CHALMERS

Formal Specification of Selected Android Core Applications and Library Functions

Master of Science Thesis in the Programme

Software Engineering and Technology

Masoumeh Al. Haghighi Mobarhan

Department of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
UNIVERSITY OF GOTHENBURG
Göteborg, Sweden, January 2011

Formal Specification of Selected Android Core applications and Libraries

Masoumeh alsadat.Haghighimobarhan,

Examiner: Reiner.Hähnle

Chalmers University of Technology
University of Gothenburg
Department of Computer Science and Engineering
SE-412 96 Göteborg
Sweden
Telephone + 46 (0)31-772 1000

Department of Computer Science and Engineering
Göteborg, Sweden January 2011

Department Of Computing Science And Engineering

Master of Science Thesis

Formal Specification of Selected Android Core Applications and Library Functions

by

Masoumeh Al. Haghighi Mobarhan

Supervisor : Dr.Richard Bubel
Examiner : Prof.Reiner Hähnle

Gothenburg,Sweden,2009-2010

Acknowledgments

I would like to express my gratitude to all those who gave me the possibility to complete this thesis project. Special thanks go to professor Reiner Hähnle for trusting me and offering the chance to work on this thesis. I am deeply indebted to my thesis supervisor Richard Bubel for all his help, support and valuable hints. He always had time for my questions and responded to them patiently. Finally, special thanks go to my family for supporting me to complete this master thesis.

List of Symbols and Abbreviations

Abbreviation	Description	Definition
JML	Java Modeling Language	page 1
KeY	Formal software development tool	page 1
SDK	Software Development Kit	page 2
API	Application Programming Interface	page 2
URI	Uniform Resource Identifier	page 6
OCL	Object Constraint Language	page 18
DL	Dynamic Logic	page 18
PIN	Personal Identifier Number	page 46
RIL	Radio Interface Layer	page 46
GSM	Global System for Mobile Communications	page 46
SIM	Subscriber Identity Module	page 46

Contents

List of Figures

Chapter 1

Introduction

Recently, mobile phones are one of the most popular devices among people; so that people's life in many aspects is dependent on them. Because of the variety of the usage of mobile phones and their role in our life, investigating and assessing the correctness of their functionalities are very vital.
Among platforms for mobile devices, Google's android platform is currently one of the most interesting developments in the mobile phone market.
The android platform consists of a Linux-based operating system, middle-ware and a set of core applications. The core applications are most likely part of all produced mobile devices running android and provide access to essential functionality.
Possible bugs in these parts affect thus all devices and have the potential of causing serious financial losses, security or even safety critical issues. For instance, the android application Phone provides an Emergency Dialer. To verify that a dialed emergency number is actually handed over to the carrier is an important safety property, while verifying that only emergency numbers are dispatched is a central security feature.
To identify critical classes and methods in the android API and core applications and also to assure that they fully satisfy all expected requirements are the purpose of this master thesis.
To achieve this goal firstly, we identified critical code pieces and then specified them in Java Modeling Language (JML) and finally verified them by KeY (formal software development tool).

The structure of the thesis is as follow:
Chapter 1: Introducing the android software platform for mobile devices and some of its features.
Chapter 2: Focusing on the specification and verification concepts and introducing JML as a specification language and KeY as a verification tool.
Chapter 3: Focusing on the specification of critical android applications like: Phone application, Screen manager and Contact application.
Chapter 4: Conclusion and Future Work.

In this chapter we introduce the android software platform and its features.

1.1 What Is Android?

"Android is a software platform for mobile devices that includes an operating system, middleware and key applications. The android SDK provides the tools and APIs necessary to begin developing applications on the android platform using the Java programming language"[1]. Android is developed by Google accompanied with the Open Handset Alliance (Intel, Motorola, Samsung, etc,...)[1]

1.2 Some Important Features

The following summarizes the most important features in android which are distinctive in comparison with other software platforms for mobile devices. Detailed information can be found in [1]

- **`Dalvik virtual machine`**: It is a virtual machine optimized for mobile devices. It is described in more details in section 1.3.2.
- **`Application framework`**: The android framework enables reuse and replacement of components. It is described in more details in section 1.3.4.
- **`Development environment`**: Android has a rich development environment including a device emulator, tools for debugging, memory and performance profiling, and a plugin for the Eclipse IDE.

1.3 Android Architecture

Fig. 1.2 shows the major components of the android operating system. The following summarizes the most important points. More detailed information about android components can be found in [2].

