Converting Midlet Navigation Graphs into JML

Master's thesis, thesis number 563

Daan de Jong

July 1, 2007

Radboud University Nijmegen

Supervisor: Dr. E. Poll

Radboud Universiteit Nijmegen

Abstract

As a result of the fast growing mobile (application) market the need for reliable and trusted mobile applications is growing. Telecom operators need to guarantee that software can be trusted and behaves according to its specification. In particular investigation is done of the current informal use of flow charts, also known as midlet navigation graphs, as a basis for formal specification that enables rigorous testing and program verification. Midlets are the most common applications running on a mobile device.

This research mostly describes the basis for a solution for modeling software specifications in such a way that it can be checked using a formal verification tool. The midlets, written in Java, are annotated with JML and verified with the tool ESC/Java2. This annotation is based on the formal specification described in this paper.

A basic definition of a formalism for midlet navigation graphs is given for modeling and verification of a midlet. A midlet navigation graph is actually a state diagram which represents the basic behavior of a midlet. The possible displays, display changes, command events and simple policies are defined in the midlet navigation graphs. Also the translation of the building blocks of midlet navigation graphs to JML is given.

The result of this research is only a first step. But the most important conclusion is that it is possible to define a standard specification to verify a midlet. First a implementation of a more complicated midlet must be produced to ensure the practical value of the definition. After that the definition of a formalism for midlet navigation graphs must be extended with more advanced aspects of a midlet like for example multi-threading.

Contents

1 Introduction

This introduction describes the reason why midlet navigation graphs would be interesting to define and what the effort is of translating them to JML. The term *midlet navigation graph* (MNG) is found by Cregut [Cre06] during his research on verification of midlets.

1.1 Background information

Mobile devices are developing fast and most of them can already be marked as full grown computers. Together with this development also mobile applications on cell phones and PDAs, are becoming more advanced. Most applications (also known as midlets) use Java source code based on MIDP 1.0 or the more recent MIDP 2.0. 3 globally describes a basic midlet and its properties. As a result of this, mobile applications are the next target for misuse of their functionalities. Some viruses and malicious mobile applications already are circulating and their number is growing. Figure 1 shows a cell phone infected with the Cabir virus. This virus replicates over blue tooth connections.

Figure 1: Mobile device infected with the Cabir virus

If we look at the basis of a virus or malware than we can conclude that most wrong behavior is caused by unexpected behavior of a mobile application. In a lot of malware applications unexpected windows are presented to the users. Another example of typical unexpected behavior is a unwanted network connection during the usage of a application. The growing use of internet connections by mobile devices is another growing problem. Most applications running on mobile devices are downloaded from the Internet. These applications are running in a sandbox where only a minimal number of functionalities of the mobile

device can be used. The sandbox restricts the permissions of the applications. These are quite harmless applications and only a bug in the operating system of the mobile device can be harmful. Mobile applications signed by the telecom operator are granted with all rights. This way these applications can use extra functionality, e.g. sending of SMS, to profit from the functionality of the applications. Telecom operators buy these applications from external companies and these applications must be developed according to the specification of the telecom operator. It is obvious that this source code cannot be trusted.

1.2 Specification

So let's say that the telecom operator buys a mobile application from a company. Typically the telecom provider specifies the functionality of the application in some sort of format and will give this to the developers. This specification mostly only describes certain (detailed) aspects of the application like security requirements or specific functionalities. Basic functionality of a midlet, such as the midlet navigation, is not embedded in these specification. Informal specification (text) is mostly used for the basic functionality of a midlet. These basic functionality includes the possible displays that are presented on the screen of the mobile device. The possible events that can be used to navigate through the screens and network connections that occur during the lifetime of a midlet.

The developers will create an application which includes this functionality. The telecom provider will check if this is the case to see if the development of the software is done well. Another possible approach could be that the telecom operator buys a finished mobile application from a company. In this case the company should provide a specification with the software to explain to the mobile operator how the software works. In both cases the specification, in any possible format, will be match against the functionalities of the developed mobile application. This match is currently been done by hand, just try as many possible scenarios as practically possible in the mobile application and see if it really works as expected. This is called testing. Testing of the application and the source code is done according to the international testing standard [tes06]. These tests are rather time consuming and the result is not satisfying.

1.3 Testing and verifying

It would be much better to match the given specification against the source code rather then just use testing as earlier mentioned. By using the source code not only can be checked if the application has the right functionalities but also if the application does not have hidden functionality which are not expected. There is a great need for a way to check a given source code against a given specification in such a way that the source code meets its specification and can be marked as trusted. The ultimate goal would be to formally verify the midlet that it meets the (formal) specification. The format of the (formal) specification and the ways to verify the midlet is yet not clear and is researched.

One of the tools for verifying a midlet is JML in combination with ESC/-Java2. JML annotations can be embedded in the midlets. These annotations form the specification for the midlets source code. The midlet embedded with the JML is the input for the tool ESC/Java2. ESC/Java2 can verify wether the midlets source code meets the specification defined by JML. Section 4 describes the usage of JML and ESC/Java2 in a more detailed way.

1.4 Outline

In chapter 2 midlets are defined and a small tutorial is provided for building a basic midlet which contains most of the common items. Chapter 3 is targeted on verification of the source code. The language JML is describe together with its usage of ESC/Java2. And how it is possible to formally verify a midlet with these techniques. Chapter 4 describes how to specify midlets using midlet navigation graphs. The conclusions from this research are defined in chapter 5. Finally some interesting further research topics are defined in chapter 6.

2 Research objectives

This chapter defines the problem definition, research questions and additional context information.

2.1 Problem definition

Checking if a given Java program meets its specification can be done by testing. Producing a formal specification (in JML) is most difficult and time consuming. A tool that checks if the given source code meets its specification will guarantee trusted source code. The flow of the target situation is shown in Figure 2. All knowledge of the existing research papers is embedded into this flow chart.

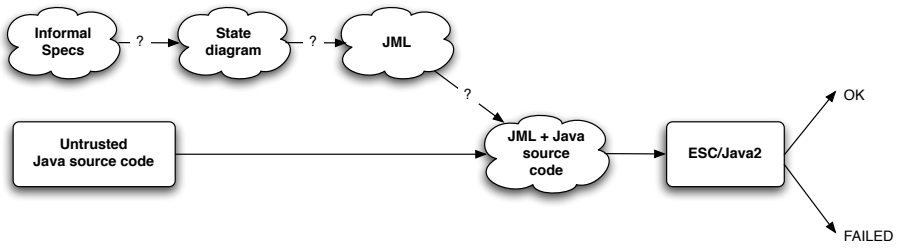

Figure 2: Idealized proposed solution

At the start of the flow chart there are the informal specs of the mobile applications. The specifications are not yet in a standard format and are informal represented as an image or in English text. Also at the start of the flow is the Java source code which can be labeled as untrusted. The specification must be modeled as a formal state diagram. This way it can be converted into JML. The next step is to convert the modeled specification into JML. This JML can then be embedded in the given Java source code. The external tool ESC/Java2 can formally check if the input source code meet the given specification. The output will simply be OK if the source code meets the specification and FAILED if not so. The focus of this research will be on how the informal specs can be expressed in a formal state diagram that can be translated to JML.

2.2 Previous research

The department Security of Systems in Nijmegen is specialized in testing and formal verification of (mobile) Java applications. A lot of research has already been done which has resulted in some interesting papers. Because the department is involved in an European research team some interesting papers are provided by other universities and companies joining this team. The most important papers are described below and related to position this research.

Worldwide telecom operators have joined in an organization to set up a standard for testing of mobile Java-based applications. [tes06] describes the procedures for testing a mobile application using MIDP 1.0 and MIDP 2.0. One of the most concrete parts of this paper is the modeling of the properties of the mobile application in a flow chart. These flow charts are still informal and are described in an image. [HO03] describes a prototype to generate JML and Java skeleton code from an UML model and verifying it with ESC/Java2. Some problems with this model are the fact that is based on UML 1.3 where UML 2.0 is already available. This prototype does not check the UML against given Java source code but it generates correct JML and Java source code. To confirm the usefulness of the previous prototype [HOP04] describes the implementation of a security protocol. This article gives an example of a security protocol and the generation of the skeleton Java source code. The used JML is not automatically generated from the specification but is done by hand. [Cre06] describes a more formal way of defining finite state machines (FSM) as a model for the specification of a mobile application. It mainly describes the security risks which can occur. This definition is far from finished and needs a lot more research for a complete formal specification of these FSMs.

2.3 Scope

As a result of discussions with the supervisor and the currently available papers the scope and side conditions are defined. The most important conditions are:

- *ESC/Java2 is used for formal verification of JML*
  ESC/Java2 is a tool which has proven itself during practical research. Therefor this tool will be seen as trusted.
- *JML is used for the formal specifications in the Java code*
  JML is simple to use because its syntax is in Java style and therefor known to Java developers. JML in combination with ESC/Java2 has been proven a useful couple.
- *No Java source code will be generated*
  In [tes06] Java skeleton source code was generated from the specifications. Because in most practical situations Java source code is delivered from external companies the research will mainly target on given source code.

2.4 Research questions

The main research question:

- **"Can a given Java midlet and a formal specification of its (security) requirements be converted to a JML annotated Java application which correctness can be verified by ESC/Java2?"**

The sub-questions raised from the main research question:

- **"Which formal format of state diagrams can be used to express the specification?"**
  How can specifications be converted to state diagrams? What is the definition of a state in the model and what means a state change in the model?
- **"How can the formal specification be converted to JML in the Java source code?"**
  The states of the formal specification must be connected to JML and the source code in such a way that code behavior is analyzed in each state or state change.
- **"How can the the generated JML be integrated into the given Java source code?"**
  Is the structure of the specification dependable of the method names and variable names or any other type in the Java source code?

