Midlet Navigation Graphs in JML

Wojciech Mostowski and Erik Poll

Radboud University Nijmegen
Digital Security Group
woj@cs.ru.nl, erikpoll@cs.ru.nl

Abstract. In the context of the EU project Mobius on Proof Carrying Code for Java programs (midlets) on mobile devices, we present a way to express midlet navigation graphs in JML. Such navigation graphs express certain security policies for a midlet. The resulting JML specifications can be automatically checked with the static checker ESC/Java2. Our work was guided by a realistically sized case study developed as demonstrator in the project. We discuss practical difficulties with creating efficient and meaningful JML specifications for automatic verification with a lightweight verification tool such as ESC/Java2, and the potential use of these specifications for PCC.

1 Introduction

Midlet navigation graph provide a way to specify security properties for Java MIDP (Mobile Information Device Profile) applications, so-called *midlets*. Based on a simpler notion of a flow graph, prescribed by the Unified Testing Criteria (UTC) [20] to test midlets in Java Verified scheme[1], Crégut proposed the notion of navigation graphs [5] as a high-level specification formalism to describe the behaviour of 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 the ways in which an application may navigate through various screens of the user interface, in interaction with the user and the network. Each node in the graph represents a different screen that is displayed, e.g. a warning message for which the user has to press 'OK' or 'Cancel', or a menu with options for the user to choose from. The arrows between nodes represent transitions the application can make, often in response to some user action. The graph can be augmented with information about sensitive midlet actions, e.g. sending an SMS or engaging in other GSM network activity. The navigation graph then 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 the Bicolano semantics [18]. He also presents an algorithm to extract a navigation

[1] http://www.javaverified.com

graph from bytecode. In this paper we present a way to express 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) [14].

Our work was guided by one of the Mobius case studies, a quiz game midlet developed by industrial partner TLS[2] [15]. 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 extended static checker for Java, ESC/Java2 [11]. We will discuss the problems we encountered with developing the JML specification and verifying it in detail.

The rest of this paper is organised as follows. Sect. 2 gives an introduction to the MIDP application structure. Sect. 3 describes midlet navigation graphs in more detail. Sect. 4 discusses the translation of the midlet navigation graphs in JML based on the Mobius case study. Finally, Sect. 6 summarises our results and discusses the potential for PCC.

2 MIDP Infrastructure

The notion of midlet navigation graphs relies on the infrastructure of the Java2 Micro Edition platform (J2ME)[3] for small devices, such as mobile phones or PDAs. The main building blocks of the J2ME platform of interest are the Mobile Information Device Profile (MIDP) Java API and the Connected Limited Device Configuration (CLDC) Java API. 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 these 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 UML.

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 associated with a midlet. The various kinds of things that can be displayed on the `Display` object are instances of the subclasses of `Displayable`, shown in Fig. 1. 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`

[2] `http://www.tls.pl`
[3] `http://java.sun.com/javame/index.jsp`

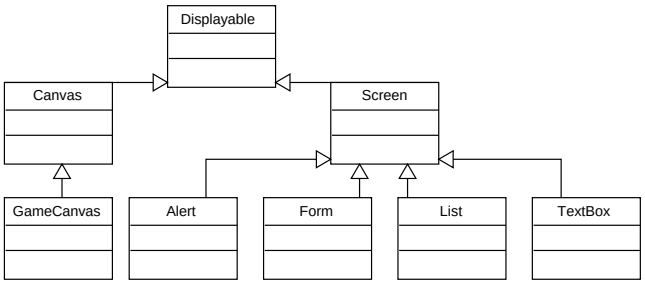

Fig. 1. Class diagram: basic MIDP GUI elements

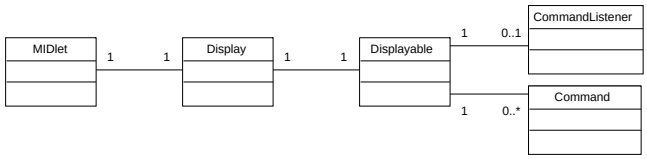

Fig. 2. Class diagram: relation between midlets, displays, and screens.

is a screen that is shown for a short period of time, either until some timeout or a key press. Alerts can also be displayed by invoking the method `void setCurrent(Alert alert, Displayable nextDisplayable)`, which will display the `alert`, and then show the `nextDisplayable`.

