Midlet Navigation Graphs in JML

Wojciech Mostowski and Erik Poll

Department of Computing Science, Radboud University Nijmegen
P.O. Box 9010, 6500 GL Nijmegen, The Netherlands
`woj@cs.ru.nl`, `erikpoll@cs.ru.nl`

Abstract. In the context of the Mobius project on Proof Carrying Code for mobile device Java applications (so called midlets), we present a way to formalise midlet navigation graphs in Java Modelling Language. Navigation graphs are used to express certain security policies of a midlet. Our resulting JML specifications are automatically verifiable with the static Java checker ESC/Java2. In the paper we relate to the earlier work on midlet navigation graphs generation from Java bytecode. Our work complements this earlier work by providing a mechanisms for establishing midlet navigation graph properties. In the paper we also discuss in detail practical difficulties with creating efficient and meaningful JML specifications for automatic verification with a lightweight verification tool. Our work was guided by one of the Mobius project realistically sized case studies developed by our industrial partners.

1 Introduction

A midlet navigation graph is a formal mechanism of specifying security properties for Java MIDP (Mobile Information Device Profile) applications, so-called midlets. The origin of the midlet navigation graphs is the Mobius project,[1] where different techniques for enabling and supporting Proof Carrying Code (PCC) for mobile devices are developed. Based on a simpler notion of a flow graph proposed in [17] as part of the Java Verified scheme[2] to test midlets, Pierre Crégut presented the notion of a navigation graphs in [13] as a high-level specification language to describe the behaviour of a Java mobile phone applications (in most cases MIDP devices are in fact mobile phones).

Essentially, a navigation graph is a graph, or finite automaton, which describes all the ways in which an application may navigate through various screens, in interaction with the user and the network. Each node in the graph represents different screen contents that is displayed, for instance a warning message for which the user has to press “OK” or “Cancel”, or a menu where the user can select one of the options. The arrow between nodes represent possible transitions the application can make, often in response to user action. The graph is also augmented with information about sensitive midlet actions, for example sending an SMS or engaging in other GSM network activity. Overall this gives a high-level specification of which potentially dangerous things a given midlet does, and under which circumstances.

In [5] Crégut gives a formal description of navigation graphs and their semantics in terms of an operational semantics of Java bytecode, more specifically in terms of the Bicolano semantics [16]. He also presents an algorithm to extract a navigation graph from bytecode. In this paper we formalise the semantics of navigation graphs in terms of a specification at source code level. The formal specification language we use for this is Java Modelling Language (JML) [11].

We guided our work with one of the Mobius project case studies, namely a quiz game midlet developed by TLS[3] for the Mobius project. This is the biggest case study of the project and it exhibited some problems during the JML annotation process and also during verification with the tool of our choice, the extended static checker for Java, ESC/Java2 [4]. We will discuss the problems we encountered and midlet JML specification caveats in detail.

[1] `http://mobius.inria.fr`
[2] `http://www.javaverified.com`
[3] `http://www.tls.pl`

The rest of this paper is organised as follows. Section 2 gives an introduction to the MIDP application structure. Section 3 describes midlet navigation graphs in more detail. Section 4 discusses the formalisation of the midlet navigation graphs in JML based on the Mobius case study. Finally, Section 6 gives a summary and discussion about our and related work.

2 MIDP Infrastructure

The notion of midlet navigation graphs relies on the infrastructure of the Java2 Micro Edition platform (J2ME)[4] for small devices such as mobile phones or PDAs. The two building blocks of the J2ME platform that we are interested in are the Mobile Information Device Profile (MIDP) Java API and the Connected Limited Device Configuration (CLDC) Java API. In particular, the former API deals with output (to the display) and input from the user via the keypad of the device. The latter API is mostly responsible for device's communication with the outside world. J2ME is often referred to as MIDP and CLDC is usually assumed to be part of J2ME/MIDP when mobile phones or PDAs are considered. The applications that run on the devices are called midlets.

GUI A navigation graph is related to the phone's display and sensitive operations that a midlet can possibly perform. In the following we discuss the relevant parts of the MIDP API. As a presentation aid we use the Unified Modelling Language (UML) [15] notation.

Every midlet, represented by the `MIDlet` class, has a unique associated `Display` object, which manages the display and the input devices. The static method `Display.getDisplay(MIDlet m)` returns the display that is associated with a midlet. The various kinds of things that can be displayed on the `Display` object are instances of the subclasses of `Displayable`. Invoking the method `void setCurrent(Displayable nextDisplayable)` on a `Display` changes what it displays, possibly after a short delay. A `List` presents a list of choices, i.e. a menu, that the user can scroll through and select. A `TextBox` allows the user to enter and edit text. A `Form` presents an arbitrary mixture of items, which can for instance be images or (read-only or editable) text fields. Finally, an `Alert` is a screen that is shown for a short period of time, either until some timeout or a some key press. Alerts can also be displayed by invoking the method `void setCurrent(Alert alert, Displayable nextDisplayable)`, which will display the `alert` (until the time out or some key press), and then show the `nextDisplayable`.