3 Midlets

A midlet is an application that conforms to the MIDP standard. Mostly a midlet is written in Java and are developed for embedded devices (e.g. cell phones, PDAs) which are running the J2ME virtual machine. Many of the existing midlets are games and applications running on cell phones. This chapter gives a global description of the structure of midlets and a basic example how to create a midlet. This basic midlet contains output to the display, input from the user and network connections. Also the life cycle of a midlet is described.

3.1 J2ME and MIDP

J2ME (Java 2 Micro Edition) combines a small version of the Java Virtual Machines (JVM) and a set of Java APIs to develop applications for mobile devices. Mobile devices do not have as much resources and memory available as normal computers. Therefor it is not possible to use the traditional JVM but a JVM with some resource constraints. So J2ME combines a resource-constrained JVM and a set of API's for the development of mobile applications.

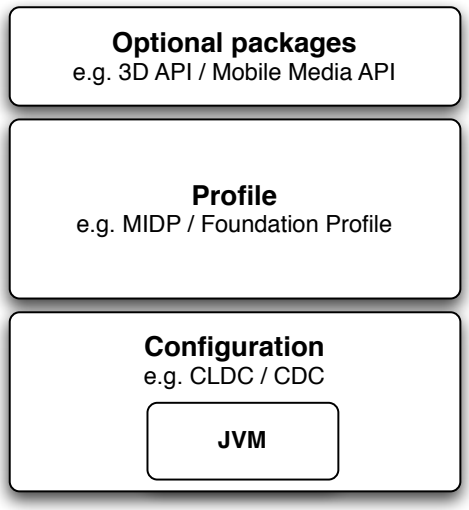

Figure 3: The J2ME framework

There are three parts within J2ME: the configuration, the profile and the optional packages. Figure 3 shows the positioning of these parts within the J2ME framework. The configuration part contains the JVM. Besides the JVM there are some extra packages in the configuration. The most common are the "Connected Device Configuration (CDC)" and the "Connected Limited Device Configuration (CLDC)". The CDC is most used in the bigger devices like PDAs and communicators. These devices have a standard of minimal 32 bits processor and 2 Mb memory. The CLDC is most used in the smaller device like

cell phones and small PDAs with only 128 to 512 KB memory. Functionalities which use a lot of processor power are removed from the JVM such as floating points and finalization. The second level in the J2ME framework are the profiles. These profiles define the APIs needed to write applications for a specific group of J2ME devices. The CLDC configuration is extended with the MID Profile (MIDP). This means that this configuration is among other things extended with an user interface (*javax.microedition.lcdui*), communication between application and the environment (*javax.microedition.midlet*) and data persistence (*javax.microedition.rms*). The CDC configuration cannot use the MID Profile and uses its own profiles like the Foundation Profile and the Personal Profile. On top of the J2ME framework there are the optional packages. These packages can be used for extra functionalities that are not common to use within a midlet.

3.2 A basic midlet

For developing a midlet the J2ME wireless toolkit is available. The toolkit needs the Java Standard Development Kit (SDK) and it contains MIDP, CLDC, the optional packages and the ability to sign your midlets for authentication on a mobile device. Also the wireless toolkit contains a emulator to test the midlet without using a real mobile device.
Implementing a basic midlet starts with extending the abstract class *javax.microedition.MIDlet*. The constructor takes care of the generation of the midlet object. The methods *startApp*, *pauseApp* and *destroyApp*(*booleanunconditional*) must be implemented in the basic midlet. Listing 1 shows a code example for a basic midlet. These methods are regulated by the Application Manager. The Application Manager is part of the mobile device and can start, stop and pause midlets and uses a queue to schedule all running midlets on a mobile device.

Listing 1: A basic midlet

```
import javax.microedition.midlet.*;

public class MyMidlet extends MIDlet
{
  public myMidlet() {}  // constructor

  public void startApp() {}

  public void pauseApp() {}

  public void destroyApp(boolean unconditional) {}
}
```

3.3 Midlet lifecycle

Figure 4 shows the most common version of the midlet life cycle.

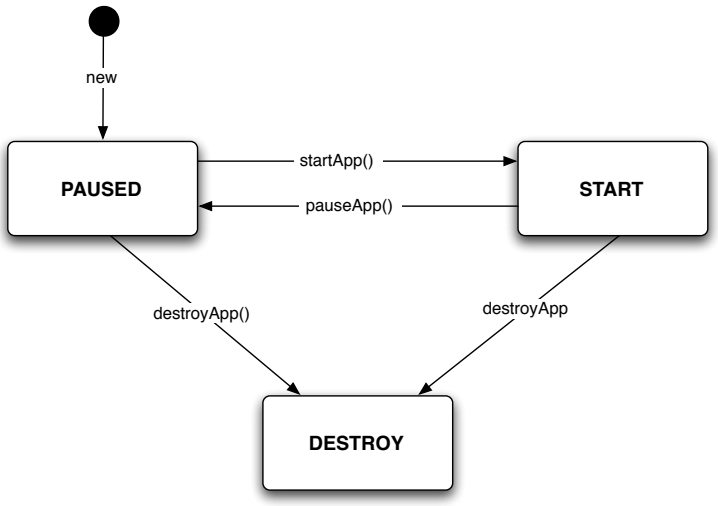

Figure 4: Midlet life cycle

When the midlet is created by calling the constructor a new midlet object is created. The initial state of this midlet is Paused. In the Paused state the midlet is in a static condition and uses as less resources as possible. The midlet can enter the Active state by calling the *startApp*() method and return to the Paused status by calling *pauseApp*(). By calling the *destroyApp*() method the midlet can be destroyed and all resources will be released.

The midlet life cycle must be implemented by the application that extends the Midlet class. The methods *startApp*(), *pauseApp*() and *destroyApp*(*booleanunconditional*) are all abstract methods. The actual implementation of these methods must be done by the developer of the application. So it is the responsibility of the implementation to correctly implement the life cycle. The states of a Midlet object are tracked and traced by an object called the $MidletState$. This object yields the current state of the midlet and tracks all state changes.

The application manager is part of the device software operating system that manages all running midlets on that device. It maintains the state of the midlet and directs the midlet through state changes. It can start, pause and destroy a midlet and he administrates all active midlets on the mobile device using a queue. The application manager is the one that must call the methods *startApp*(), *pauseApp*() and *destroyApp*(*booleanunconditional*). The midlet itself can call these methods but this is not the normal procedure. If a midlet itself wants to start, stop or pause, it can call the methods $notifyPaused()$, $notifyDestroyed()$ and $resumeRequest()$. These methods are defined in the class $Midlet$ which is extended by the implementation of the midlet. The methods $resumeRequest()$ signals the application manager that the midlet wants to enter the active state. The application manager itself can decide when and if the midlet can enter the active state. The application manager will call the

startApp() method to let the midlet enter the active state. So the method *ResumeRequest*() itself will not cause a state change.

3.4 Using the display

Figure 5 shows the class diagram for displaying on the screen. The *Display* object can allow a *Displayable* object to print on the screen. The *Displayable* class is an abstract class and can use a *Canvas* object to print on the screen or it can use an subclass of the interface *Screen*.

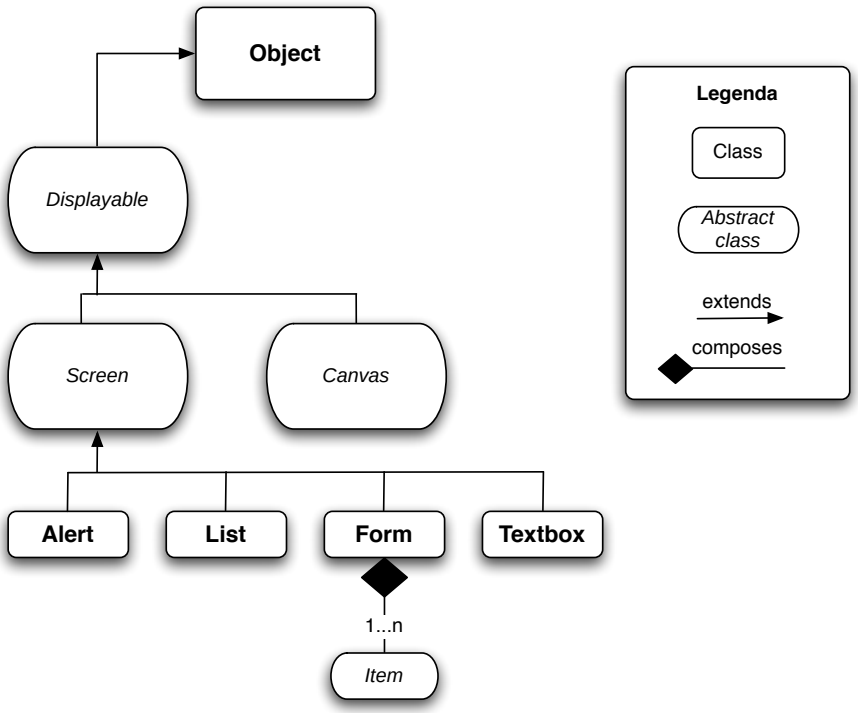

Figure 5: Class diagram for displaying in a midlet

So in total there are five different objects that can be printed on the screen:

- **Canvas**
  A canvas class provides a blank screen on which a midlet can draw. Using a canvas it is possible to draw or print text on the screen.
- **Alert**
  An alert is an object on the screen that contains text and an image. Its goals is to inform users about errors or exceptional behavior of the midlet.
- **Form**
  A form is an object on the screen that contains a mix of items. For example

text fields, images, date fields, gauges and choice groups. It is possible to place any subclass item of the Item class on the Form object.

- **List**
  A list is an object on the screen that contains selectable choices. The user can choose one of the elements in the list as input for the midlet.

- **Textbox**
  A textbox is an object on the screen that allows users to enter and edit text. Textboxes are most used in a midlet if the midlet needs some kind of input like a name or an account number.