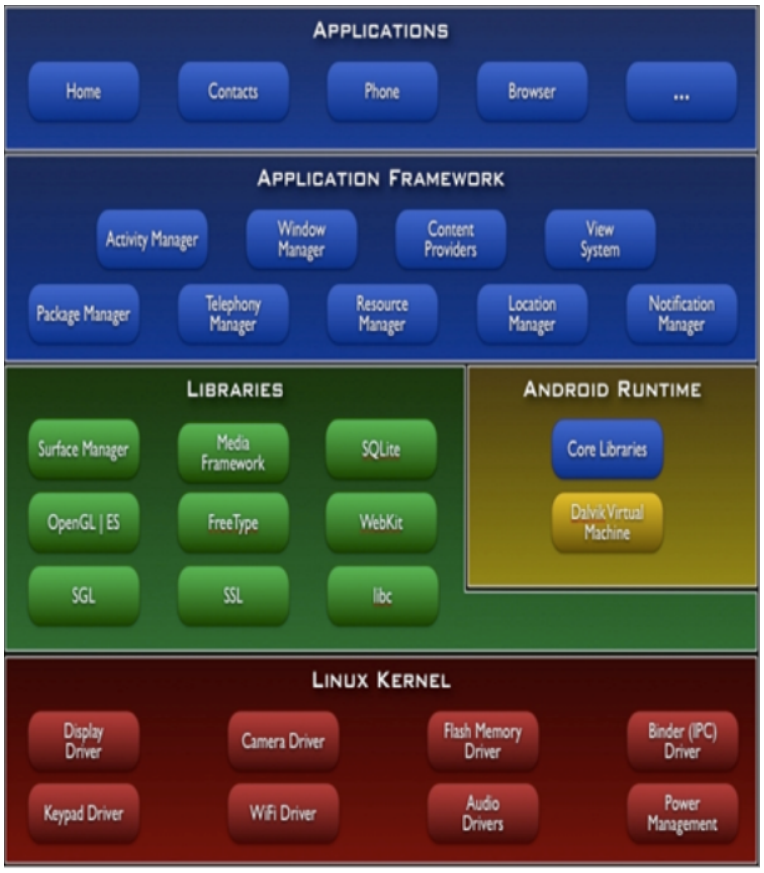


Figure 1.1: Android Operating System

1.3.1 Linux Kernel

This layer is designed as an abstraction layer between the hardware and the other layers. The Linux Kernel is version 2.6 for core system services such as security, memory management, process management, network stack, and driver model.

1.3.2 Android Runtime

Android Runtime layer consists of two parts.

- **`Core libraries`**: Most of the functionalities available in the core libraries of the Java programming language, e.g., math, text, IO, etc. are provided by the Core Libraries of the Runtime Layer.

- **`Dalvik Virtual Machine`**: Every android application is compiled into a byte code format for android's own register-based virtual machine called Dalvik. Dalvik VM executes files in the Dalvik Executable (.dex) format which is optimized for minimal memory footprint. The (.dex) files are created by the (dx) tool. This tool transfers files compiled by a Java language compiler into the (.dex) format. Moreover constants and duplicate strings used in multiple class files are included only once in the (.dex) file to preserve limited space [3].

1.3.3 Libraries

Developers can access a set of C/C++ libraries through the android application framework. These libraries are used to access hardware and database or sometimes to perform security or performance critical tasks.

1.3.4 Application Framework

Developers have full access to the same framework APIs used by the core applications. One of the important goals in android's architecture is the reuse of components; any application can expose its capabilities and any other application can use those capabilities.
All APIs are written in the Java programming language.

1.3.5 Applications

Finally, the applications layer includes a set of core applications including an Email client, Phone, Calendar, Maps, Browser, Contacts, and others. All applications are written in the Java programming language. Every new application written by developers is part of this layer.

1.4 Application Components

An important feature of android is the reusability of components. In android, applications can make use of elements of other applications (if they have sufficient permissions). For example, if the application needs to display a list of contacts which has already been developed by another application and is available to others, it can call that list to do the work, rather than developing it is own. Further, the application does not incorporate the code of the called application or link to it. The applications just start up that piece of the other application when the need arises.

To enable this feature, android starts an application process when any part (component) of an application is needed and also it instantiates the Java objects for that part. So, unlike applications in other systems, android applications do not have just one single entry point for whole application. For example they do not have a `main()` function. Rather, they are composed of some components. These components can be instantiated and run as needed. There are four types of components:

Activities: "An *activity* presents a visual user interface for one focused endeavor the user can undertake".[2] For example, an activity might display a message to the user and request a response. It might be a scrolling list of images or contacts that the user can choose from. An application might consist of one or several activities. For example, a Phone application might have an activity for dialing a number, a second one to display a list of logged calls, and other activities for displaying missed calls and so on. Functionality of an application is dependent on all of its activities. On the other hand, each activity is stand alone and independent of other activities so that it can be called by other applications.

Actually, every application has one activity which is the starter of the application. It is the first activity that the user can see when the application is launched. Moving to the next activity is done in the current activity by simply starting the next one.

Services: "A *service* does not have a visual user interface that the user can interact with, but rather runs in the background for an unlimited period of time" [2]. For example a service can be used to calculate something in the background and provide the result to every activity that is interested. All of this process is done in the background and the user can also perform any other action.

In other words, every action which needs to keep running even after leaving the screen, should be handled by a service.

A good example is a media player. To play music, initially, the media player application runs an activity displaying a list of songs. Afterwards the user selects one song, expecting it keeps running in the background even if she leaves the player application. To give this ability to the application we should implement the player as a service instead of an activity. So, after selecting a music, the activity starts a service to play the music. The music keeps playing even after the starter activity leaves the screen and the user attempts to perform some other action.

Broadcast receivers: "A *broadcast receiver* is a component that does nothing but receives and reacts to broadcast announcements"[2]. Broadcast announcements are some events which happen in the system and are received and responded by any application which is interested in them. Although the system has many broadcasts originally, applications can also initiate other broadcasts if it is needed. Furthermore, an application can have any number of broadcast receivers to answer different announcements which are important for the application. Some of the original broadcasts in the system are: announcement for changed time zone, the battery is low, a picture is taken and so on.

Content providers: "A *content provider* makes a specific set of the application's data available to other applications."[2] Every content provider extends the ContentProvider base class and is responsible to control a specific type of data. The data can be stored in different formats like a file system or an SQLLite database. Every content provider implements a set of methods so that every application can use them to save or retrieve data of the type it controls. Other applications call these methods by using a ContentResolver object and call its methods instead of calling ContentProvider methods. A ContentResolver can communicate to any content provider.