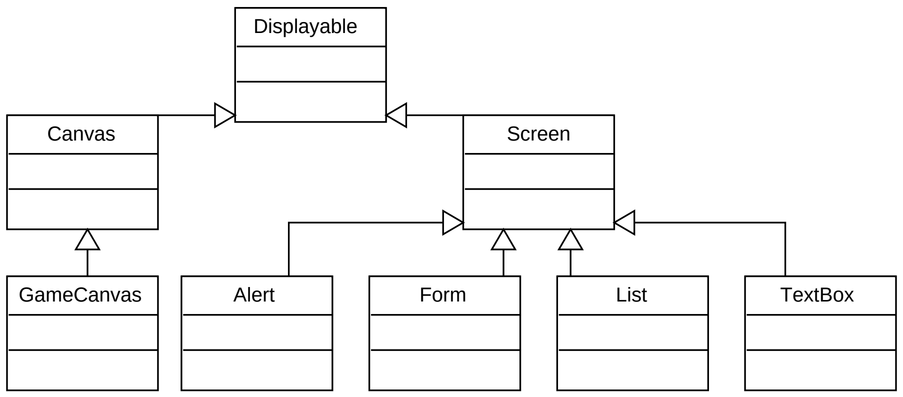

Fig. 1. Class diagram: basic MIDP GUI elements

[4] `http://java.sun.com/javame/index.jsp`

In its turn, a `Displayable` object may have a `CommandListener` associated with it, which implements a method `void commandAction(Command c, Displayable d)` to handle incoming command event occurring on some `Displayable d`. `Commands` are basically user actions, such as soft buttons that the user can select from a list of choices on the screen. Figures 1 and 2 depict the relevant class structure. Note that control passes back and forth between the MIDlet and the

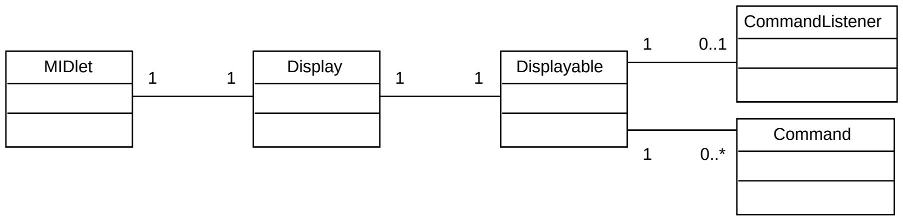

Fig. 2. Class diagram: relation between midlets, displays, and screens.

platform. For instance, when a MIDlet calls `setCurrent()` to change the display, it hands over control to the platform, and when after that a user action occurs, the platform hands back control to the midlet by a call back to `commandAction()` on the `CommandListener` associated with that `Displayable`. The behaviour of the midlet is determined by:

- the current `MIDlet` and its `Display`,
- the current `Displayable` shown on that `Display`,
- the `Commands` that the midlet provides (if any),
- the associated `CommandListener` (if any).

The display and the midlet should never change. The MIDP platform controls which `Displayable` is shown, and offers a midlet API calls to change it. The midlet is in charge of the `Commands` and `CommandListeners` and their associations to `Displayables`.

2.1 Sensitive Operations

The second relevant part of the MIDP infrastructure are the APIs responsible for network communication and personal information access. Following the Mobius project guidelines these two kinds of operations are considered sensitive with respect to security. An unwanted or uncontrolled network communication may result in (a) sensitive data being sent out from the phone, or (b) unwanted network usage charges. Access to personal information database (e.g. the phone book) may also result in secure information leakage. Among other things, the Mobius project aims at providing Proof Carrying Code facilities to establish secure behaviour of mobile applications w.r.t. above mentioned properties.

The high level API structure for network communication and personal information access is very simple. In principle we only need to be concerned with the `Connector` class that provides static methods for establishing different kinds of network connections (SMS, Internet), and a few classes that encapsulate these different connections, like `MessageConnection` or `HttpConnection`. Figure 3 gives a simplified view.

Similarly for personal information access there is one utility class `PIM` that provides methods for accessing the phone book (contact list), and a few classes that represent a single contact or the whole contact list (Figure 4).