For using the screen of a midlet the package *javax.microedition.lcdui* is needed. The user interface on a midlet differs from the user interface of a normal Java application. A Java application uses AWT and Swing for its user interface but these are developed for desktop applications and are too big for small devices. So the midlet uses a user interface defined in the MID Profile. There can only be one type of the five possible types on the screen visible at the time. The placement of the object on the screen is managed by the mobile device itself.

Listing 2: Displaying on the screen

```
import javax.microedition.midlet.*;
import javax.microedition.lcdui.*;

public class MyMidlet extends MIDlet
{
  private Form helloForm;
  private Display myDisplay;

  public myMidlet() {}

  public void startApp() {
    // Get reference to the display
    myDisplay = Display.getDisplay(this);

    // Display the helloForm
    myDisplay.setCurrent(helloForm);
  }

  public void pauseApp() {}

  public void destroyApp(boolean unconditional) {}
}
```

Listing 2 shows a code example from a midlet which displays a form on the screen. The class uses two objects. A object of the type *Form* is created which represents a *Form* object to be displayed on the screen. Also a object from the type *Display* is created. This object takes care of the actual printing on the screen. It holds a reference to the current display and can change the display. Whenever the *startApp* method is called a reference to the display

is generated and the object *helloForm* is put onto the screen. The method *setCurrent*(*Displayable nextDisplayable*) is responsible for putting the *Form* object on the screen.

3.5 Catching events

A user is able to interact with midlets. The interaction takes place using events. These events are created by the user and the midlet can handle the events. The events are handled by the midlet using callbacks. There are three types of events that can occur:

- **Command events**
  The *commandAction* method can be used to catch command events coming from the user. Command buttons are for example the "OK" and "CANCEL" buttons on the mobile device. These command events are specially for navigating through a midlet. Table 1 shows the eight possible types of command events that can be used. These command events are mostly at the top of the keyboard of the mobile device just above the numbers.

Type	Description
BACK	Request to move to the previous screen
CANCEL	Request to cancel an operation
EXIT	Request to exit the MIDlet
HELP	Request to display help information
ITEM	Request to map the *Command* to an "item"
OK	Specify positive acknowledgement from a user
SCREEN	Apply to the screen as a whole, such as Save
STOP	Request to stop an operation

Table 1: Possible command event types

- **Keyboard events**
  The canvas class can handle keyboard events from the user. Possible keyboard events could be the controls for a game or the editing of text by a user. The *Canvas* class provides three methods to handle keyboard events: *keyPressed*(*intkeyCode*) (called when a key is pressed), *keyReleased*(*intkeyCode*) (called when a key is released) and *keyRepeated*(*intkeyCode*) (called when the user holds the key for a longer period). All these methods have one parameter, the unicode value of the key. In the *Canvas* class the values are defined as constants. For example the constant *KEY_NUM0* is the number 0 on the keyboard of the mobile device.

- **Pointer events**
  Pointer events are not always present in every mobile device. Example of pointer events are a trackball or a touch screen. Just like the keyboard events the *Canvas* class provides three methods to handle pointer

events: $pointerPressed(intx, inty)$ (called when a pointer is pressed), $pointerReleased(intx, inty)$ (called when a pointer is released) and $pointerDragged(intx, inty)$ (called when the user drags the pointer on the display). These methods have two parameters, the x and the y position of the pointer when the pointer event is occurring.

The *commandAction* method is a abstract method of the interface *commandListener* shown in Listing 3. This interface works like a daemon and calls the *commandAction* method whenever the user pushes a command button. The *commandListener* interface will provide the instance to the *addCommand* method on a *Displayable* object to receive events on the screen. The *Displayable* object is the object that represents the display on the screen of the mobile device. The *commandAction* method must be implemented in any class which implements the interface
commandListener. The method *commandAction* is called whenever a command event c occurs on display d.

Listing 3: CommandListener interface

```
package javax.microedition.lcdui;
public interface CommandListener
{
        public void commandAction(Command c, Displayable d);
}
```

A *Command* object "exitCommand" is created in the midlet which represents the "EXIT" button on the mobile device. In the code example the *if* statement is used to catch the right command event. If the command event is of type "Exit" then the *commandAction* method is called with the command event object as parameter. The *if* statement will be executed and the midlet will be destroyed (will enter the Destroy state).

Listing 4 shows an implementation of the *commandAction* method.

Listing 4: Implementing command events

```
import javax.microedition.midlet.*;
import javax.microedition.lcdui.*;

public class MyMidlet extends MIDlet
                      implements CommandListener
{
  private Form helloForm;
  private Display myDisplay;
  private Command exitCommand;

  public myMidlet() {
    exitCommand = new Command("Exit", Command.EXIT, 1);
    helloForm = new Form("example");
    helloForm.addCommand(exitCommand);
    helloForm.setCommandListener(this);
  }

  public void startApp() {
    myDisplay = Display.getDisplay(this);
    myDisplay.setCurrent(helloForm);
  }

  public void commandAction(Command c, Displayable s) {
    if (c == exitCommand) {
      destroyApp(false);
      notifyDestroyed();
    }
  }

  public void pauseApp() {}

  public void destroyApp(boolean unconditional) {}
}
```

3.6 Implementing network connections

Important functionality of a midlet is its ability to make network connections. Network connections are needed for example for sending an SMS or downloading data from the internet. Figure 6 show the class diagram for network connections in midlets. All classes are interfaces where the *Connection* class is the most basic interface connection. This interface only provides methods for opening and closing connections. All other interfaces provide specific methods for their type of connection. For example the interface *inputConnection* provides an object from which data can be read. All network connections within a midlet are created using the *open* method of the *Connector* class. If the connection is successfully opened then this method will return an object that is an implementation of one of the connection interfaces. The *open* method has one parameter a *string* that consists of three parts: the protocol, target and parameters.

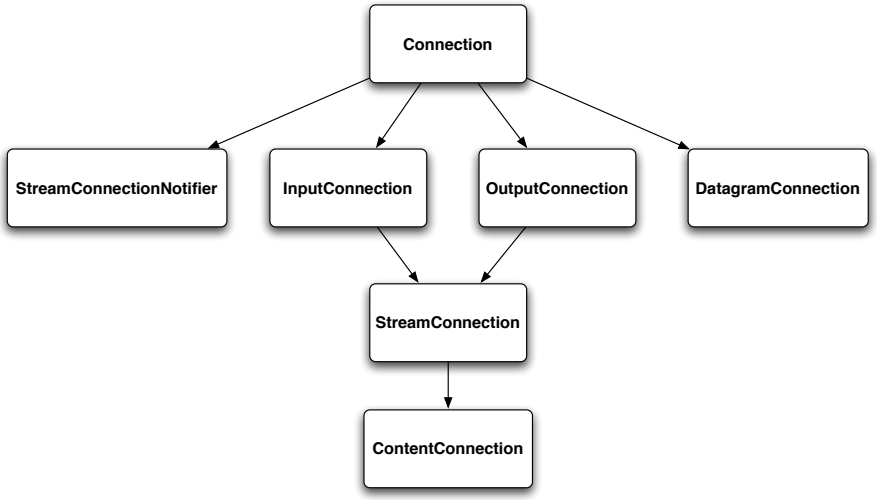

Figure 6: Class diagram for network connections in a midlet

Type	**Description**
file	File input/output
comm	Serial port communication
socket	TCP/IP socket communication
datagram	Datagram communication
http	Web server communication

Table 2: Possible connection types

The parameter has the form protocol:[target][params]. The protocol part describes the type of connection that will be created. Tabel 2 shows the possible connection types. The target is optional and can be a host name, a network port number, a filename or a communication port number. The params is also optional and can be information needed to make the connection. Listing 5 shows a code example where the method *Connector.open*(*Stringurl*) is implemented.

Listing 5: A connection method

```
public void connect() throws IOException {
    HttpConnection hc = null;
    hc = Connector.open("http://www.sf.org");
}
```

The method *connect* generates an instance of the *HttpConnection* and connects the mobile device through http with the web server from http://www.sf.org.

The listing in Appendix A combines all previous code examples to one connection midlet.

4 Static checking with JML and ESC/Java2

Java source code can be verified using JML and the static checking tool ESC/Java2. Using these tools many errors can be detected in an early stage of the development of an application namely at compile time. This will be cost reducing and, more important, will improve the quality of the source code. This chapter will describe the basic ideas and techniques used for static checking Java source code.

4.1 JML

JML stands for "Java Modeling Language". It is a formal behavioral interface specification language for Java. JML can be used as a powerful tool for Java and is based on the "Design by contract" principle. The "Design by contract" principle is based on the fact that a class and its clients have a contract with each other. The clients who want to use this class must meet this contract. In practice this means that the client must guarantee the pre-conditions before it can call the method. A nice thing about "Design by contract" is that it can be used within the Java source code of the application. So the conditions in the contract can be executed along with the source code from the application. A static checking tool, in this case ESC/Java2 can use the JML annotated Java source code for a static check of the application. So any error in the code can immediately be detected by ESC/Java2.

The JML specifications can be added inside the Java code. To be sure that Java does not interpret the JML annotations as source code the JML annotations start with a special prefix. Listing 6 shows the two possible JML annotations.

Listing 6: JML as comments in Java

```
//@ <JML annotations>

/*@
        <JML annotations>
@*/
```

4.1.1 Relevant keywords

There are a limited number of JML keywords that can be used to annotate Java. The following keywords are most important for this research and therefor shortly described.

requires
The *requires* keyword describes a precondition for a method. This condition must always be true before the method can start executing. If the condition does not hold then the result of the method is undefined. This means that it is not possible to say anything about the behavior of code during its execution.