1.5 Activating Components

When there is a request that should be handled by a specific component, firstly, android starts the application process of the component if it is not running and then creates an appropriate instance of the component if it is not available. To activate a content provider we use a ContentResolver request whose target is the content provider. The other components, such as: activities, services, broadcast receivers, are activated by asynchronous messages called intents.
"Intent is an object that holds the content of the message. For activities and services, it names the action being requested and specifies the URI of the data to act on"[2], among other things. For example, it might include some information to call an activity to display contact information of a specific person. For broadcast receivers, the intent object contains the name of the action being announced. For example it might announce the interested elements that the battery charge is low.

1.6 The androidManifest.xml File

Before describing Intent filters more precisely, we should talk about Manifest.xml files. Every android application must have an androidManifest.xml file (with precisely that name) in its root directory. The manifest includes essential information that must be known by the system before running the application's code. Among other things, the manifest does the following:

- The name of the Java package for the application. For instance *package="com.android.phone"* is a tag in the androidManifest.xml file of Phone application which describes the package name of the application.

- The name of the components that the application is composed of, the classes that implement each of the components (activities, broadcast receivers,content providers, services), and the capabilities of the components. For example, which Intent message they can handle. By adding Intent filters for every Component, the android system knows what the components are and which component can handle a specific launched Intent message.

- It declares the permissions that other applications must have to gain access to application's components.
- It declares the minimum level of the android API that the application requires.
- It lists the libraries that the application must be linked against and some other duties.

1.7 Intent Filters

As it is said in previous parts, to activate a component (activity, service, broadcast receiver) an Intent object is required. Usually an Intent object contains the name of the target component. In this case, android finds that component (based on the declarations in the manifest file) and activates it.
But sometimes the target name is not explicitly mentioned in the Intent object, so android must search and find the best component to respond to the requested intent object. android handles this by comparing the Intent object to the intent filters of all components described in the Manifest file. It is feasible because described components in the Manifest file have intent filters which inform android of the kinds of intents the component is able to handle.
The following listing is an example of describing an activity along with two intent filters in a Manifest file:

Listing 1.1: Manifest file

```
< ?xml version="1.0" encoding="utf-8"?>
 <manifest . . . >
    <application . . . >
        <activity android:name="com.example.project.
           FreneticActivity"
                  android:icon="@drawable/smallpic.
                     png"
                  android:label="@string/
                     freneticLabel"
                  . . .  >
            <intent-filter . . . >
                <action android:name="android.intent
                   .action.MAIN" />
                <category android:name="android.
                   intent.category.LAUNCHER" />
            </intent-filter>
            <intent-filter . . . >
                <action android:name="com.example.
                   project.BOUNCE" />
                <data android:mimeType="image/jpeg"
                   />
                <category android:name="android.
                   intent.category.DEFAULT" />
            </intent-filter>
        </activity>
        . . .
    </application>
</manifest>
```

There are separate methods for activating each type of component by passing an *Intent*: Detailed information can be found at [2].

- ”An **activity** is launched (or given something new to do) by passing an Intent object to Context.startActivity() or Activity.startActivityForResult().” [2]
  The first activity is launched by the Home application. The next activity often is started by the previous one. If the activity expects a result back from the launched activity, it calls startActivityForResult() instead of star-

tActivity(). For example, if it starts the Contact List activity that lets the user pick a contact number, the selected number is expected to be returned. The result(selected number) is returned in an Intent object that is passed to the calling activity's onActivityResult() method. So the calling activity needs to implement this method and handles the return value as it needs.

- "A **Service** is started (or new instructions are given to an ongoing service) by passing an Intent object to Context.startService(). Android calls the service's onStart() method and passes it the Intent object."[2]
  Furthermore, it is possible to bind a component to an running service. To do this, an intent can be passed to Context.bindService() to establish an ongoing connection between the calling component and a target service. For example, if we have an activity to control the music playback by the user, the activity need to establish a connection with the music playback service. The activity would call bindService() to set up that connection, and then call methods defined by the service to affect the playback.

- "A **Broadcast** can be initiated by passing an Intent object to methods like Context.sendBroadcast(), Context.sendOrderedBroadcast(), and Context.sendStickyBroadcast() in any of their variations." [2]

  Actually, android delivers the Broadcasts to the system, and every interested broadcast receiver, receives that by its onReceive() method.
  Interested receivers must implement the onReceive() method. They act corresponding to the information they receive. For example in the following code the activity implements a receiver for the *SCREEN_OFF* announcement.

—— Java ——

```java
private BroadcastReceiver mBroadcastReceiver =
                              new BroadcastReceiver() {
    public void onReceive(Context context, Intent intent) {
        if (Intent.ACTION_SCREEN_OFF.
                        equals(intent.getAction())) {
             finish();
              }
         }
    }
```

—— Java ——

1.8 Activity Lifecycle

The following summarizes some important points. Detailed information can be found in [1]. Activities are the most used components in an android application also they are the only components with which the user can interact with. Every application might consist of one or several activities. Usually one activity is marked as the first activity that should be presented to the user when the application is launched. The next activity can be started by the current activity. Activities in the system are managed as an *activity stack*. New started activities are always placed on the top of the stack and they stay running, until a new activity is started. The previous activity always remains below the new activity in the stack and also they will not come to the foreground again as long as a new activity exists. An activity is essentially in one of the following four states:

1. **`active`** `or` **`alive`**: When an activity occupies the foreground of the screen.

2. **`paused`**: When an activity has lost the focus but it is still visible. It happens when an new non-full size or transparent activity has focus on top of the stack. In this state activity is completely alive, but can be killed by the system in extreme low memory situation.

3. **`stopped`**: When an activity is completely hidden by another activity. It still preserves all state and member information, however, it is no longer visible to the user. It is killed by the system when the memory is needed.