3 Navigation Graphs

In [5] two formalisations are given to deal with midlet navigation graphs. The first formalisation deals with the MIDP GUI structure. We described it intuitively using UML, [5] gives a more

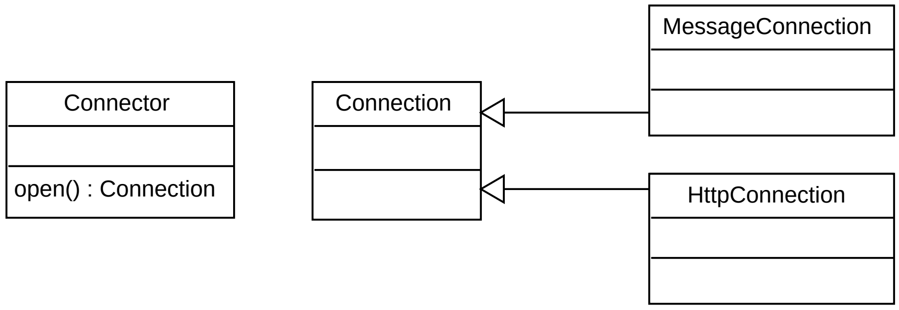

Fig. 3. Class diagram: network connection related API

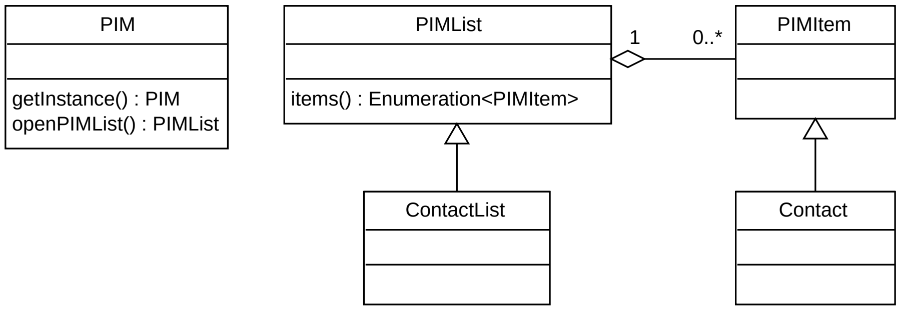

Fig. 4. Class diagram: personal information related API

detailed semantics in terms of the Bicolano [16] semantics of the execution of the midlet bytecode. There the formalisation of the GUI is needed to develop the formal algorithm for generating a navigation graph out of the exiting application's bytecode. For our purposes the lightweight UML representation of the GUI is sufficient to formalise the GUI behaviour in JML by annotating the GUI parts of the MIDP API. Although we do not use the detailed formalisation of the GUI from [5], there is a close correspondence between that formalisation and ours. For example, our $1 - 0..1$ relation between `Displayables` and `CommandListeners` is the relation g.list in [5], where g denotes the state of the GUI and the whole relation maps `CommandListeners` to `Displayables`. Similarly, the relation g.coms corresponds to our $1 - 0..*$ mapping between `Displayables` and `Commands`.

The second part of the formalisation in [5] gives a formal notion of a midlet navigation graph itself. Putting aside the complex notation a midlet navigation graph is essentially an oriented multigraph. The nodes of this graph are possible midlet states, i.e. different application screens. The arrows of the graph are transitions between the screens caused by user actions, i.e. commands. Finally, in [5] the arrows (transitions) also have interpretations. In particular, these interpretations indicate what possible sensitive operations can be performed during a given transition.

Such a notion of a graph can also be easily represented by a UML state chart diagram. The states of the diagram represent application screens, the arrows represent user commands, and arrow guards can be used to mark sensitive operations. A very simple example is given in Figure 5. In this example, from the main screen a user can choose the Send command, in which case at most one SMS would possibly be sent over the network, or press End, in which case the application will simply terminate. Furthermore, using the UML state stereotypes we can indicate additional properties of screens, e.g. whether an alert screen is displayed only for a given period of time indicated by the timeout parameter (and hence performing a transition to another screen without user interaction). This last aspect is not covered in [5].

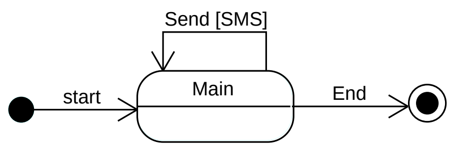

Fig. 5. Statechart diagram: a simple midlet navigation graph

4 Navigation Graphs in JML

In this section we give the semantics of a navigation graph in terms of JML. In other words, we define a mapping from navigation graphs to JML annotations. Our effort is divided into two parts. We start with the first part, which is to specify, in a generic way, the midlet API calls. The API specifications can then be used when specifying a particular midlet behaviour to reflect a given midlet navigation graph in the second part that we discuss later.

4.1 Relevant API Methods

For all our JML annotations we want the specifications to be easy for the verification tool to make the verification process efficient. Then we also want to include as many as possible navigation graph related specifications in the API itself, so that specification of a concrete midlet is easier and quicker. In the end such considerations lead us to the approach of having navigation graph-centric API specification, rather than all purpose, detail rich, generic API specification. Ideally, the API specifications should not specify anything else apart from the properties that we will need later to specify the midlet graph behaviour. This results in a relatively compact API specification that is easy to maintain and simple for the verification tools. Throughout our specifications we will often use JML host variables. Ghost variables are essentially the same as regular Java variables, only that they visibility is limited to specifications.