The actual code within the method is not important because a precondition is always defined for conditions at the start of the method.
A simple example:

```
//@ requires dollar >= 0;
public void addDollar(int dollar);
```

The method *addDollar* has the precondition that the parameter *dollar* must be equal to or bigger than zero. If the parameter is smaller than zero the source code does not meet the precondition and its correct behavior cannot be guaranteed.

ensures
The *ensures* keyword describes a postcondition for a method. This condition must always be true after the termination of the method. Just like the precondition the result must be true because else the behavior of the code cannot be guaranteed.
A simple example:

```
//@ ensures newDollar == 3;
public void addDollar(int dollar);
{
  newDollar = newDollar + dollar;
}
```

The method *addDollar* has the postcondition that the value of the variable *newDollar* must be three after the method has been terminated. If *newDollar* is not three after the termination then the behavior of the code cannot be guaranteed.

assignable
The *assignable* keyword is used to state which of the variables are allowed to change during the execution of the method. If a variable is changed during the execution and this variable is not stated as an *assignable* variable then the checker will throw an error. The default behavior of a method is that it can assign all variables. A simple example:

```
//@ assignable newDollar;
public void addDollar(int dollar);
{
  newDollar = newDollar + dollar;
}
```

In the method *addDollar* the value of the variable *newDollar* can change during the execution. This variable is allowed to change its value because it is annotated as an *assignable* variable. If the *assignable* annotation was missing then ESC/Java2 will throw an error whenever the variable *newDollar* is assigned a new value.

invariant
The *invariant* is a condition that must be hold during the execution of the

whole application. If the condition is not met then ESC/Java2 will show an error. The definition of the invariant is therefor done at the beginning of a class. A simple example:

```
public class myMidlet extends MIDlet
{
  //@ invariant ( newDollar > 0 && newDollar < 100 );
}
```

The invariant states that the value of the variable *newDollar* must be bigger than 0 and smaller than 100. ESC/Java2 will check this invariant at the beginning and at the end of every methodn.

constraint
The keyword *constraint* must be met during the execution of the whole application just like the *invariant*. The *constraint* describes the relationship between the pre- and postcondition. A simple example:

```
public class myMidlet extends MIDlet
{
  //@ constraint ( newDollar == 10 ==> \old(newDollar) < 10 );
}
```

The constraint puts a condition on the value of the variable *newDollar*. If the value of *newDollar* becomes 10 than the old value (defined with \old) of *newDollar* must be smaller than 10. So the value of *newDollar* can only be increased to 10 and not decreased to 10.

4.1.2 Relevant expressions

There are a limited number of JML expressions that can be used to annotate Java. The following expressions are most important for this research and therefor described.

\result
The expression \result is used to get the result of a method. This variable holds the return value of the method after its termination. A simple example:

```
//@ ensures \result > 0;
public int addDollar(int dollar);
{
  newDollar = newDollar + dollar;

  return newDollar;
}
```

The method *addDollar* returns the value of the variable *newDollar*. The postcondition of this method states that the return value of this method (the value of *newDollar*) must be bigger than zero.

\old(<name>)
The \old(<name>) is already stated in the definition of the *constraint*. This expressions holds the old value of *name* after it is assigned a new value.

a $\Longrightarrow$ b
The $\Longrightarrow$ expressions has also been stated in the definition of the *invariant*. This expression implies the that if condition a is true than also condition b must be true.

4.2 ESC/Java2

ESC/Java2 stands for Extended Static Checker for Java version 2. According to its developers [esc] it is

> A programming tool that attempts to find common run-time errors in JML-annotated Java programs by static analysis of the program code and its formal annotations.

The term "Static checking" comes from the fact that the checking of the source code is done without running the program. Basically there are two kinds of checking:

- Static checking
  Static checking is done at runtime and is a real verification method.
- Runtime checking Runtime checking is done at runtime and only test one situation to check if the source code behaves right.

The addition "Extended" is because ESC/Java2 can not only catch errors like type checkers do but it can actually reason about errors. This reasoning results in the catching of errors that are normally caught at runtime checking such as null-pointer exceptions and array-out-of-bound exceptions.

4.2.1 How to use ESC/Java2

The use of ESC/Java2 can be best shown in an example. Listing 7 shows a simple example of JML annotated Java.

Listing 7: JML embedded Java

```
import javax.microedition.midlet.*;
public class myMidlet extends MIDlet
{
  //@ invariant ( newDollar >= 0 );

  private int newDollar = 0;
  public myMidlet() {}
  public void startApp() {}
  public void pauseApp() {}
  public void destroyApp(boolean unconditional) {}

  //@ assignable newDollar;
  //@ ensures \result > 0;
  public int addDollar(int dollar) {
    newDollar = newDollar + dollar;
    return newDollar;
  }
}
```

Figure 7 shows the result running the code from listing 7 in ESC/Java2.

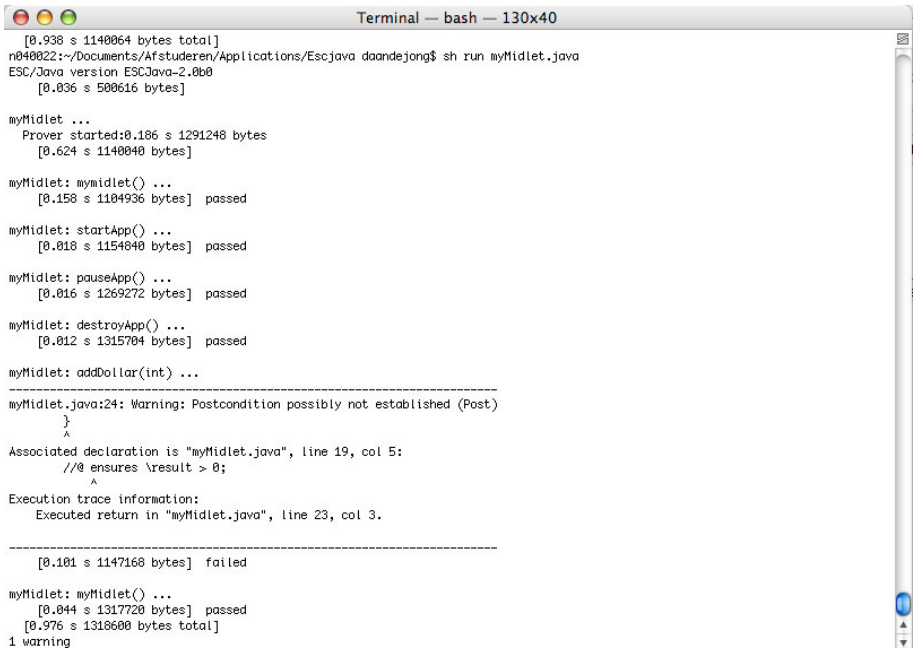

Figure 7: Static checking using ESC/Java2

The example causes one warning in ESC/Java2. The warning tells us that the postcondition, defined with the *ensures* keyword, may be broken. This means that ESC/Java2 cannot verify that the method *addDollar* returns a value that is equal or bigger than zero. The warning is fixed by adding an extra precondition for the *addDollar* method. To ensure that it is not possible

to return a value less than zero the method must require that the value of the parameter *dollar* is not less than zero. After adding this precondition ESC/Java returns no errors and the midlet is verified. Listing 8 shows the correct JML annotated Java source code.

Listing 8: JML embedded Java

```
import javax.microedition.midlet.*;
public class myMidlet extends MIDlet {
  //@ invariant ( newDollar >= 0 );

  private int newDollar = 0;
  public myMidlet() {}
  public void startApp() {}
  public void pauseApp() {}
  public void destroyApp(boolean unconditional) {}

  //@ assignable newDollar;
  //@ requires dollar >= 0;
  //@ ensures \result >= 0;
  public int addDollar(int dollar) {
    newDollar = newDollar + dollar;
    return newDollar;
  }
}
```

5 Midlet navigation graphs

Annotating the Java source code of midlets with JML and running the code through ESC/Java2 is a way to verify the behavior of midlets. To keep control of the behavior of midlets first a convenient formalism for expressing specifications has to be defined. This chapter describes the development of a standard for the definition of a formalism for midlet navigation graphs. Also the translation from the formal specification to JML is defined.

5.1 Basic assumptions

The definition of a formalism for midlet navigation graphs are defined with the following goals as a starting point:

- *The specifications must only have one possible interpretation*
  There are a number of different levels for specifying a midlet. The first one is simply to put the specification of a midlet in plain text. This specification is rather informal and can be interpret differently and is therefor not very useful. There is a need for a formal specification that only has one interpretation so it can be widely used.
- *Independent of the implementation of the midlet*
  In a typical situation the source code of the midlet is developed by an untrusted party. The midlet is developed using the (formal) specification. It could be possible that the actual implementation of the midlet is obfuscated because the developer does not want to open the source code. Another possible situation would be that the source code of the midlet is huge and there is not really a good code structure. In these cases it is too difficult and time consuming to annotate the implementation with JML. Every midlet uses API calls to use the basic functionalities of a mobile device such as the display and command events. The annotation of JML can be in the API for these basic functionalities.
- *Must be possible to define policies*
  The formal specification must be used to express basic (security) policies for the midlet. This is no problem if the policy talks about API functionality but becomes a problem if it talks about functionality of the midlet itself. This assumption is actually in contrast with the previous assumption because in most cases the policy talks about the functionality of the midlet.

The title of this section is “midlet navigation graphs”. This title is chosen because the formal definition is based on navigation graphs [Cre06]. Navigation graphs can be interpreted quite literally, it describes a way to navigate through a midlet. This navigation is defined in a formal specification so it can be verified whether the midlet actually follows a given navigation graph.

5.2 An example: A login system

To make the formal definition of midlet navigation graphs more clear and contextual let's start with an example. In a normal situation the developer of a midlet is given an specification of the properties of that midlet. Let's assume we like to develop a simple midlet which can show a secret phrase to persons who have the right pin-code which allows entrance to the secret display.