4. **`finishing`**: When an activity is *paused* or *stopped* the system can drop the activity from the memory by killing its process or asking it to finish.

The diagram in Fig. 1.2 describes the whole lifecycle of an activity and the paths an activity may takes between states. The colored ovals are major states of an activity.

All methods in a rectangle can be overridden by the user to perform appropriate tasks when the activity transitions between states, however, super classes should be called up when implementing these methods.

The entire life time of an activity starts from the first call to onCreate() method until a single final call to onDestroy() method. The onCreate() method is called when the activity is first created. It should be implemented by all activities. It is where most of the initialization should go, e.g., create views, bind data to lists, etc. In onDestroy() method all remaining resources will be released. The visible life time of an activity starts from a call to onStart() until a call to onStop(). During this time, the activity is visible to the user though it might not be in the foreground. The needed resources can be maintained as long as the activity is visible.

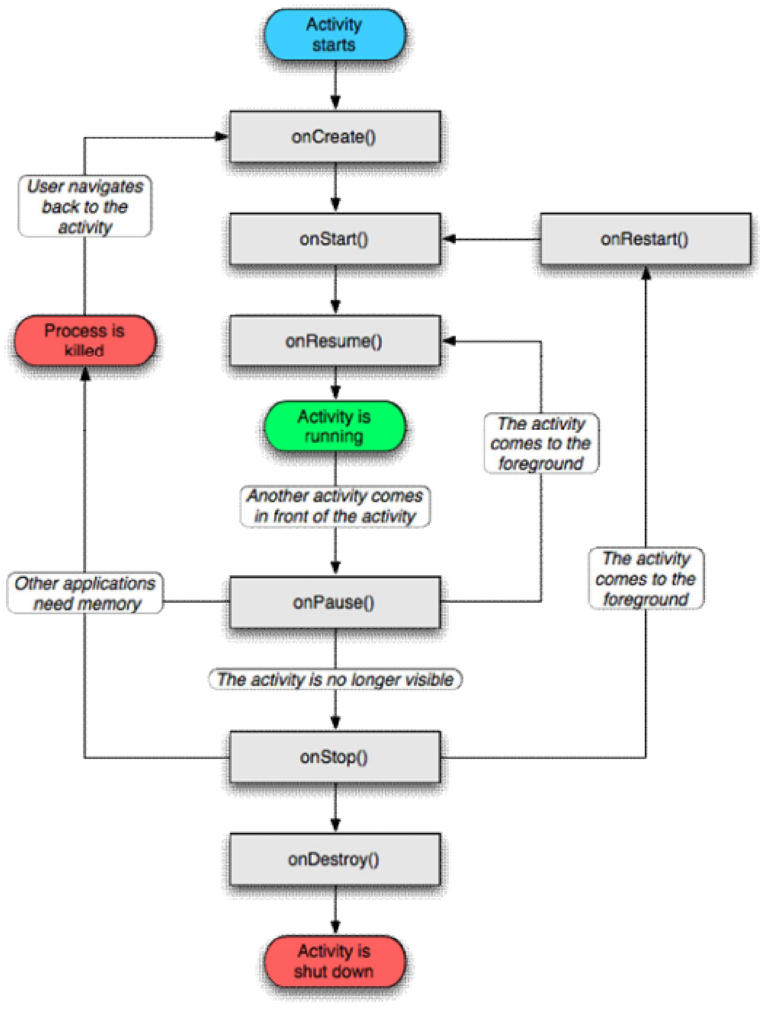

Activity starts
onCreate()
User navigates back to the activity
onStart()
onRestart()
Process is killed
onResume()
Activity is running
The activity comes to the foreground
Another activity comes in front of the activity
The activity comes to the foreground
Other applications need memory
onPause()
The activity is no longer visible
onStop()
onDestroy()
Activity is shut down

Figure 1.2: Activity Lifecycle

The foreground life time of an activity starts from a call to onResume() until a call to onPause(). During this time activity is on the top of the *Activity stack* and is in front of all other activities and the user can interact with.

Chapter 2

Specification and Verification Concepts

To gain confidence that critical modules and classes in android will meet their desired functionality, we use the formal specification language JML and the tool KeY to prove the correctness of the code with respect to its formal specifications. In this chapter, we introduce first, JML as a formal specification language and the KeY tool as a prover. Later on, we use them to specify and verify a small android application.

2.1 JML (Java Modeling Language)

JML (Java Modeling Language) is a language tailored to Java for formally specifying the sequential behavior and interfaces of Java programs. JML follows the Design by Contract methodology.

"JML specifications are usually annotated to which entities they refer. They are written as comments starting with the symbol /*@ and also optionally ending with @*/. Because they are the same as Java comments, they are ignored by the Java compiler and they are just recognized by tools which handle JML like the JML checker"[4].

2.1.1 Precondition And Postcondition

JML contracts state what is guaranteed under which conditions. It states that if the module is called in any state which satisfies the precondition, then the postcondition is true in any terminating state of the method. For example in a cell phone that needs an unlock password to unlock the cell phone, the unlock method is specified in this way:

Precondition: cell phone is locked and the user is not authenticated, entered password by the user is correct.
Postcondition: the cell phone is unlocked and the user is authenticated.