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 events occurring on some `Displayable d`. `Commands` are user actions, such as buttons that the user can select on the screen. Fig. 2 shows the relevant class structure. Note that control passes back and forth between the midlet and the platform. When a midlet calls `setCurrent(...)` to change the display, it hands over control to the platform; when after that a user action occurs, the platform hands back control to the midlet by a call back to `commandAction(...)`. The behaviour of the midlet is determined by: (i) the current `MIDlet` and its `Display`, (ii) the current `Displayable` shown on that `Display`, (iii) the `Commands` that the midlet provides (if any), (iv) 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`.

Sensitive Operations The second relevant part of the MIDP infrastructure are the APIs responsible for network communication and personal information management. These operations are possibly security-sensitive. 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 (e.g. the phone book) may result in unwanted information leakage.

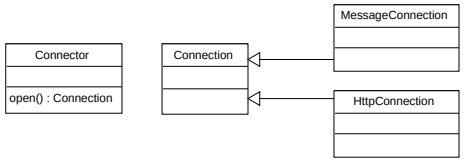

Fig. 3. Class diagram: network connection related API

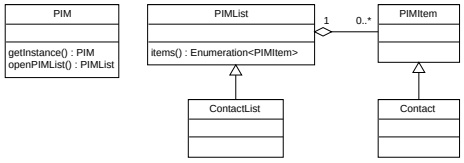

Fig. 4. Class diagram: personal information related API

The high level API structure for network communication is very simple. In principle it only involves 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`. Fig. 3 gives a simplified view.

For personal information management (PIM) 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 (Fig. 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. In [5] a more detailed semantics is given in terms of the Bicolano semantics [18] of Java bytecode; there the formalisation of the GUI is needed to develop the algorithm to generate navigation graphs out of bytecode. For our purposes the lightweight UML representation of the GUI is sufficient to represent the GUI behaviour in JML by annotating the parts of the MIDP API dealing with the GUI. Although we do not use the detailed formalisation of the GUI from [5], there is a close correspondence between that formalisation and our JML representation of the graphs. E.g., our $1 - 0..1$ relation between `Displayables` and `CommandListeners` is the relation $g.\mathsf{list}$ in [5], where g denotes the state of the GUI and the whole relation maps `CommandListeners` to `Displayables`. Similarly, the relation $g.\mathsf{coms}$ in [5] corresponds to our $1 - 0..*$ mapping between `Displayables` and `Commands`.

The second part of the formalisation in [5] gives a formal definition 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, as they indicate which sensitive operations that may be performed during a given transition.

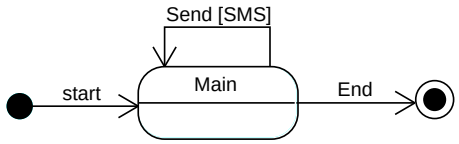

Fig. 5. Statechart diagram: a simple midlet navigation graph

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 Fig. 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].

4 Navigation Graphs in JML

This section gives 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

We want our JML specifications to be easy to verify for the verification tools. Hence we only specified those aspects that are needed for navigation graphs, and we avoided constructs that are challenging for verification, as discussed later. Our specifications often use so-called ghost variables [4], which are specification-only variables, to model relevant aspects of the state of the platform (incl. the GUI).

As mentioned before, two aspects of the API need to be specified: the GUI and security-sensitive operations. For the GUI, we need to specify the `Display` class, where a ghost field `current` tracks what is being displayed:

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

  //@ ensures current == nextDisplayable && 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 to be displayed temporarily (either with a time-out or with a confirmation button) before updating the display to show `nextDisplayable`. This could be specified by writing a complex specification that keeps track of the sequence of displayed screens, including these alerts. However, verification would be much more difficult and with little added value. Instead, we introduce a ghost field, `preAlert`, which records that an alert screen is temporarily displayed. So `currents` tracks the displayables being shown over time, ignoring temporary `Alert` displayables.