The first thing that is typical for developing such an application is to define its specification in words (an informal specification). The informal specification should be like this.

- After the application is started an intro display shows. This display provides the basic information of the application. The user can press the "OK" button to go to the login display. This display contains a text field where the user can type the pin-code to unlock the application. The pin-code is hard coded and its value is "1234". If the user enters the correct pin-code then the display will appear which shows the secret message, the secret display.

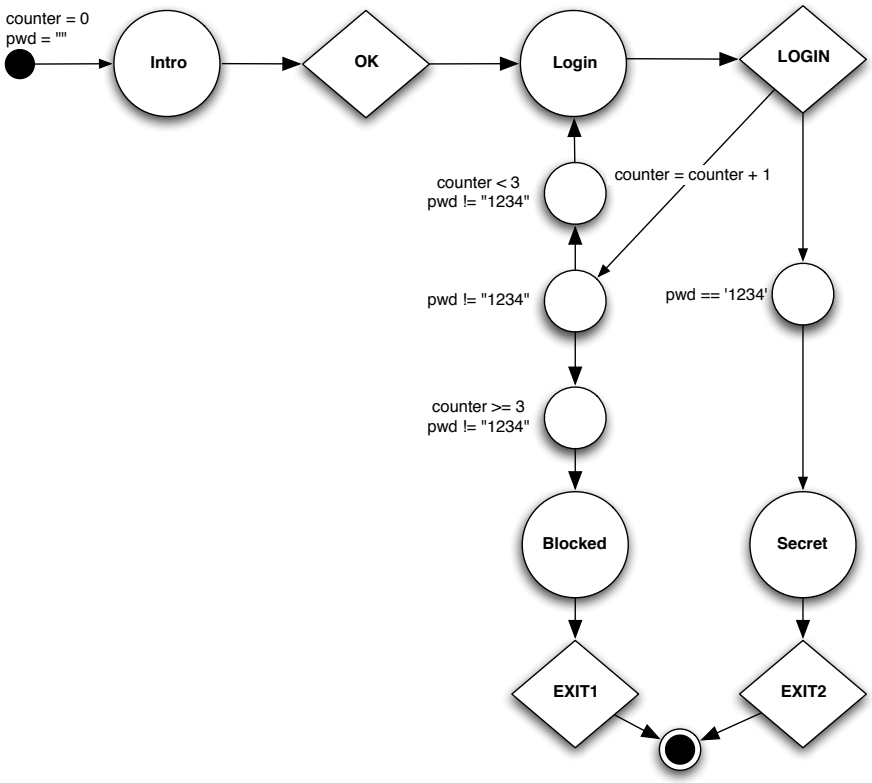

Figure 8: Midlet navigation graph of a login system

In the secret display the user can quit the application by pressing the "EXIT" button. If the user enters a wrong pin-code the application will return to the login display again. An extra security policy is defined that the user has only three changes to enter a correct pin-code. After three wrong entered pin-codes the application will show a display with the message that the user is blocked. If the user presses the "EXIT" key then the application quits. This policy only holds during the lifetime of the midlet.

Figure 8 shows the resulting midlet navigation graph, the formal specification of the midlet. This formal specification desribes the basic behavior of the midlet, the displays, command events and the security policy as described earlier. The next subsection will make clear what the representation of the formal specification actually means.

Appendix B contains the source code of this login midlet.

5.3 Defining midlet navigation graphs

In this section the definition of midlet navigation graphs is explained. The definition is done by looking at Figure 8. This subsection will make clear what the meaning is of the building blocks in this figure. Each subsection contains a building block within a midlet navigation graph. From each building block first an example is stated. After the example the formal definition is explained. Finally the translation to JML is defined.
We start with a simple format for navigation graphs and introduce more features one by one.

5.3.1 Two displays and a transition

Example Figure 9 contains a part of the example midlet of subsection 5.2.

Figure 9: Example MNG with two display states and a transition

One of the functionalities of that midlet is showing the login display. After the midlet starts the "Intro" display is shown. If the user presses the "OK" button the display changes into the "Login" screen. In figure 9 the two displays "Intro" and "Login" are defined as circles. The changing from one display to the

other display is defined as an arrow, a transition between the displays. Pressing of the "OK" button is left out in this example.

Formal definition The shapes in Figure 10 labeled with $ss == S1$ and $ss == S2$ are display states. These states each represent a particular state of the mobile device display. To avoid confusion between the terms screen and display the following definitions are used. By screen we mean the physical screen of the mobile device. By display we mean something that is displayed on the screen. The variable ss holds the current display state of a midlet. If the value of ss in the midlet becomes $S1$ it means that the midlet is currently displaying display $S1$ on the screen of the mobile device. The state $S2$ is another possible display variant on the screen. The arrow between the two states $S1$ and $S2$ represents the actual state change, in this case the changing of the display. During the transition the value of ss will be changed from $S1$ to $S2$. The *set* statement above the arrow describes that the value of ss becomes $S2$ during this transition.

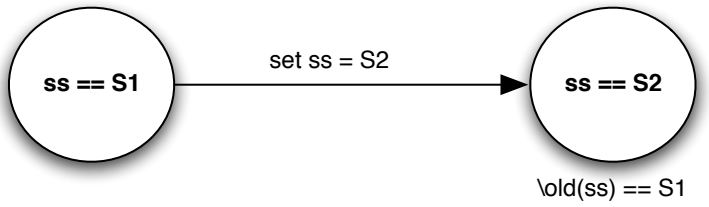

Figure 10: MNG with two display states and a transition

There is one method that changes the display from $S1$ to $S2$. This is the API method $setCurrent(Displayable\ nextDisplayable)$ from the *Display* class as mentioned in subsection 3.4. A note must be made that all displays must be unique. For a display this means that every display state represents a unique *Displayable* object.

Translation to JML The most negative approach would be that no source code of the midlet is available. This could be because the source code is for example obfuscated. This way all source code of the implementation of the midlet cannot be annotated using JML. What is left is to annotate the API with JML because this is always available. To annotate the API based on Figure 10 requires the introduction of global variables. These global variables are JML variables. Figure 10 requires only one global JML variable for the tracking of the display state of the screen, ss. The notation $set\ ss = S1$ describes that $S1$ is assigned to ss. The prefix *set* always describes an assignment of a global variable. State $S1$ represents the *Displayable* object and is unique in the FSM. Also the state $S2$ has an extra notation, $\backslash old(ss) == S1$. This notation describes a precondition for $S2$. The midlet can only reach state $S2$ if the previous state was $S1$. This means that there only can be a transition from $S1$ tot $S2$.

As mentioned the goal is to put all JML into the API methods. The

translation of Figure 10 to JML can mostly been done in the API. First the globals, invariants and constraints are defined for the states. The method *setCurrent*(*Displayable nextDisplayable*) must meet the pre- and postcondition.

Listing 9: JML from the display states

```
package javax.microedition.lcdui;

public class Display {

//@ public static model Displayable ss;

//@ invariant ss == S1 || ss == S2;

//@ constraint ss == S2 ==> \old(ss) == S1;
//@ constraint \old(ss) == S1 ==> ss == S2;

//@ assignable ss;
//@ requires ss == S1 && nextDisplayable == S2;
//@ ensures ss == nextDisplayable;
public void setCurrent(Displayable nextDisplayable);
```

Listing 9 shows the JML translated from Figure 10. The invariant is needed to define which values the JML variable *ss* can have. In this case this can only be $S1$ or $S2$. The constraints define the possible state changes that are allowed during the execution of the midlet. The state can change from $S1$ tot $S2$. If the display state is $S1$ this means that if the current value of *ss* becomes $S2$ than the old value of *ss* can only be $S1$. If the current display state is $S2$ than also the old value of *ss* needs to be $S1$. The API method *setCurrent*(*Displayable nextDisplayable*) causes the state change. In this case the precondition for this method is that the value of *ss* must be $S1$. The postcondition of this method must ensure that the value of *ss* is $S2$. To give *ss* the value of the next display the postcondition is added which states that the assignment has been done.

5.3.2 Two displays, a command event and a transition

Example Figure 11 contains another part of the midlet example in subsection 5.2. This figure is an extended version of the example in the previous subsection. In this example a command event (see subsection 3.5) is added. The display changes from "Intro" to "Login" by pressing the "OK" button. This button is drawn as a diamond. So the transition from "Intro" to "OK" is an event that the user presses a key. The transition from "OK" to "Login" is because the display has changed from "Intro" to "Login".
Important to notice is that Figure 11 has three states but there is no actual difference between these states. The visual difference (diamond and circle) is to avoid unclear midlet navigation graphs in complex midlets. With the visual difference between the states the midlet navigation graph can be interpret must faster.

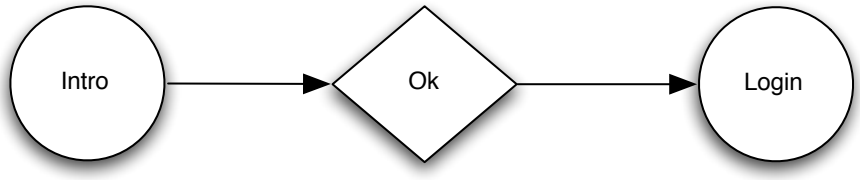

Figure 11: Example MNG with two displays states, a command event state and two transitions

Formal definition Figure 12 is an extension of Figure 10. The shape tagged with $ce == E1$ is a command event. This can be seen as a new state in the FSM, a command event state. So in total there are three states, two display states and one command event state. If the user presses the "E1" command button on the mobile device then the state changes from display $S1$ into command event $E1$. The command event state introduces a new global variable in the midlet navigation graph, ce.
In the command event state $E1$ the variable ss still has the value $S1$. So in state $E1$ there is information about the current display and also the current command event available. Therefor the state $E1$ is tagged with $ss == S1$. This is a pre-condition for the state change to $E1$. The command event $E1$ can only occur when the current display state is $S1$. The command event state also adds another precondition. To enter state $S2$ the value of ce must be $E1$. This means that the display $S2$ can only be shown if the user pressed the command button $E1$.