— JML (2.1) —

```
/*@  public normal_behavior
  @  requires  cellPhoneIsLocked && !authenticated;
  @  requires  enteredkey  == correctKey;
  @  ensures  !cellPhoneIsLocked && authenticated;
  @  assignable cellPhoneIsLocked,authenticated;
  @*/
public boolean checkKey(int enteredkey){...}
```

— JML (2.1) —

As we know every thing between /* and */ is invisible by Java and it is recognized just with JML.
JML annotations come together with visibility modifiers which are helpful to formulate sensible contracts. The public keyword here means the same thing in JML as it does in Java language. It indicates that this JML specification is visible to all other classes in the application. Moreover public specifications can only refer to public methods and fields.
The `normal_behavior` states that this specification describes the state in which the method terminates normally without throwing any exception.
The JML keyword `requires` is used to describe the precondition. As it is mentioned earlier, a precondition is a condition that must be satisfied (by the caller) before calling a method.
To state a precondition after `requires,` a boolean typed JML expression has to be given. Actually, all side-effect free Java expressions are JML expression plus certain extensions like ==> or quantifiers \forall, \exists etc.
In our example, the `requires` clause states that to unlock the cell phone by this method, the cell phone should be locked and a correct password is needed. These are described in JML expressions.

On the other hand, the JML keyword `ensures` is used to describe the postcondition. The postcondition of a method specifies the responsibilities of a method. It states that, if precondition is fulfilled in normal termination state of the method, the postcondition should be true.
In our example, it means, if the user enters a correct password, the cell phone should be unlocked in termination state.

This is described in the JML expression after `ensures.`Also a boolean typed JML expression has to be given.

2.1.2 Class Invariant

Invariants are conditions that must be true on entry and exit of every method of a class. Although invariants are declared on the class level not in the method level, they will be added as precondition and postcondition of every method in the class implicitly. So, they should be held while the object is alive. For example the following code is an invariant:

— JML (2.2) —

```
//@ public invariant correctKey>0;
```

— JML (2.2) —

It means that `correctKey` field should be greater than zero while as the object is alive.

2.1.3 Model And Ghost Fields

Sometimes it is convenient to introduce an extra field, only for the purpose of the specification.
Actually, operation contracts and instance invariants in JML may only talk about instance and static fields occurring in the Java program they annotate. Since instance fields may only occur in classes and not in interfaces, so to specify interfaces we need to introduce extra fields.
To do this, JML introduces `model` and `ghost` fields. `model` fields are like normal fields in a Java class with the difference that,

- they only exist in the JML specification and they are ignored by Java.
- they are abstract fields.
- we cannot assign to them.
- JML uses a ”represents” clause to associate model fields to a concrete implementation.
- they change their value whenever the representation changes.

Example:

— JML (2.3) —

```
/*@ public model  int key;
  @  private represents key <- correctkey;@*/
```

— JML (2.3) —

A `ghost` field also is like a normal field with the difference that:

- We can assign to it, using `set,` in JML annotations.
- A `set` statement can be used at any place where also a Java statement could occur.

Example:

— JML —
```
//@ public ghost int key;
   ...SOME JAVA CODE HERE...
//  @ set key=1234;
```
— JML —

In the above code, first a ghost variable of type `int` is declared. Afterwards in the middle of code between Java codes, it is assigned to "1234".

2.1.4 Model Method

`model` methods are JML methods that can only be used in specifications. They are declared in Java comments and cannot be used in regular Java implementation code. Like `model` fields, they are declared only for the purpose of specification. They can be used as auxiliary methods to compute something or perform something else just for the purpose of specification. The following is an example:

— JML (2.4) —
```
 /*@ public model nullable String[] getEmergencyNumbers() {
   @ ...
   @ return emergency;}
```
— JML (2.4) —

the following code is an example of using a `model method` in specification code:

— JML (2.5) —
```
@ invariant (\forall int i;i>=0 && i<getEmergencyNumbers().length;
                                    getEmergencyNumbers()[i]!=null );
```
— JML (2.5) —

2.1.5 Assignable Clause

The `assignable` clause is used to mention all fields that can be modified by a method. For example in the listing (2.1), just `cellPhoneIsLocked,authenticated` are allowed to be modified after termination of the method. On the other hand a method allows modifying a field *Loc* if any of the following factors are true[5]:

- *Loc* is mentioned in an `assignable` clause.
- A location mentioned in an assignable clause depends on *Loc* (e.g., model fields and data groups).
- *Loc* is not allocated once the method started execution(Loc is a local variable).
- *Loc* is a formal parameter or a local variable of the method.

Using \`nothing` (e.g., `assignable`\nothing) states that none of the fields are allowed to change.

2.1.6 Inheritance In Specification

JML specifications are inherited by subclasses and classes implementing interfaces. The JML keyword *also*, at the beginning of the specification indicates that a specification is combined with specifications inherited from ancestor classes and from interface being implemented or as simple conjunction. So an invariant of a class is inherited by all its subclasses, on the other hand an operation redefined in a subclass does not inherit the operation contract from the superclass.

2.1.7 Some JML Keywords [4]

`Operators:`

- **`result`**: It is used in `ensures` clause of a non-void method. Its value is the value returned by the method.
- **`forall`**: It is an universal quantifier. For example:
  `(\forall int i,j; 0 <= i && i<j && j<10;a[i]<a[j])` says that the values a[0] ... a[9] are sorted. The body must be a type of boolean.
- **`exists`**: It is an existential quantifier. For example:
  `(\exists int i; 0 <= i && i<10; a[i]==20)` says, there exists an element in array `a` that equals to 20. The body must be a type of boolean.
- \**`old`**: An expression of the form `\old(Expr)` refers to the value that the expression Expr had in the pre-state of a method.

`Modifiers:`

- **`nullable`**: Because any member field, formal parameter, method return type and bound variable whose type is a reference type, in JML are defined as `non-null` implicitly, we use this modifier to declare them as `nullable`, when it is necessary. For example, if there is an member field which can have a null value, we add `nullable` modifier in its declaration.