To specify the GUI behaviour, we also have to specify the `Displayable` class that represents particular screens on the display and the `Command` class that represents 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). However, in practice most midlets define separate commands for each screen, and requiring it makes verification simpler, as we do not need to use sets (or some representation of sets such as lists) to track the set of displays associated with a command, and then use set theory in verification. As for command listeners, the platform enables only one per screen:

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

public class 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, the specification of `CommandListener` should reflect the 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[4]:

```
public interface CommandListener {
  //@ requires   c.displayable == d;
  //@ assignable \everything;
  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 security-sensitive API calls, that may result in say network usage or access to private information, we want our API specification to track the number of invocations. For this we declare suitable static ghost variables to count the number of invocations. This allows us to express restrictions on these numbers in a specification for midlet. For example, for the `open` method of the `Connector` class, which establishes new network or SMS connections, we can specify

```
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. Fig. 6 shows the complete navigation graph of this midlet.

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,

[4] The `assignable \everything` clause in the spec means that classes implementing this method are in principle free to have any side-effects.

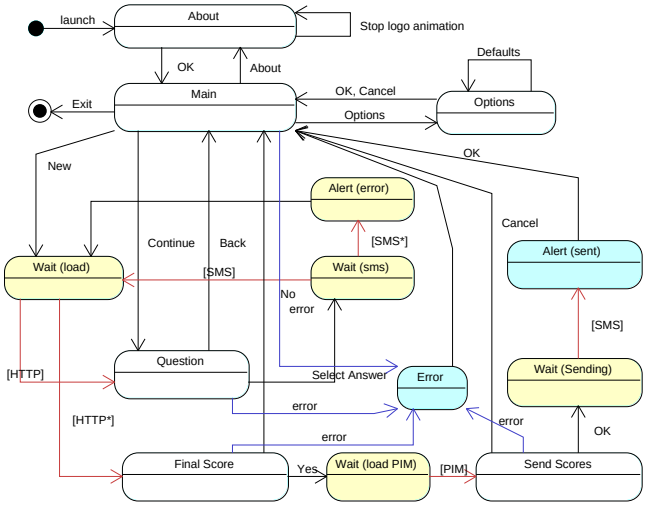

Fig. 6. State diagram: navigation graph for the Mobius game

the main game screen, etc. Most of these classes use the Singleton pattern, meaning there will only ever be a single instance of them. These classes also implement the `CommandListener` interface to handle user actions. The class `QuizQuestion` encapsulates a single quiz question and a displayable object for this question. Finally, there are two utility classes that handle network connections and PIM access. The whole application consists of 13 small classes.

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 specification 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, we quickly run into problems with completing this invariant, because for every screen – i.e. `Displayable` – the application uses we need some program variable (like `mainMenu.list`) to refer to it. Such variables do not always exist. Some objects of type `Displayable` are created on the fly, and cannot be referred

to from the class `QuizMidlet`. For example, this is the case for the `FinalScore` screen in the game, which 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` object, the screen that displays the score, is only visible locally in the `quizFinished` method and it cannot be referred to in a global midlet invariant. There are other examples of such locally created displayables in the 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 in scope for 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 invariants of all objects to hold in all visible states (i.e. all pre- and post-states of all method calls) during the execution of our midlet. This obviously does not hold in the code above. Our invariant is broken in all the states between the state where `FinalScore.displayable` is set and the state when the display is updated by invoking `setCurrent`. In these intermediate states `display.current` may point to a reference that is not stored in `FinalScore.displayable` anymore.

A (standard) trick we use to solve this problem is introducing a global boolean guard, `Display.displayUpdated`, to switch invariants on and off at appropriate points, as shown below.

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

  //@ ensures current == nextDisplayable && 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' the invariant. The specification of `setCurrent` ensures that the guard is re-established, so that the invariant has to hold after every call to `setCurrent`.

Screen Transitions The invariant above suffices 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, e.g. to confirm sensitive operations. Note that we have already skipped such screens in the specifications above, and in the navigation graph too. The non-deterministic character of such screens would make the graph and the specification unnecessarily complex. Furthermore, we are interested in verifying the application itself rather than the whole environment it runs in.