As mentioned earlier we need to deal with two aspects of the API behaviour: the GUI and sensitive operations. For the GUI routines we need to specify the following. First, the `Display` class responsible for assigning display object to midlets:

```
public class Display {
  // Display represents the manager of the display.
  // There is exactly one instance of Display per MIDlet

  //@ public non_null ghost MIDlet midlet;
  //@ public non_null ghost Displayable current;
  //@ public non_null ghost Alert preAlert;

  //@ ensures \result != null && \result.midlet == m;
  //@ assignable \nothing;
  public /*@pure@*/ static Display getDisplay(/*@non_null@*/ MIDlet m);

  //@ ensures current == nextDisplayable;
  //@ ensures preAlert == null;
  //@ assignable current, preAlert;
  public void setCurrent(/*@non_null@*/ Displayable nextDisplayable);

  //@ ensures current == nextDisplayable;
  //@ ensures preAlert == alert;
  //@ assignable current, preAlert;
  public void setCurrent(/*@non_null@*/ Alert alert,
                         /*@non_null@*/ Displayable nextDisplayable);
}
```

For the second setCurrent method we made a practical simplification. This method causes an alert screen (timed or with a confirmation button) to be displayed before the actual screen that one wants to show (nextDisplayable). To reflect this more accurately we could go for writing a more complex specification that would keep track of the sequence of displayed screens. However, verification would be substantially more difficult and with little added value. Instead, we record the information in a variable that an alert screen was requested to be displayed before the actual screen.

Next we specify the Displayable class that represents particular screens on the display, and the Command class that represent input events. To simplify things we assume that each Command object is bound to only one Displayable. Generally, this does not have to be the case (commands can be reused through different displayables). In practice most midlets define separate commands for each screen. The approach of having this one-to-one correspondence have impact on the performance of verification tools, as we do not have to employ, e.g., set theory in the reasoning. As for command listeners the platform enables only one per each screen:

```
public class Command {
  // The (only) Displayable object this command is attached to
  //@ public ghost Displayable displayable;
}

public Displayable {
  //@ public ghost CommandListener commandListener;

  //@ ensures cmd.displayable == this;
  //@ assignable cmd.displayable;
  public void addCommand(/*@non_null@*/ Command cmd);

  //@ ensures commandListener == l;
  //@ assignable commandListener;
  public void setCommandListener(/*@non_null@*/ CommandListener l);
}
```

Finally, for the CommandListener interface we simply put preconditions that reflect MIDP platform guarantees, namely that the current displayable and command are not null, and that the invoked command is in fact associated with the given displayable:

```
public interface CommandListener{

  //@ requires c.displayable == d;
  public void commandAction(/*@non_null@*/ Command c,
                            /*@non_null@*/ Displayable d);
}
```

If we trust the platform not to behave abnormally, these assumptions are safe.

For the remainder of the API specifications we want to limit the number of invocations (or totally prohibit) of API methods that may result in either network usage and private information access. The approach here is to first identify all such commands in the API and then declare suitable static ghost variables to count the number of invocations of such sensitive methods. The client code can then specify the limits on those security counter variables. One example of such method is the open method of the Connector class that establishes new network or SMS connections:

```
public class Connector {
  //@ public static ghost int openCount;

  //@ ensures Connector.openCount <= \old(Connector.openCount) + 1;
  //@ assignable Connector.openCount;
  public static Connection open(/*@non_null@*/ String name);
}
```

4.2 Midlet Annotations – the Mobius Case Study

We will present the JML annotations for midlets using the Mobius demonstration midlet. The midlet implements a simple mobile phone quiz game. The security sensitive operations of the quiz game are using an HTTP connection to download game questions, sending answers and scores in SMS messages, and also accessing the Personal Information Manager (PIM), i.e. the phone book. Figure 6 shows the complete graph that we use in the following.

The application class structure is as follows. There is a singleton class `QuizMidlet` which is the main application container. Then there are several classes to represent different screens of the game: main menu, options screen, about screen, the main game screen, etc. Most of these classes have a single instance. These classes also implement the `CommandListener` interface to handle user actions for each of the screens. The class `QuizQuestion` encapsulates a single quiz question and a displayable object associated with this question. Finally, there are two utility classes that handle network connections and PIM access. The whole application consists of 13 relatively small classes.