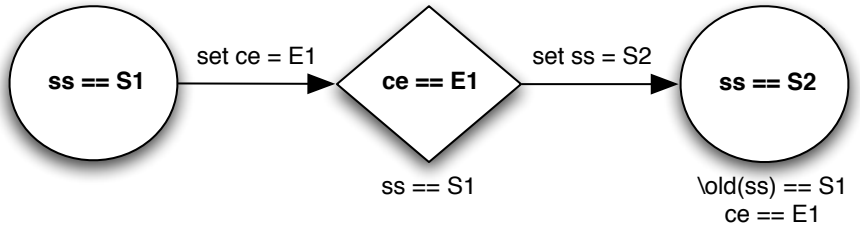

Figure 12: MNG with two display states, a command event state and two transitions

Translation to JML Listing 10 shows the JML translated from the FSM in Figure 12. The addition of command events introduces a new global JML variable ce which holds the last command event that occurred. Keep in mind that this JML is also an addition to the JML in the previous subsection. Because the command events are always executed by API methods, all of the JML can be added to the API. First the global variable ce is defined. This variable is a

Command object so it can store the actual command action object. Just like the display state an invariant is defined for the possible values that *ce* can have during the lifetime of the midlet. In this case the only value can be $E1$.
The method *commandAction*(*Command c*, *Displayable d*) is responsible for catching the command events. This method requires that the current display state is $E1$. After the termination of the method it must ensure that the value of *ce* is $E1$. To give *ce* the value of the next command event the postcondition is added which assigns the value of *Displayable d* to *ce*.
In the *CommandListener* class this JML must be added:

Listing 10: JML from the command event state

```
package javax.microedition.lcdui;

public interface CommandListener {

//@ public static model Command ce;

//@ invariant ce == E1;

//@ assignable ce;
//@ requires c == E1 && ss == S1;
//@ ensures ce == c;
void commandAction(Command c, Displayable d);
```

Listing 11: JML from the command event state and the display state combined

```
package javax.microedition.lcdui;

public class Display {

//@ public static model Displayable ss;

//@ invariant ss == S1 || ss == S2;

//@ constraint ss == S2 ==> \old(ss) == S1;
//@ constraint \old(ss) == S1 ==> ss == S2;

//@ assignable ss;
//@ requires ss == S1 && nextDisplayable == S2 && ce == E1;
//@ ensures ss == nextDisplayable;
public void setCurrent(Displayable nextDisplayable);
```

5.3.3 Start state

Example Figure 13 shows the start state which is also present in the example in subsection 5.2. If the midlet starts the constructor is executed and the midlet enters the *Pause* state as seen in 3.3. The midlet is in its initial state represented by the black circle in 13. The midlet enter the *Active* state during the execution of the *startApp*() method. Typically the default display state is set in this method. The value of *ss* is set to *Intro* in the *startApp*() method.

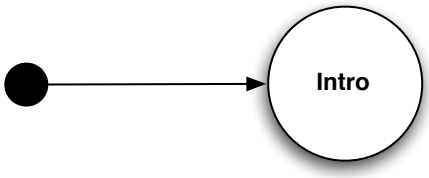

Figure 13: Example MNG with a start state

Formal definition The start state is only interesting for the initialization of the global variables. Figure 14 shows the start state in an FSM.

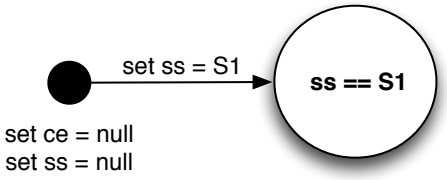

Figure 14: MNG with a start state

Translation to JML We must ensure that at the start of the midlet lifetime all variables are initialized. The midlet can be started by invoking its constructor. The midlet enters the $PAUSE$ state. To initialize the global variables we assume that both *ss* and *ce* have the default value *null*. The midlet can be started, e.g. change from the $PAUSE$ state to the $START$ state by invoking the abstract method *startApp*(). During the *startApp*() method the first display is shown on the screen. In the case of the login midlet this is the $INTRO$ display. The following listing shows the initialization of global variables in a midlet $MyMidlet$.

```
import javax.microedition.midlet.*;
import javax.microedition.lcdui.*;

public class MyMidlet extends MIDlet implements
    CommandListener {

public MyMidlet {
        //@ assume ce == null;
        //@ assume ss == null;
}
```

5.3.4 Final state

Example Figure 15 shows the final state in the example of 5.2. The midlet can be destroyed by calling the *destroyApp*(*boolean unconditional*) method or the $notifyDestroyed()$ method. Typically the midlet is destroyed by the user entering the "EXIT" button. The black circle represents the final state of the midlet.

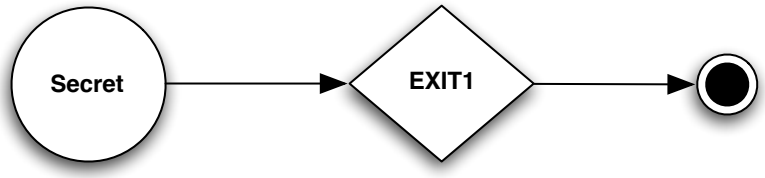

Figure 15: Example MNG with a final state

The final state in Figure 16 is almost similar to the start state. The *destroyApp*(*booleanunconditional*) method is always executed at the end of the midlet. After this method is executed all globals must have their default values. But this is mostly to complete the FSM rather than to add extra restrictions on the source code.

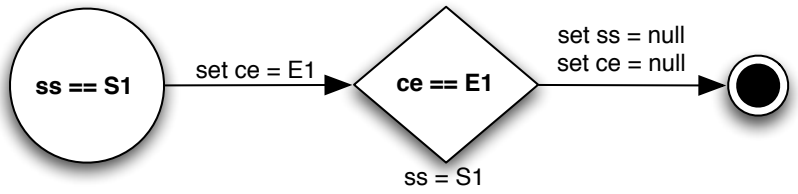

Figure 16: MNG with a final state

Translation to JML The *destroyApp*() method must ensure that all variables are reset to their initial value.

```
package javax.microedition.midlet;

public abstract class MIDlet {

//@ assignable ce;
//@ assignable ss;
//@ ensures ce == null;
//@ ensures ss == null;
protected abstract void destroyApp(boolean unconditional);
```

Appendix C shows the annotations for the login midlets' displays and command events including the default values.

5.3.5 Conditions

Example Figure 17 show a part of the example midlet which specifies a (security) policy. Let's assume the midlet shows the "Login" display. Here the user can enter a password. If the password is correct, its value must be "1234", then the display state changes. Practically an extra state is defined where the global variable *pwd* equals "1234". In the state where the display shows the "login" display, the global variable has not the value "1234" but the initial value or a wrong value. Only if the value of *pwd* has the value "1234" then the state of the display can change to "Secret".

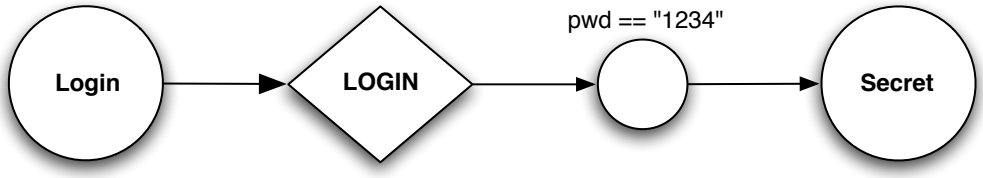

Figure 17: Example MNG with an extra condition

Formal definition Figure 18 shows a more formal definition of a midlet navigation graph with a condition.

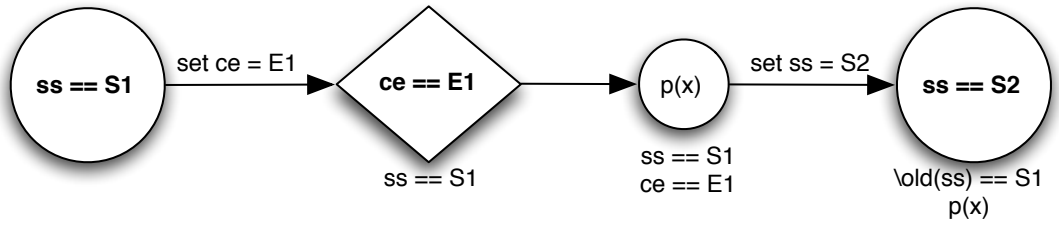

Figure 18: MNG with an extra condition

There could be specifications where you like to add some restrictions on transitions. Therefor we introduce the global variable x which represents a random variable. The condition $p(x)$ is a pre-condition for entering the state displayed as a small circle in Figure 18.
The basis of the midlet navigation graph are the display states (in this case $S1$ and $S2$). The extra condition adds an extra restriction on the transition from one display state to another display state. In the formal definition the

display can only change from $S1$ to $S2$ if the condition is met of the variable x. Otherwise the display state stays the same.

Translation to JML Like the name already says the condition adds an extra (pre) condition. The condition $p(x)$ must be met to execute. Also the changing of the display gets an extra pre-condition. Before the display can change first $p(x)$ must be met, so an extra precondition is added to the *setCurrent*(*Displayable nextDisplayable*) method of the *Display* class.
For this midlet navigation graph in Figure 18 we have the following JML annotations:

Listing 12: JML with an extra condition

```
package javax.microedition.lcdui;

public class Display {

//@ public static model Displayable ss;

//@ invariant ss == S1 || ss == S2;

//@ constraint ss == S2 ==> \old(ss) == S1;
//@ constraint \old(ss) == S1 ==> ss == S2;

//@ assignable ss;
//@ requires ss == S1 && nextDisplayable == S2 && ce == E1 &&
    p(x);
//@ ensures ss == nextDisplayable;
public void setCurrent(Displayable nextDisplayable);
```

Of course you need a function that actually controls the condition $p(x)$. In the login example this could be a method *checkPassword*(). If the given password is correct this method can make the condition true.