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 add annotations to limit the possible screen changes. The precondition limits the set of commands that can be invoked on this screen, the postcondition describes how the `FinalScore` 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 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. For any method that ultimately calls down to one of these operations we need to add a contract for the associated ghost variable that, as described at the end of Sect. 4.1, tracks the number of invocations.

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

Then we allow the sensitive operations to be performed by the implementations of the `commandAction` that correspond to transitions in the graph:

```
public class SendScores {
  //@ ensures c == cmd_ok ==>
     MessageConnection.smsSent <= \old(MessageConnection.smsSent) + 1;
  //@ assignable MessageConnection.smsSent, ...;
  public 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.

If properties are expressed in postconditions, we have to consider the issue of non-termination. Verification would have to be done using total correctness to ensure that, say, limits on the number of SMS sent are not broken in non-terminating executions of a method. Still, for `commandAction` methods which do not have the `MessageConnection.smsSent` in their assignable clause, the contract does rule out that any SMS are sent in non-terminating executions.

5 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 midlet code study and in the MIDP API follow the singleton pattern [8], i.e. only one instance of these classes is ever created. Knowing that a class is only ever going to have a single instance could in principle simplify reasoning. However, specifying that a class follows the singleton pattern in JML, and verifying it with ESC/Java2, can be a bit clumsy. E.g., specifying that `Options` is a singleton class could be specified by an invariant

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

saying that all `Options` objects are equal to the static `instance`. However, this invariant is typically too strong, given the visible state semantics of invariants. If the singleton object is created by a static get-method as is done in the code, e.g.

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

then the constructor call `new Options()` by itself does not establish the invariant, as it will not hold till after the assignment to `instance`.

A possible solution is to make the constructor a helper:

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

effectively telling ESC/Java2 to inline calls to the constructor, but it turns out that ESC/Java2 reasons in a rather unpredictable way when a private constructor is declared as `helper`.

In the end we let ESC/Java2 complain about the possibly unsatisfied `this == instance` invariant after a call to `new Options()` in `getInstance()` and suppressed this warning with the `@nowarn` directive.

It would be nice to have a standard, e.g. following ideas from [19], simple way of specifying singleton behaviour in JML, so that no necessary complications and side conditions are introduced during reasoning.

Visible State Semantics of Invariants JML uses the visible state semantics for object invariants [13]. This means that all invariants of all objects of all types have to hold in all *visible states*, which includes all post-states of constructor calls and all pre- and post-states of method calls[5].

This semantics makes verification very hard, and non-modular[6]: when verifying a method in one class one should take into consideration breaking invariants of other objects, of any class, that happen to be allocated. To enable modular verification, ESC/Java [7] uses a slightly weaker and potentially unsound semantics. ESC/Java2 [11] uses the same semantics, but now includes features to warn users about potential problems [12, 10].

Working on the case study, this generated several false positives, where the code was incorrectly deemed to be correct. The root of the problem was that when checking code that calls API methods, ESC/Java2 will assume that these API methods preserve all invariants. When checking the implementation of these API methods the tool would probably warn that they do not preserve invariants,

[5] Except those constructors and methods designated as `helper`'s

[6] However, the semantics is sound, unlike more simple-minded semantics for object invariants!

but as we are only ever checking the midlet code – i.e. client code of the API – and never implementation of this API, this issue can easily go undetected.

The example below illustrates this:

```
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 && values.length == 1;

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

Checking the `modifyArray` method with ESC/Java2 does not show any problems. However, it is clear that a call to `APIClass.setArray` breaks the class invariant for `ClientClass`. However, the tool assumes that the API class will re-establish all invariants.

Running the tool on an implementation of the `setArray` method would probably reveal that the invariant of `ClientClass` might be broken, but typically one treats the API as a black box and one does not look at implementations of it. Unfortunately the inconsistency warning of ESC/Java2 [12, 10] does not catch these situations.

Sound and modular verification techniques that cope with class invariants do exist [6], and are for instance used in Spec# [1]. Some verification tools, like KeY [3], allow a very flexible invariant semantics; there it is up to the user to choose which class (or even method) is responsible for a given invariant, but that means soundness is up to the user too.

