*/ Displayable mainGuiElement = new TextField();  
    //@ invariant cmdExit.displayable == mainGuiElement;  
    //@ invariant cmdAbout.displayable == mainGuiElement;  
  
    ...  
  
}
```

State Transitions

```
class MyMainScreen implements CommandListener {

    Command cmdExit; Command cmdAbout; Displayable mainGuiElement;
    //@ invariant mainGuiElement.listener == this;

    //@ requires c.displayable == d;
    //@ requires midlet.display.current == d;
    //@ requires d == mainGuiElement;
    //@ requires c == cmdExit || c == cmdAbout;
    //@ ensures c == cmdAbout ==>
        midlet.display.current == midlet.aboutScreen.mainGuiElement;
    void commandAction(Command c, Displayable d) {
        ...
    }
    ...
}
```

A problem - Reference Accessibility

```
class MyMainScreen implements CommandListener {  
  
    Command cmdOptions; Displayable mainGuiElement;  
  
    //@ ensures c == cmdOptions ==>  
        midlet.display.current == options.guiElement;  
    void commandAction(Command c, Displayable d) {  
        if (c == cmdOptions) {  
            OptionsScreen options = new OptionsScreen(midlet);  
            options.show();  
        }  
        ...  
    }  
}
```

A problem - Solution

(Yes, I know, in KeY this is not a problem at the moment, but Wolfgang is working on it ;))

```
class OptionsScreen implements CommandListener {
```

```
    Displayable guiElement;
```

```
    //@ static ghost Displayable staticGuiElement;
```

```
    void show() {
```

```
        guiElement = new TextBox();
```

```
        //@ set OptionsScreen.staticGuiElement = guiElement;
```

```
        midlet.getDisplay().setCurrent(guiElement);
```

```
    }
```

```
    //@ ensures c == cmdOptions ==>
```

```
        midlet.display.current == Options.staticGuiElement;
```

But Then... Invariant Semantics

```
class MyMIDlet {  
    //@ invariant display.current == mainScreen.mainGuiElement ||  
                 display.current == OptionsScreen.staticGuiElement;  
}
```

```
class OptionsScreen implements CommandListener {  
  
    void show() {  
        guiElement = new TextBox();  
        //@ set OptionsScreen.staticGuiElement = guiElement;  
        midlet.getDisplay().setCurrent(guiElement);  
    }  
}
```

Invariant Semantics - Solution

- Spec# solves this problem with unpacking / packing of objects
- General idea: switching off and on invariant checking
- Here can be done with a simple boolean predicate:

```
class MyMIDlet {  
    //@ invariant Display.stableState ==>  
        display.current == mainScreen.mainGUIElement ...;
```

```
void show() {  
    //@ set Display.stableState = false;  
    guiElement = new TextBox();  
    //@ set OptionsScreen.staticGuiElement = guiElement;  
    midlet.getDisplay().setCurrent(guiElement);  
    // setCurrent sets stableState back to true  
}
```

Sensitive API Calls

- Limit sensitive API calls
- General idea:
 - Count calls in the API specifications with a ghost field
 - Limit the number of calls in the MIDlet specification

```
class MessageConnection {  
    //@ static ghost int count;  
    //@ ensures count == \old(count) + 1; modifies count;  
    void send(/*@ non_null */ Message m);  
}
```

```
class MyMainScreen implements CommandListener {  
    //@ ensures MessageConnection.count ==  
        \old(MessageConnection.count) + (c == cmdAbout ? 1 : 0);  
    void commandAction(Command c, Displayable d) { ... }  
    ...  
}
```

Other Things

Open issues:

- Concurrency (solved in a dirty way)
- Formal correspondence: Graph \equiv ? JML specs
- Singleton objects enforced by the platform
- Termination

Not discussed:

- The actual case study - many nasty details
- Object visibility and ownership, e.g. the case study has circular object dependencies
- Possible problems with KeY, tool interoperability

Verification with ESC/Java2

- Mobius case study (almost) fully verified
- Serious competitor of KeY
- Fast, automatic
- Problem tracing is difficult
- “Strict” correctness semantics gives a headache sometimes
- Accurate, judging from the amount of headache it gives
- Future, current JML community efforts

Conclusions

- **MIDlet navigation graph** - security policy deemed **important** and also **problematic** by the industry
- Relatively **easy** for formal verification tools
- However, graph representation in JML not that straightforward
 - **Specification engineering**
 - Open question: should such policies be expressed with navigation graphs or something more formal
- Different tools give different **level of confidence** - trade off on tool complexity
- Acceptable confidence can be achieved with **ESC/Java2**
- Hopefully will be better off with KeY

The End

Questions?