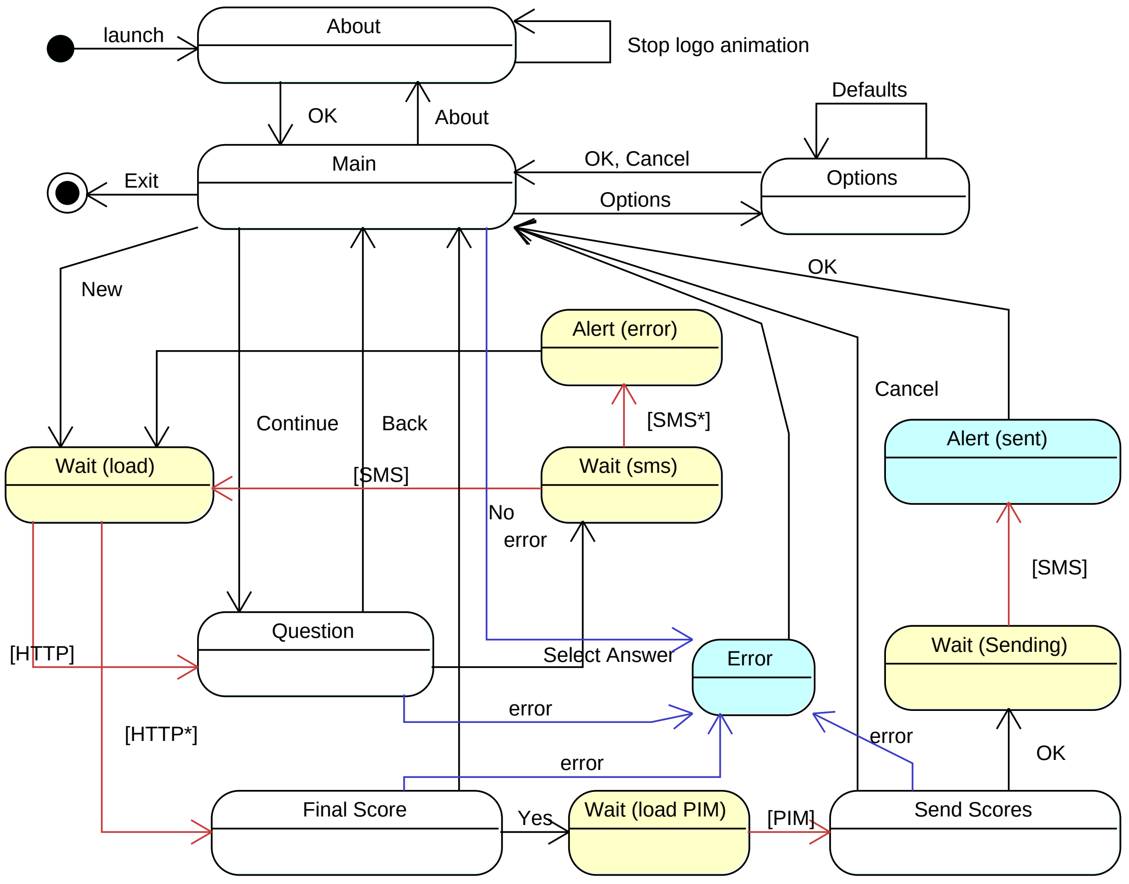

Fig. 6. Statechart diagram: Mobius game navigation graph

We start with specifying possible screens and screen transitions of our midlet. As it turns out this is the more difficult part and also one that exhibits the biggest problems with accurate reflection of the navigation graph in JML. Later we deal with the calls to sensitive operations.

Screen State To specify the navigation graph accurately we need to keep track of screen changes in our midlet. For this we use the `current` instance field of the `Display` object associated with our midlet. A suitable invariant enforces the limit on the set of possible screens, as follows:

```
public class QuizMidlet extends MIDlet {

  private /*@ spec_public non_null @*/ Display display;
  private /*@ spec_public non_null @*/ MainMenu mainMenu;

  /*@ invariant display.current == mainMenu.list ||
        display.current == About.about.alert ||
        display.current == Options.opts.form ...;  @*/
```

However, shortly we run into problems with completing this invariant. Namely, from the visibility point of view of the invariant we always need to have direct access to the reference of the current screen object, i.e. all possible object references of the `Displayable` type (like `mainMenu.list`) should be references visible from the `QuizMidlet` class. This is not always the case, as in the example of the `FinalScore` screen, whose instance is created locally in the `MainMenu` class:

```
public void quizFinished() {
  int score = currentGame.getScore();
  ...
  FinalScore finalScore = new FinalScore(midlet, score);
  finalScore.show(getDisplayable());
}
```

The `finalScore` reference is only visible locally in the `quizFinished` method and the displayable object displayed by the `FinalScore` class cannot be part of the global midlet invariant. There are other examples of such locally created displayables in our midlet code.

To solve this problem we turn to static ghost variables. Although the problematic displayable objects are created locally, there is only one object of a given kind created and possibly active at a time. So we simply store such locally created displayable object in a static ghost variable. Since we can make the static variable public it will be reachable by all our specifications. The fact that it is static lets us make sure that we keep track of only the most recently (and thus current) created displayable object of a given type, and also that access to this ghost variable is object reference independent. For the `FinalScore` class the relevant annotations are the following:

```
public class FinalScore implements CommandListener {
  //@ public static ghost Alert displayable;

  public void show(Displayable next) { ...
    Alert alert = new Alert("Final Scores");
    //@ set FinalScore.displayable = alert;
    alert.setTimeout(Alert.FOREVER); ...
    midlet.getDisplay().setCurrent(alert);
  }
}
```

Then our invariant can refer to the `FinalScore.displayable` field:

```
  /*@ invariant ...
        display.current == Options.opts.form ||
        display.current == FinalScore.displayable || ...; @*/
```

However, this solution brings up another problem. The visible state semantics of invariants requires all the invariants to hold in every visible state during the execution of our midlet. This obviously does not hold in the above code. Our invariant is broken in all the states between the state where `FinalScore.displayable` is set and the state when the display is updated with `setCurrent`. In these intermediate states `display.current` may point to a reference that is not stored in `FinalScore.displayable` anymore.