6 Evaluation and Discussion

An issue often overlooked in research on program verification is coming up with interesting properties to verify. We have shown a way to specify midlet navigation graphs by means of JML annotations. This required some generic annotation of the MIDP API, which provides a 'ghost state' to talk about the relevant platform infrastructure – ghost fields that track which `Displayable` is being shown, and hence which `CommandListener` is active, etc. – and contracts for some API calls that describe their effect on this ghost state. An individual midlet can then be annotated to express conformance to a midlet navigation graph, by introducing an invariant that restricts the possible `Displayable`s that can be shown, and contracts for `commandAction` methods that specify the screen transitions that are allowed. Additional restrictions on security-sensitive API calls, saying in which states these may occurs, can be expressed if API specifications for these methods are added to track their usage.

The translation of a midlet navigation graph to JML, which we did by hand, could be automated, as done for state diagrams by the AutoJML tool [9]. An alternative to coding up the midlet navigation graph in JML pre- and postconditions, as we have done, would be to use special specification constructs for temporal properties [21] or CSP-style contraints on method sequences [16].

Our approach has been shown to work on a non-trivial (albeit still very small) midlet, the Mobius Quiz game demonstrator, which consists of 13 classes and 1350 lines of code, which was then verified using ESC/Java2. Once the midlet navigation graph is expressed by JML annotations, further annotations of the midlet are needed for the verification to go through, e.g. to rule out `NullPointerExceptions`, etc. This further annotation took an effort in the order of days.

As explained in Sect. 4.2, a technical complication is any use of dynamically created `Displayable` objects in midlet code, as additional static (ghost) fields have to be introduced to refer to these objects in specifications.

As discussed in Sect. 5, the main complication is the semantics of (object) invariants. JML's visible state semantics, which ESC/Java2 tries to check (ESC/Java2 is not guaranteed to be sound in this respect), is often stronger than we really need or want. The need for more flexible ways for dealing with invariants is of course widely recognised. Spec# [1] provides one such an approach, and alternatives are systematically compared in [6].

It is very important that the API specifications used are not too rich (i.e. too expressive) and *only* specify the aspects relevant for the navigation graphs. Otherwise the job of annotating and verifying the midlet can become *much* more complicated. Also, as discussed in Sect. 5, Singleton classes occur often in the MIDP API and in typical midlets. Specifying this in JML is a bit clumsy, and (hence) verification with ESC/Java2 seems more complicated than it needs to be. Better ways of dealing with this (possibly by additional primitives in JML) could simplify matters substantially.

A major caveat in our work is that we ignore concurrency. However, the concurrency patterns used in midlets are very simple – essentially, implementations of `commandAction` sometimes start up a worker thread to hand back control to the GUI as soon as possible – so might well be verifiable using a simple approach.

Midlet navigation graphs express safety properties of code, constraining the possible behaviour. This means that crashing of a midlet, say with a `NullPointerException`, can never violate the policy expressed by a navigation graph. This might allow verification to be simplified further: if we can guarantee that code never catches say `NullPointerException`s – which could be checked using a simple static analysis – then in the verification we could safely ignore the possibility of `NullPointerException`s. This is supported by some verification tools, e.g. the KeY tool [3] offers an option for such simplified reasoning.

PCC The overall goal of the EU project Mobius, in which this research was carried out, was to provide a Proof-Carrying Code (PCC) framework for Java on mobile devices [2]. PCC [17] involves (i) some security policy, (ii) some untrusted

code, and (iii) a proof that this code obeys the policy that can be checked. To ultimately use our approach in a PCC scenario, it has to be possible to distinguish the JML annotations expressing the desired security policy (i.e. the navigation graph) from any additional JML annotations that are needed for the verification to go through. For the former we must check that these really express the security policy we want. For the latter we do not: we don't care what these are, as long as the proof goes through.

It seems possible to make this distinction here. However, the JML annotations expressing the desired security policy – the navigation graph – do have to refer to program variables (namely the `Displayable`s that the program uses), so we cannot quite have these annotations – our "policy" – completely independent of the midlet code.

Also, a malicious midlet could contain specification statements (`set`-statements) that affect the values of the ghost state used in the API specification, making any certification meaningless; it would have to be checked that there are no such statements in the midlet code.