- **`pure`:** `pure` is only applied to methods and constructors. A pure method has an implicit specification that does not allow any side-effects and its specification has the following clause implicitly :
  `assignable \nothing`

- **`object_creation(the class name of the created object)`:**
  This keyword is used to introduce any new created object, even those that are local objects. Actually this keyword does not belong to the official JML standard but will be interpreted by KeY. It can be used in the following way: If an object of type T is allowed to be created, the term `object_creation(T)` has to be added to the assignable-clause. Actually this keyword is useful for KeY system for verification process which is described in the following section.

2.2 What Is KeY?

"The KeY system is a formal method tool that integrates design, implementation, formal specification and formal verification as seamlessly as possible. Its intention is to provide a platform that allows close collaboration of conventional and formal software development methods. At the core of the system there is a novel theorem prover for the first-order Dynamic Logic for Java with a user-friendly graphical interface" [6]. The KeY-Tool aims to formally prove that the implementation actually satisfies the specification written in JML or OCL(Object Constraint Language).
The foundation of the KeY system is based on induction plus symbolic execution which are the verification paradigm of the KeY system. The following principles describe more details:

- A deductive verification system, meaning that its core is a theorem prover, which proves a formula of a suitable logic. The KeY approach employs a logic called Java Card DL, which is an instance of *Dynamic Logic*(DL).[7] Dynamic Logic has a rich type system and a sequent calculus. This predicate logic can easily be used to describe and reason about data structures, the relations between objects and the values of variables, in short: about the states of (Java) programs. Moreover DL extends the first-order logic and the calculus so that we can describe and reason about the behavior of programs, which requires to consider not just one but several program states.[6] Syntactically, DL extends full first-order logic with two additional operators: the diamond $\langle . \rangle$. and the box $[.]$. modality.

In both cases, the first argument is a program and the second argument is another DL formula.
For example:

$$\langle p \rangle \phi$$

is true in a state s if execution of p terminates normally when started in s and results in a state where ϕ is true. As for the other operator, a formula

$$[p]\phi$$

is true in a state s if execution of p, when started in s, does either not terminate or result in a state where ϕ is true.[8]
In explanation of state s we can say: a state can be seen as current snapshot of the memory when running a program. It describes the value of all variables and fields of a program. A formula in JavaCardDL is evaluated in such a state. For example:

$$x \doteq 0 \mathrel{-\!\!>} \langle x++; \rangle x \doteq 1$$

is a Java DL formula. It states that, if $x \doteq 0$ is fulfilled before executing the program sequence $x++;$, then the program sequence will terminate and $x \doteq 1$ will hold after its execution. The statement that

$$x++;$$

will terminate, is implied by the usage of the operator $\langle\rangle$. If no termination is required, the operator [] can be used instead[6].
Deduction with the KeY-Prover is based on a sequent calculus for a Dynamic Logic for JavaCard (JavaDL). A sequent has the form:

$$\phi_1, ..., \phi_n \Rightarrow \psi_1, ..., \psi_m (n, m >= 0)$$

where the ϕ_i and ψ_i are JavaDL-formulas. The formulas on the left-hand side of the sequent symbol are called antecedent and the formulas on the right-hand side are called succedent. The intuitive meaning of a sequent is: if we assume all formulas $\phi_1, ..., \phi_n$ hold, then at least one of the formulas $\psi_1, ..., \psi_m$ holds. The order of formulas within $\phi_1, ..., \phi_n$ and within $\psi_1, ..., \psi_m$ does not matter.

- The actual verification process in the KeY system can be viewed as *symbolic execution* of source code.
  Symbolic execution interprets a program similar to a normal interpreter, but instead of using concrete values for variables or fields, it uses symbolic values (terms). For example, to find out whether the sequent

$$\Rightarrow \langle o.next.prev = o; \rangle o.next.prev \doteq o$$

is valid, at first, the calculus rules transform it into an equivalent but longer sequence of statements:

$$\Rightarrow \langle ListEl\ v; v = o.next; v.prev = o; \rangle o.next.prev \doteq o$$

This way, we have reduced the reasoning about the complex expression

$$o.next.prev = o$$

to reasoning about several simpler expressions.
This process is called unfolding, and it works by introducing fresh local variables to store intermediate computation results. Now, when analyzing the first of the simpler assignments (after removing the variable declaration), one has to consider the possibility that evaluating the expression *o.next* may produce a side effect if o is null (in that case an exception is thrown). However, it is not possible to unfold *o.next* any further.
Something else has to be done, namely a case distinction. This results in the following two new goals:

$$!(o \doteq null) \Rightarrow \{v := o.next\}\langle v.prev = o; \rangle o.next.prev \doteq o$$

$$o = null \Rightarrow \langle\ \text{throw new NullPointerException();}\ \rangle o.next.prev \doteq o$$

The validity of the simplified formulas is specified by calculus rules.
Moreover, unbounded loops and recursion are either handled by induction over data structures occurring in the verification target or by specifying loop invariants and variants.[7]