The intuition behind the solution to this last problem is to, so to say, suspend the validity of the invariant for the relevant intermediate states. In other words, we want to mark the relevant code with annotations that would relax the requirement on the invariant to hold in this context. In

Spec# a suitable framework has been developed [12] to deal with similar problems in a general way. There, objects can be packed and unpacked to indicate whether invariants should be checked for those objects. In short, when unpacked object do not have to satisfy the invariants. We employed a similar, but simpler solution. We introduce a global boolean guard, `Display.displayUpdated`, for the invariant:

```
public class Display {
  //@ public static ghost boolean displayUpdated;

  //@ ensures current == nextDisplayable;
  //@ ensures preAlert == null;
  //@ ensures Display.displayUpdated;
  //@ assignable current, preAlert, Display.displayUpdated;
  public void setCurrent(/*@non_null@*/ Displayable nextDisplayable);
  ...
}

  /*@ invariant Display.displayUpdated ==>
        display.current == Options.opts.form ||
        display.current == FinalScore.displayable || ...; @*/
```

By setting `Display.displayUpdated` to false we can temporarily switch off checking of the invariant. The specification of `setCurrent` ensures that the guard is reestablished again, so that the invariant has to hold after every call to `setCurrent`.

Screen Transitions The specifications above are sufficient to limit the set of screens of a given midlet. In the next step we need to specify when and how screen transitions happen. In general this is a very difficult problem: midlets are concurrent applications and screens can be changed by the J2ME environment at any time without user interaction. A notable example of this is an incoming call on a mobile phone, or simply midlet environment warning screens to, e.g., confirm sensitive operations. Note that we have already skipped such screens in the specifications above, including the navigation graph too. The non-deterministic character of such screens would make the graph and the specification unnecessarily complex.

Apart from the cases mentioned above the midlet screen transitions are triggered by user input and actions. All user actions are handled by different implementations of the `commandAction` method. This is where we want to put our annotations to limit the possible screen changes. In the precondition we limit the set of commands that can be invoked on this screen, in the postcondition we describe how the screen will change after processing the command:

```
//@ requires c==cmd_yes || c==cmd_no;
//@ requires next == midlet.mainMenu.list;
//@ ensures c==cmd_no ==> midlet.display.current == next;
//@ ensures c==cmd_yes ==>
      midlet.display.current == SendScores.displayable;
//@ assignable ...;
public synchronized void commandAction(Command c, Displayable d) {
  if (c == cmd_no) {
    midlet.getDisplay().setCurrent(next);
  } else if (c == cmd_yes) {
    SendScores sendScores = new SendScores(midlet);
    sendScores.show(score, next);
  }
}
```

Sensitive Operations In the last step we limit the sensitive operations our midlet performs. The method here is to first annotate every method with JML statement prohibiting any sensitive operations[5] using the API ghost static counters described at the end of Section 4.1:

```
//@ ensures Connector.openCount == \old(Connector.openCount);
//@ assignable ...;
public synchronized void commandAction(Command c, Displayable d) {
  ...
}
```

Then we allow the sensitive operations to be performed by the transitions in the graph:

```
public class SendScores {

//@ ensures c == cmd_ok ==>
     MessageConnection.smsSent <= \old(MessageConnection.smsSent) + 1;
//@ assignable MessageConnection.smsSent, ...;
public synchronized void commandAction(Command c, Displayable d) {
  ...
  else if (c == cmd_ok) {
    WaitAlert.getInstance().show(midlet, Consts.SENDING_RESULT);
    String s = numberField.getString();
    sendSMS(s);
  ...
}
```

A similar approach is used to limit access to personal data (e.g. the phone book) in the MIDP environment.

5 Encountered Specification and Verification Issues

During the verification of this case study with ESC/Java2 two practical issues surfaced. The first one has to do with singleton pattern classes, the second one with visible state semantics of invariants.

Singleton Objects Many classes in our case study are defined to be singletons, e.g. `Alert` or `Options` follow the singleton pattern [6]. On top of that the MIDP environment environment guarantees singleton instances of some objects, in particular it allocates only one instance of the midlet class. Knowing that a class is going to have a single instance simplifies specification and verification substantially, for example no aliasing between objects of the same type has to be considered.