Of course, to then verify and provide PCC certificates for midlets, instead of using ESC/Java2, one should use a sound verification approach that can provide proofs (certificates) [2].

Acknowledgements This work is supported by the MOBIUS project in the Information Society Technologies programme of the European Commission and the CHARTER project in the ARTEMIS Embedded Computing Systems Initiative. We would also like to thank the anonymous reviewers for their insights.

References

1. M. Barnett, K. Leino, and W. Schulte. The Spec# programming system: An overview. In *CASSIS'04*, volume 3362 of *LNCS*, pages 151–171. Springer, 2004.
2. G. Barthe, P. Crégut, B. Grégoire, T. Jensen, and D. Pichardie. The MOBIUS proof carrying code infrastructure. In *FMCO'07*, number 5382 in LNCS, pages 1–24. Springer, 2008.
3. B. Beckert, R. Hähnle, and P. H. Schmitt, editors. *Verification of Object-Oriented Software: The KeY Approach*, volume 4334 of *LNAI*. Springer, 2007.
4. P. Chalin, J. R. Kiniry, G. T. Leavens, and E. Poll. Beyond assertions: Advanced specification and verification with JML and ESC/Java2. In *FMCO'05*, volume 4111 of *LNCS*, pages 342–363. Springer, 2006.
5. P. Crégut. Extracting control from data: User interfaces of MIDP applications. In *Trustworthy Global Computing*, volume 4912 of *LNCS*, pages 41–56. Springer, 2008.
6. S. Drossopoulou, A. Francalanza, P. Müller, and A. Summers. A unified framework for verification techniques for object invariants. In *ECOOP*, volume 5142 of *LNCS*, pages 412–437. Springer, 2008.
7. C. Flanagan, K. Leino, M. Lillibridge, G. Nelson, J. Saxe, and R. Stata. Extended static checking for Java. In *PLDI'2002*, pages 234–245. ACM, 2002.
8. E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns*. Addison Wesley, 1999.

9. E. Hubbers and M. Oostdijk. Generating JML specifications from UML state diagrams. In *Forum on Specification & Design Languages (FDL'03)*, pages 263–273. ECSI, 2003.
10. M. Janota, R. Grigore, and M. Moskal. Reachability analysis for annotated code. In *SAVCBS*, pages 23–30. ACM, 2007.
11. J. Kiniry and D. Cok. ESC/Java2: Uniting ESC/Java and JML. In *CASSIS'04*, volume 3362 of *LNCS*, pages 108–128. Springer, 2004.
12. J. Kiniry, A. E. Morkan, and B. Denby. Soundness and completeness warnings in ESC/Java2. In *SAVCBS'06*, pages 19–24. ACM, 2006.
13. G. Leavens, E. Poll, C. Clifton, Y. Cheon, C. Ruby, D. Cok, P. Müller, J. Kiniry, and P. Chalin. JML reference manual. Available from `http://www.jmlspecs.org`, 2003-2007.
14. G. T. Leavens, A. L. Baker, and C. Ruby. *JML: A Notation for Detailed Design*. Kluwer Academic Publishers, 1999.
15. Mobius. Deliverable D5.1 – Selection of case studies. Mobius, `http://mobius.inria.fr`, 2005.
16. M. Möller, E. Olderog, H. Rasch, and H. Wehrheim. Linking CSP-OZ with UML and Java: A case study. In *Integrated Formal Methods*, volume 2999 of *LNCS*, pages 267–286. Springer, 2004.
17. G. C. Necula. Proof-carrying code. In *POPL*, pages 106–119. ACM, 1997.
18. D. Pichardie. Bicolano: a Java bytecode semantics in Coq. `http://mobius.inria.fr/twiki/bin/view/Bicolano`, 2006.
19. C. Pierik, D. Clarke, and F. S. de Boer. Creational invariants. In *ECOOP Workshop on Formal Techniques for Java-like Programs (FTfJP'2004)*, 2004.
20. The Java Verified Program. Unified Testing Criteria for Java technology-based applications for mobile devices, version 3.0, 2009.
21. K. Trentelman and M. Huisman. Extending JML specifications with temporal logic. In *AMAST'2002*, volume 2422 of *LNCS*, pages 334 – 348. Springer, 2002.