5.3.6 Assignments

Example Figure 19 shows the part of the example in the login system where an extra assignment is added to a transition. If the user of the login midlet sees the login display on the screen of his mobile device. He enters a wrong password and pushes the "login" button on his device. In the security policy is stated that it is only possible to enter a wrong password three times. After the third try the midlet will show the "Blocked" display.
An extra state is added after the user enters the wrong password. To keep track of the total number of tries a new global variable is introduced, *counter*. The midlet can only enter the extra state after the counter has incremented its value. In the definition of the start state the initial value of *counter* is defined (and set to zero).

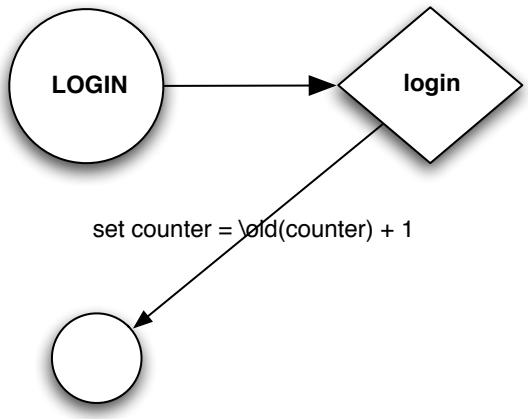

Figure 19: Example MNG with an extra assignment

Formal definition There could be specifications where you like to let transitions do an assignment to a given global variable x. This variable changes during the transitions. So the transition from the state where $ce == E1$ to state where $x == X1$ ensures that the assignment of x is performed.

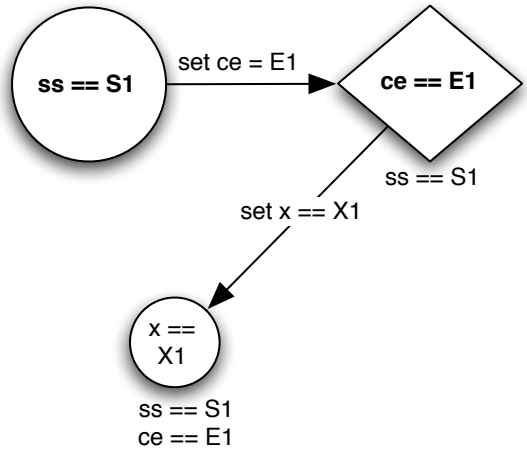

Figure 20: MNG with an extra assignment

Translation to JML There must be a piece of code in the source of the midlet that causes the assignment of x. Let's assume this method is called $method()$. This method must ensure that the assignment of x is done.

```
//@ assignable x;
//@ ensures x == X1;
public void method();
```

5.3.7 Branching transitions

Example Figure 21 shows the part of the navigation graph where the checking of the correct input password takes place. This checking is actually an expansion of the condition statements in section 5.3.5. If the user enters his password and hits the "Login" button then the midlet checks if the password is valid. If the password is valid, its value equals "1234" then the midlet enters the left state. If the value equals not "1234" then the midlet enters the right state. So executing a command event causes a divided transition.

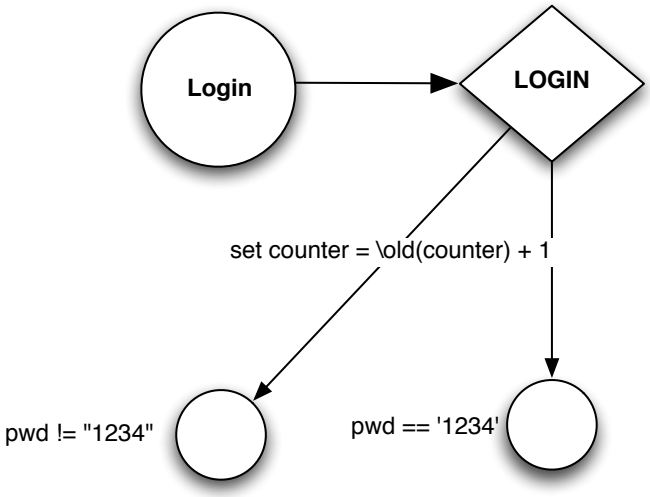

Figure 21: Example MNG with branching transitions

Formal definition In Figure 22 a more formal definition is made from the branching transitions. If the midlet shows the display $S1$ and the user hits the command event $E1$ then the dividing is a fact. The midlet enters state $x1$ if the p1(x) is met and enters state $x2$ if the p2(x) is met.

Because we want to annotate the API rather than the implementation itself this could be a problem. In the most positive case branching is based on the execution of a API method but in general this won't be the case. There are three solutions to still prove this kind of dividing transitions:

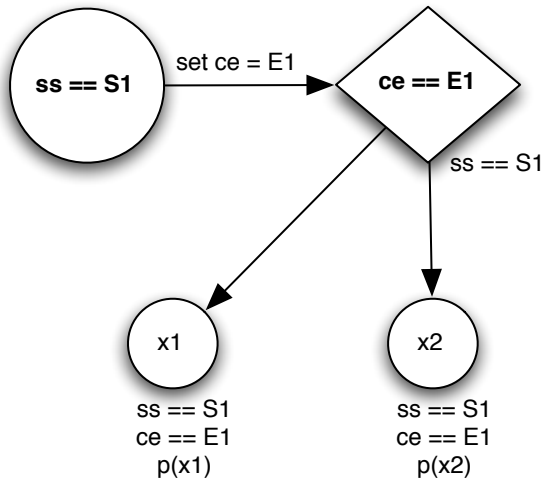

Figure 22: MNG with branching transitions

- We implement the method ourself and force the developer to use it.
  This way we already implement part of the midlet including the correct JML annotations. The developer must use this method in his implementation. This solution is not really scalable in practical situations.
- We define an abstract method which must be implemented by the developer.
  The abstract method can be seen as an addition to the API and therefor this method becomes implementation independent. This way we can partly annotate the method.
- Let the developer implement the methods.
  This is harder because we have to analyze all source code to find out whether it performs well. This will cost a lot of effort and must therefor be avoided.

Translation to JML The difference with the previous FSM is that the transition branch based on the execution of an method in the implemenatation. In the JML the method *method*() must be annotated with some extra postcondition. This postcondition ensures that the right transition is execution based on the right result of this method. The method *method*() is the implemented method which checks the password. It returns false if the password is incorrect (p(x1) is met) and returns true if the password is correct (p(x2) is met).
The following JML annotations are added. This would be the JML annotation if the method returns a boolean.

```
//@ ensures (p(x1) ==> \result == false);
//@ ensures (p(x2) ==> \result == true);
public boolean method();
```

6 Conclusions

In this chapter the conclusions are stated and based on the research objectives from chapter 2, the experiences during this research and the definitions of the midlet navigation graphs. This conclusion contains observations, interpretations and insights from all the facts described in this paper.

Research objective Let's take a look back at Figure 23 from chapter 2 which shows the idealized proposed solution.

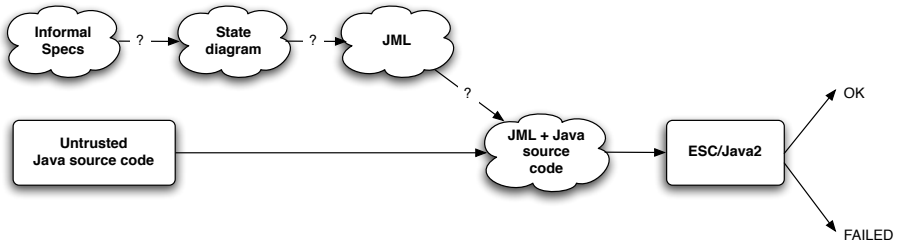

Figure 23: Idealized proposed solution

The question was and still is if it is possible to rebuild the current informal use of flow charts to a formal one. During the research I mostly worked on the following two aspects:

- Definition of a formalism for midlet navigation graphs
  In Figure 23 this is the building block tagged with "State diagram".
- Translation from midlet navigation graphs to JML
  In Figure 23 this is the building block tagged with "JML".

The definition of a formalism for midlet navigation graphs as defined in this paper for the displays and command events are complete and can be used on other midlets. The (extra) policies for midlets as defined are still basic. These must be used on several complex midlets to guarantee its practical value. It is not clear if these definitions of policies can be used for every wanted behavior of a midlet. The definition is based on simple midlets. It is not yet clear if this definition is still useful for formalizing the behavior of complex midlets.

The translation from midlet navigation graphs to JML are defined in such a way that most of the JML is added to the API. The translation from displays, display changes, command events and command events changes are defined in there own API classes. The extra policies are mostly translated to JML in a very abstract way. To verify a midlet these annotations cannot be copied one on one to the midlet implementation or API. To let ESC/Java2 verify the midlet many extra annotations must be added and these are not defined in this research. Therefor this research does not hold an example of an JML annotated

midlet based on the defined translation in this research. The building block in Figure 23 tagged with "JML + Java source code" is still not really clear. So the actual implementation of these translations to JML must be defined in a further research.

Context The idea behind this research was to make a basis for the definition of midlet navigation graphs, and the translation from these midlet navigation graphs to JML. The results defined in this paper are a small step towards a full grown verification tool for midlets. The definitions are not yet complete but for someone who is working 24/7 with midlets this can be the basis to develop such a tool. Important is that this basis is used by someone who has a lot of experience with midlets or/and JML because with this he can improve the definitions based on his practical experiences. There already are signs that this method for verification of midlet can be completely automated. For example using the AutoJML tool ([aut]) because this tool shows many equalities for defining a FSM for a Java application.