JML does not provide a common stereotype to declare singleton classes. At the base of specifying a singleton behaviour we need to specify an invariant of this form:

```
  private /*@ spec_public @*/ static Options instance = null;
  //@ invariant this == instance;
```

This invariant is very strong, in the sense that the visible state semantics of invariants may cause problems in establishing this invariant. For example, the constructor call by itself does not establish it. Only when we call the constructor from the static `getInstance` method the invariant is established:

```
//@ ensures \result != null;
//@ ensures \result == instance;
//@ ensures \old(instance) == null ==> \fresh(instance);
```

[5] Note, that this step is not really necessary as long as `Connector.openCount` is not listed in the `@assignable` clause.

```
//@ assignable Options.instance;
public static Options getInstance() {
  if (instance == null)
    instance = new Options();
  return instance;
}
```

One way of overcoming such problems is to put an explicit `@assume` statement at the end of the constructor that establishes the invariant:

```
public Options() {
  ...
  //@ assume this == instance;
}
```

During verification explicit assumption statements override the current state of reasoning. This may easily lead to inconsistent reasoning. Another solution is to make the constructor a helper method:

```
public /*@ helper @*/ Options() { ... }
```

This says that the method should not be verified as such. Instead whenever it is called it should simply be inlined by the verification tool. This solution seems much better, however it turns out that ESC/Java2 reasons in an incorrect way when the helper statement is applied to a constructor.

In any case, we would like to see a more generic and simple way of specifying a singleton behaviour in JML, so that no necessary complications and side conditions are introduced during reasoning.

5.1 Visible State Semantics of Invariants

Visible State Semantics of Invariants in JML require all invariants to hold in every visible state during program execution. In practice this means that *all* the invariants have to be checked on *every* method entry and exit. During the work on the case study we encountered a situation when this may lead to inconsistent reasoning, and in turn to incorrect verification result. The base of the problem is that we are not verifying the API implementation, because it is defined only by its specification and there is no accompanying code. Consider this small example:

```
public class APIClass {
  //@ requires array.length == 1;
  //@ ensures array[0] == v;
  //@ assignable array[0];
  public static void setArray(/*@ non_null @*/ int[] array, int v);
}

public class ClientClass {
  private /*@ non_null @*/ int[] values = {1};
  //@ invariant values[0] == 1;
  //@ invariant values.length == 1;

  public void modifyArray() {
     APIClass.setValue(values, 2);
  }
}
```

The verification of the `modifyArray` method with ESC/Java2 does not show any problems. However, it is clear that a call to `APIClass.setValue` breaks the class invariant for `ClientClass`. This happens because the reasoning assumes that the API class is responsible for maintaining all invariants. Only the verification of the `setArray` method would reveal that the invariant of

`ClientClass` might be broken. In our scenario this is not acceptable as we have to treat the API class as a black box and use only its specification.

Some verification tools, like KeY [1], allow very flexible invariant semantics. There it is up to the user to choose which class (or even method) is responsible for a given invariant. For ESC/Java2 such flexible choice is not possible. Instead the tool offers a mechanism to discover inconsistent reasoning. By code reachability analysis the tool can spot the state in which inconsistent assumptions are introduced. A manual analysis of a resulting warning is required to find the source of inconsistency. However, we have to strongly note, that even with the reachability analysis enabled, the above example is successfully verified by ESC/Java2 and the violated invariant goes unnoticed. Thus the reachability analysis does not provide a complete solution to the problem. We believe that additional specification consistency checks (e.g. that method postconditions do not contradict invariants) are necessary in such cases.

6 Summary and Discussion

In this paper we have shown a way to annotate midlet source code with JML specification to establish a midlet navigation graph property. Part of our JML specifications is contained in the midlet API generic annotations that are used by particular midlets. Each midlet is then annotated with particular JML specification to ensure its correct behaviour w.r.t. the midlet navigation graph. We described some of the most notable problems we encountered during our work, including specific verification issues. Due to the requirements of our project and the lightweight nature of the verification tool we used some practical simplifications have been made to ease up the specification and verification process. In the remainder of this paper we discuss related and future work.

We see our work as a complement of Pierre Crégut's work [5] in the following sense. He provides a formalism for the graphs and presents a mechanism to generate graphs out of the bytecode. We, on the other hand, produce JML specifications based on a graph and annotate the source code. Additionally, our formalisation allows program verification tools that support JML to be used to verify that a midlet obeys the constraints imposed by a navigation graph. Using the infrastructure developed in the Mobius project, verification could also be carried out at bytecode level, using BML [3], the bytecode level counterpart of JML, and ultimately produce Proof-Carrying Code, where the bytecode is accompanied with a certificate (proof) that it obeys the constraints imposed by a navigation graph.

In [10] a formal relation between security automata and JML annotations is given. From the point of view of our work [10] concentrate on the sensitive operations invocations only rather than the whole navigation graph. We believe that our work could be incorporated into [10].

As we have shown, midlet navigation graphs can be easily expressed using UML diagrams [15]. To produce our JML specifications we could have strived to use existing techniques [14, 8, 7] for expressing formal properties based on UML diagrams. We did however find particular problems with employing such techniques. For example, the methods in [14, 8] assume a single point of entry and control for the specified application (the applications there are Java Card applets that satisfy this requirement). However, midlets are controlled concurrently by the running environment and the user input, so the mentioned techniques are not easily applicable. We are currently working on how to extend these techniques to match the MIDP environment requirements and to be able to automatically generate JML specification from navigation graphs.