Reflection This type of research and the subject of the research has been quite new for me. Therefor it was much more difficult then I first expected. The experiences that I have with Java, JML and ESC/Java2 where minimal and also midlets was a complete new subject. The biggest mistake I made was to force myself to end up with the verification of a midlet navigation graph formalized in JML for a complex midlet. Because the focus was to complete the formalism of midlet navigation graphs, much more then only the most important parts of the midlet. The result is therefor much more "flat" then one would expect. The definitions in this research are still not really proven and the completeness of it must be doubted.
If I could start all over I would focus on a very simple midlet with only a few displays and some events. From this midlet I would make a proof of concept and a JML translation which could be easily used on other midlets. This way there would be a working basis for another researcher to continue. With the current results the basis must still be extended to complete a proof of concept such as the JML implementation between the displays and command events.

7 Further research

The research on midlet navigation graphs is still far from complete. The definition of a formalism for midlet navigation graphs in this paper is just a basis for further research. More research must be done in this area. This chapter describes the possible directions that are interesting for research.

Definition of formalism for midlet navigation graphs The formal definition of the midlet navigation graphs needs further research. There are still parts of a midlet that can be defined in the formal specification. First of all the current definition must be extended by other parts of a midlet such as multi-threading or network connections. Another research can be the usage of the definitions on much more complex midlets.

Implementing JML in Java The translation to JML has been done in an abstract way for the building blocks of the midlet navigation graphs. Research must be done to see how these JML translation must be implemented in the midlet or even better in the API. And do the translations in this research hold if there are implemented in much more complex midlets?

Automatic tool If the two points mentioned are completed then they must be implemented using an automatic tool in stead of by hand. Research must be done to look whether it is possible to automate the process of annotate a midlet using its midlet navigation graph. Interesting would be to look if the AutoJML tool [aut] can be used for this translation.

References

[aut] `http://autojml.sourceforge.net/`. Sourceforge project page from the AutoJML tool.

[Cre06] P. Cregut. Midlet navigation graphs. June 2006.

[esc] `http://secure.ucd.ie/products/opensource/ESCJava2/`. Official website from the ESC/Java2 verification tool.

[HO03] E. Hubbers and M. Oostdijk. Generating JML specifications from UML state diagrams. *In Proc. Forum on specification and Design Languages (FDL '03')*, September 2003.

[HOP04] E. Hubbers, M. Oostdijk, and E. Poll. Implementing a formally verifiable security protocol in Java Card. In Proceedings of the 1st International Conference on Security in Pervasive Computing, volume 2802 of LNCS, pages 213–226. Springer-Verlag, 2004.

[tes06] Unified Testing Criteria for Java Technology-based Applications for Mobile Devices, May 2006. http://Javaverified.com/docs/UTC_2_1.pdf.

APPENDIX

A Connection midlet

Listing 13: A connection midlet

```
import javax.microedition.midlet.*;
import javax.microedition.lcdui.*;
import java.io.*;
import javax.microedition.io.*;

public class MyMidlet extends MIDlet
                      implements CommandListener
{
  private Form helloForm;
  private Display myDisplay;
  private Command exitCommand;
  private Command okCommand;

  public myMidlet() {
    exitCommand = new Command("Exit", Command.EXIT, 1);
    okCommand = new Command("Ok", Command.OK, 1);

    helloForm = new Form("example");
    helloForm.addCommand(okCommand);
    helloForm.addCommand(exitCommand);
    helloForm.setCommandListener(this);
  }

  public void startApp() {
    myDisplay = Display.getDisplay(this);
    myDisplay.setCurrent(helloForm);
  }

  public void commandAction(Command c, Displayable s) {
    if (c == exitCommand) {
      destroyApp(false);
      notifyDestroyed();
    }
    else if (c == okCommand) {
      try {
        connect();
      }
      catch (IOException e) {}
    }
  }

  public void connect() throws IOException {
      HttpConnection hc = null;
      hc = Connector.open("http://www.sf.org");
  }

  public void pauseApp() {}

  public void destroyApp(boolean unconditional) {}
}
```

B Source code login midlet

Listing 14: Source code login midlet

```
import javax.microedition.midlet.*;
import javax.microedition.lcdui.*;

public class Login extends MIDlet implements CommandListener {

  public Form intro;
  public TextBox login;
  public Form blocked;
  public Form secret;

  public Display myDisplay;

  private Command OK;
  private Command LOGIN;
  private Command EXIT1;
  private Command EXIT2;

  private StringItem introText;
  private StringItem blockedText;
  private StringItem secretText;

  private int counter;
  private String pwd;

  public Login() {
    initiliaze();
  }

  public void initiliaze() {
    myDisplay = Display.getDisplay(this);

    OK = new Command("Ok", Command.OK, 1);
    LOGIN = new Command("Login", Command.OK, 1);
    EXIT1 = new Command("Exit", Command.EXIT, 1);
    EXIT2 = new Command("Exit", Command.EXIT, 1);

    introText = new StringItem("Intro", "This is a simple
        loginsystem");
    blockedText = new StringItem("Blocked", "You are blocked")
        ;
    secretText = new StringItem("Secret", "Secret phrase: MNG"
        );

    intro = new Form(null, new Item[] {introText});
    intro.addCommand(OK);
    intro.setCommandListener(this);

    login = new TextBox("Enter the secret phrase:", "", 256,
        0);
    login.addCommand(LOGIN);
```

```
    login.setCommandListener(this);

    blocked = new Form(null, new Item[] {blockedText});
    blocked.addCommand(EXIT1);
    blocked.setCommandListener(this);

    secret = new Form(null, new Item[] {secretText});
    secret.addCommand(EXIT2);
    secret.setCommandListener(this);

    counter = 0;
    pwd = "1234";
  }

  public void startApp() {
                myDisplay.setCurrent(intro);
  }

  public void pauseApp() {}

  public void destroyApp(boolean unconditional) {}

  public void commandAction(Command c, Displayable s) {
    if (c == OK) {
        myDisplay.setCurrent(login);
    }
    else if (c == LOGIN) {
      if (login.getString().equals(pwd)) {
        myDisplay.setCurrent(secret);
      }
      else {
        counter = counter + 1;
        if (counter >= 3) {
          myDisplay.setCurrent(blocked);
        }
        else {
          login.setString("");
          myDisplay.setCurrent(login);
          }
                }
    }
    else if (c == EXIT1 || c == EXIT2) {
      destroyApp(false);
      notifyDestroyed();
    }
  }
}
```

C JML annotations login midlet

If we use the translation on the example in Figure 8 the following JML must be added to the *CommandListener* class:

Listing 15: JML from the command event states in the example

```
public interface CommandListener;

//@ public static model Command ce;

//@ invariant ce == null || ce == OK || ce == LOGIN || ce ==
    EXIT1 || ce == EXIT2;

/*@ constraint (ce == OK ==> \old(ce) == null) &&
        (ce == LOGIN ==> \old(ce) == OK) &&
        (ce == EXIT1 ==> \old(ce) == LOGIN) &&
        (ce == EXIT2 ==> \old(ce) == LOGIN) &&
        (ce == null ==> \old(ce) == OK || \old(ce) == LOGIN ||
             \old(ce) == EXIT1 || \old(ce) == EXIT2);
@*/
/*@ constraint (\old(ce) == null ==> ce == OK) &&
        (\old(ce) == OK ==> ce == LOGIN || ce == null) &&
        (\old(ce) == LOGIN ==> ce == OK || ce == null) &&
        (\old(ce) == EXIT1 ==> ce == null) &&
        (\old(ce) == EXIT2 ==> ce == null);
@*/

//@ assignable ce;
//@ requires (c == OK && ce == null && ss == Intro) ||
        (c == LOGIN && ce == OK && ss == Login) ||
        (c == EXIT1 && ce == LOGIN && ss == Blocked) ||
        (c == EXIT2 && ce == LOGIN && ss == Secret);
//@ ensures ce == c;
void commandAction(Command c, Displayable d);
```

In the *Display* class the following JML must be added:

Listing 16: JML from the display states of the login midlet

```
public class Display;

//@ public static model Displayable ss;

//@ invariant ss == null || ss == Intro || ss == Login || ss
    == Blocked || ss == Secret;

/*@ constraint (ss == null ==> \old(ss) == Intro || \old(ss)
    == Login || \old(ss) == Secret || \old(ss) == Blocked) &&
        (ss == Intro ==> \old(ss) == null) &&
        (ss == Login ==> \old(ss) == Login) &&
        (ss == Secret ==> \old(ss) == Login) &&
        (ss == Blocked ==> \old(ss) == Login);
@*/
/*@ constraint (\old(ss) == null ==> ss == Intro) &&
        (\old(ss) == Intro ==> ss == Login || ss == null) &&
        (\old(ss) == Login ==> ss == Blocked || ss == Secret
            || ss == Login || ss == null) &&
        (\old(ss) == Secret ==> ss == null) &&
        (\old(ss) == Blocked ==> ss == null);
@*/

//@ assignable ss;
//@ requires (ss == Intro && ce == OK && (nextDisplayable ==
    Login) ||
        requires (ss == Login && ce == LOGIN && (
            nextDisplayable == Login || nextDisplayable ==
            Blocked || nextDisplayable == Secret)) ||
        requires (ss == Blocked && ce == EXIT1 &&
            nextDisplayable == null) ||
        requires (ss == Secret && ce == EXIT2 &&
            nextDisplayable == null) ||
        requires ((ss == Intro || ss == Login || ss == Blocked
             || ss == Secret) && nextDisplayable == null) ||
        requires (ss == null && nextDisplayable == Intro);
//@ ensures ss == nextDisplayable;
public void setCurrent(Displayable nextDisplayable);
```