An issue related to the previous paragraph that we have not discussed in detail in the paper is concurrency. Our case study follows a simple thread usage pattern. In our approach we have verified the threads in separation, but we have not verified the interaction between the threads. Although the interactions are very limited and should be easy to verify, we did not make any such attempt because ESC/Java2 does not support concurrent reasoning. Few of the verification tools can handle concurrency, and even if they can, complex specifications and verification methodology [2, 9] is usually involved. Thus we also left the concurrency aspect for future work.

Finally, ESC/Java2 was our primary tool choice in this work. We would like to employ other verification tools for our case study. In particular, we would like to attempt the verification with the KeY tool [1], which in principle is more powerful than ESC/Java2 and can also be run in automatic mode. Currently, however, the JML interfaces of the two tools are not fully compatible, and some extra specification maintenance is required to use both KeY and ESC/Java2 on our case study.

Acknowledgements This work is supported by the Information Society Technologies programme of the European Commission, Future and Emerging Technologies under the Mobius project (IST-2005-015905).

References

1. Bernhard Beckert, Reiner Hähnle, and Peter H. Schmitt, editors. *Verification of Object-Oriented Software: The KeY Approach*, volume 4334 of *LNAI*. Springer, 2007.
2. Bernhard Beckert and Vladimir Klebanov. A dynamic logic for deductive verification of concurrent Java programs with condition variables. In *Proceedings, 1st International Workshop on Verification and Analysis of Multi-threaded Java-like Programs (VAMP), Satellite Workshop CONCUR 2007, Lisbon, Portugal*, 2007.
3. Lilian Burdy, Marieke Huisman, and Mariela Pavlova. Preliminary design of BML: A behavioral interface specification language for Java bytecode. In *Fundamental Approaches to Software Engineering (FASE 2007)*, volume 4422 of *LNCS*, pages 215–229. Springer-Verlag, 2007.
4. Patrice Chalin, Joseph R. Kiniry, Gary T. Leavens, and Erik Poll. Beyond assertions: Advanced specification and verification with JML and ESC/Java2. In *Formal Methods for Components and Objects (FMCO) 2005, Revised Lectures*, volume 4111 of *LNCS*, pages 342–363. Springer, 2006.
5. Pierre Crégut. Extracting control from data: User interfaces of MIDP applications. In *Trustworthy Global Computing 2007*, volume 4912 of *LNCS*, pages 41–56. Springer-Verlag, 2008.
6. Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns*. Addison Wesley, 1999.
7. Engelbert Hubbers and Martijn Oostdijk. Generating JML specifications from UML state diagrams. In *Forum on Specification & Design Languages FDL'03*, pages 263–273. ECSI (European Chips and Systems Initiative) CD-ROM, 2003.
8. Engelbert Hubbers, Martijn Oostdijk, and Erik Poll. From finite state machines to provably correct Java Card applets. In Dimitris Gritzalis, Sabrina De Capitani di Vimercati, Pierangela Samarati, and Sokratis K. Katsikas, editors, *Security and Privacy in the Age of Uncertainty, 18th IFIP Information Security Conference*, pages 465–470. Kluwer Academic Publishers, 2003.
9. Marieke Huisman and Clément Hurlin. The stability problem for verification of concurrent object-oriented programs. In *Verification and Analysis of Multi-threaded Java-like Programs (VAMP)*, 2007.
10. Marieke Huisman and Alejandro Tamalet. A formal connection between security automata and JML annotations. In *Fundamental Approaches to Sofware Engineering (FASE) 2009*, LNCS, 2009. To appear.
11. Gary T. Leavens, Albert L. Baker, and Clyde Ruby. *JML: A Notation for Detailed Design*. Kluwer Academic Publishers, 1999.
12. K. Rustan M. Leino and Angela Wallenburg. Class-local object invariants. In *ISEC '08: Proceedings of the 1st conference on India software engineering conference*, pages 57–66, New York, NY, USA, 2008. ACM.
13. Mobius. Deliverable D5.1 – Selection of case studies. Mobius, `http://mobius.inria.fr`, 2005.
14. Wojciech Mostowski. Rigorous development of Java Card applications. In T. Clarke, A. Evans, and K. Lano, editors, *Proceedings, Fourth Workshop on Rigorous Object-Oriented Methods, London, U.K.*, March 2002. Available from `http://www.cs.ru.nl/~woj/papers/download/room2002.ps.gz`.
15. Object Modeling Group. *Unified Modelling Language Specification, version 2.2*, February 2009.
16. David Pichardie. Bicolano: a Java bytecode semantics in Coq. `http://mobius.inria.fr/twiki/bin/view/Bicolano`, 2006.
17. Unified Testing Initiative. Unified Testing Criteria for Java technology-based applications for mobile devices, version 3.0. The Java Verified Program, `http://javaverified.com`, 2009.