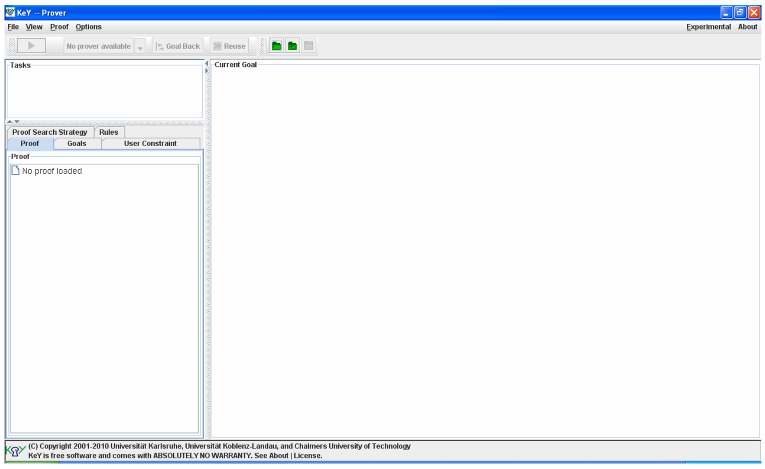
In the KeY system, proof obligations are formulas that have to be proved valid. They are designated formulas occurring in the root sequent of the proof. Actually they are generated to prove that the methods and classes respect certain parts of their specification.The Proof Obligation Browser in the KeY system provides a selection of proof-obligations to verify different aspects of a method contract. A method contract for a method m of a class C consists in general of a *precondition*, *postcondition*, *assignable*.[8] The most standard proof obligation is total correctness of a method implementation with respect to its contract. For more information about foundation of the KeY system and its usage please refer to [6].

Using KeY

In this section we introduce the user interface of the KeY system briefly. In order to use the KeY system as a verification tool, we first need to run the KeY prover. This is done either by:

- Start it directly via Java Web Start from the KeY website[9]
- Download the pre-compiled bytecode version from the KeY website[9]

After starting the KeY system, we see the following user interface: As we can see



(C) Copyright 2001-2010 Universität Karlsruhe, Universität Koblenz-Landau, and Chalmers University of Technology
KeY is free software and comes with ABSOLUTELY NO WARRANTY. See About | License.

Figure 2.1: KeY Prover

the interface is divided to three panes:

- a small pane on the top left which lists the proof obligations loaded
- a tabbed pane on the bottom left containing tabs for:
  - the proof history (the tab Proof),
  - a list of currently open goals (the tab Goals),
  - a pane for choosing the rule application strategies (the tab Proof Search Strategy),
  - a summary of all available rules (the tab Rules) and finally
  - a pane for some user specific constraints (the tab User Constraint).
- a big pane on the right which shows the sequence representing the currently selected goal.

After starting the KeY system, in order to verify a piece of code, first, we should load a proof obligation. A proof obligation can be loaded by selecting the menu item Load in the menu File and choosing the base folder of the Java program to be verified. After KeY has parsed the program as well as its JML specification, a dialogue will appear which lists all available kinds (EnsuresPost, RespectsModifies, PreservesInv, etc.)(see Fig. 2.3) of proof obligations for all methods and all classes. As we know a proof obligation is a formula that has to be proved valid.

After selecting a proof obligation type, and its parameter(the method contract and the set of assumed invariants), the proof obligation will be created as a Java Card DL formula and will appear in the pane on the right. Later on we can start the automatic rule application to transform the formula under consideration by clicking on the ▶ button either in the toolbar or in the tab Proof Search Strategy.
Of course, we can apply rules manually when it is needed. The KeY system provides an easy to use, context sensitive highlighting and selection mechanism. When the mouse is positioned over a function or predicate symbol, the corresponding term or formula will be highlighted. Clicking on such a symbol will open a context menu which lists all rules which are (manually) applicable to the term or formula. The selection of a rule from the context menu will apply this rule.

2.3 Specifying and Verifying a Sample Android Application

Before specifying some critical APIs in android, there is a small application (Login application) which can be a good start for getting familiar with android applications and JML specifications and also the usage of the KeY system.
To do this, we specify first the sample application with JML and then we verify that with the KeY system.
Fig. 2.2 shows the user interface of the Login application:

2.3.1 Specification with JML

In this small android application, the program asks the user to enter a correct password and then to press the "OK" button. If the entered password was true, the application shows the message "the password is correct", otherwise "the password is not correct".
What is important for us in this application is the correctness functionality of the "OK" button when it is clicked. In other words, the user expects a correct reaction when the "OK" button is clicked. So we try to specify the methods that handle the "OK" button with JML, as completely as possible, afterwards we verify them with the KeY system.
The implementation of the application is done by `EnterPassword` and `Listener` classes.
The `EnterPassword` class is an activity component that the user can interact with. It is where the user interface is implemented and initialized, also it has to extend the Activity base class.
On the other hand, `Listener` class is responsible to perform the appropriate

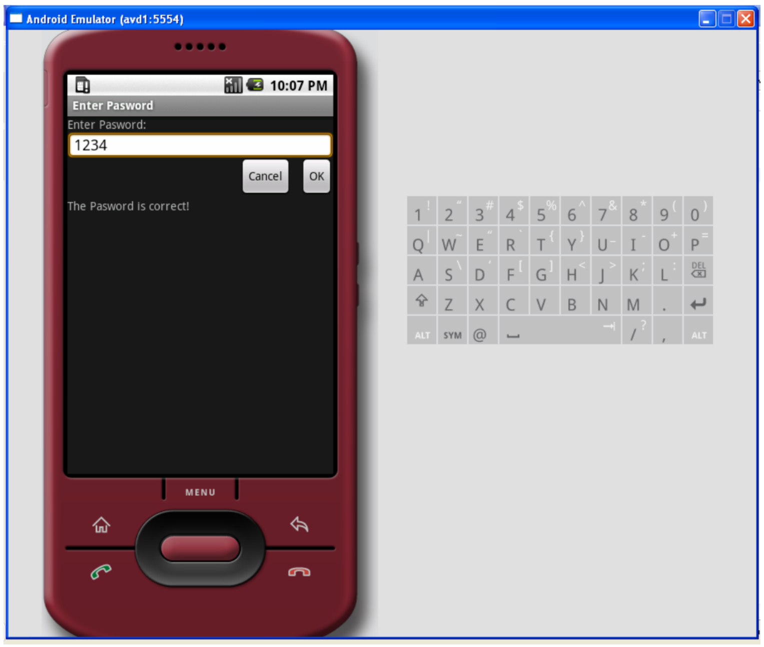


Figure 2.2: Log In Application

action when the "OK" button is clicked by the user. `Listener` class implements `OnClickListener` interface, so it has to implement the `onClick()` method. This class implements reaction, in response to the user click on the "OK" button.
First, we explain the `EnterPassword` activity which starts the application, later on we focus on the `onClick()` and two other methods in the `Listener` class and try to specify them with JML.
The `EnterPassword` class is implemented as follows:

— Java + JML (2.6) —

```
public class EnterPassword extends Activity {
        ...
        /** Called when the activity is first created. */
        public void onCreate(Bundle savedInstanceState) {
           super.onCreate(savedInstanceState);
           ...
           Listener listener=new Listener(tempInputtext,tempText);
           okButton.setOnClickListener(listener);
     }
```

— Java + JML (2.6) —

As we can see in the above code, the `onCreate()` method is implemented in the beginning of the class. As is mentioned earlier in section 1.8, every activity has to implement the `onCreate()` method. This method is called when the activity

starts. This is the place where most initialization should go.
`setOnClickListener()` is called in the body of `onCreate()`. This is where we assign a `OnClickListener` to "OK" button.
The `Listener` class is the place where we perform appropriate action in response to a click on the "OK" button. The following code is the implementation along with the specification of `Listener` class.

── JML ──
```
public class Listener implements OnClickListener
{
//@ public ghost boolean  userHasLoggedIn=false;
 private /*@ spec_public  @*/ int passCode=1234;
 private /*@ spec_public  @*/ TextView tempText;
 private /*@ spec_public  @*/ EditText tempInputtext;
 private /*@ spec_public  @*/ int textLength;
```
── JML ──

As it is obvious, all fields are defined as `spec_public` because they are private and cannot be used in the specification of a public method. By using `spec_public` we state that they are `public` just for the purpose of specification. Then we can use them in a `public` specification.
The following code is the specification and Java code for the `onClick()` method, the place where the user enters a password and clicks "OK" button and then expects to see a message accordingly. Further, if the user enters a wrong password for three times she can not log in anymore and she sees an alert message (other functions can be implemented here instead of showing a message).

── JML (2.7) ──
```
/*@ public normal_behaviour
  @
  @ requires tempInputtext.getText().length()>0 &&
  @                  incorrectPINEntered<3 && !userHasLoggedIn;
  @ ensures true ;
  @ assignable userHasLoggedIn,incorrectPINEntered;
  @
  @ also
  @ public normal_behavior
  @
  @ requires tempInputtext.getText().length()==0 &&!userHasLoggedIn;
  @ ensures !userHasLoggedIn ;
  @ assignable \nothing;
  @
  @ also
```

```
16   @ public normal_behavior
17   @
18   @ requires incorrectPINEntered>=3 && !userHasLoggedIn ;
19   @ ensures !userHasLoggedIn ;
20   @ assignable \nothing; @ */
21
22  public void onClick(View v)
23    {
24    if( tempInputtext.getText().length()==0 )
25        {
26        //tempText.setText("Please Enter The Password!");
27            return;
28       }
29    int intPass;
30    if (incorrectPINEntered < 3) {
31         intPass=getPassword(tempInputtext);
32         if (isPasswordCorrect(intPass)) {
33                incorrectPINEntered = 0;
34          } else {
35                incorrectPINEntered++;
36
37              }
38        }
39    else
40        {
41         //tempText.setText("You have entered the password
42          //                      incorrectly more than 3 times!");
43           return;
44        }
45    }
```

JML (2.7)

The specification code in the first line states the situation that the method terminates in a normal behavior without any exception. The next section in line 3 is the first `required` clause, it states the first precondition of the method. It represents the condition that the length of input string "password" is greater than zero and the incorrect entered password by the user does not exceed the maximum number(3) and the user has not logged in yet.
The next clause in line 5 is the `ensures` clause or postcondition of the method regarding the first precondition. It states that if the requirements(precondition) are fulfilled after termination of the method, here, we do not expect any specific result so we just say `true`.

The `userHasLoggedIn` field is a `ghost` field that is declared as an auxiliary field for the purpose of specification and it is used to indicate if the user has logged in. We set this field to `true` whenever the user has logged in, otherwise it stays `false`. This field is assigned to `true` in the body of the `isPasswordCorrect()` method.
The next clause is the `assignable` clause in line 6. It states the locations that can be modified after termination of the method: `userHasLoggedIn, incorrectPINEntered` .
The `also` JML keyword in line 8 describes the other condition of executing the method. It states that if the password field is empty and the user has not filled it in yet and just clicks the "OK" button, after termination of the method, the `userHasLoggedIn` field, should be false and no field may be changed during execution.
The last precondition of the method starts at line 15. It states the condition that the incorrect entered password by the user exceed the maximum number(3) and the user has not logged in yet. The postcondition of this state in line 19 shows that after termination of the method `userHasLoggedIn` should be false and no field may change.
In lines 26 and 41 the code is commented out just for the purpose of the verification, because the KeY system did not support string literals when this thesis was prepared(it does in the meantime) .

The next method that we are going to specify is `getPassword()`. Actually this method is used to simplify the verification process and its specification is very simple. We just expect the result of the method to be equal or greater than zero.
The specification and implementation of the `getPassword()`method is:

— Java + JML (2.8) —

```
/*@ private normal_behavior
  @ requires true;
  @ ensures \result >= 0;
  @ assignable \nothing;
  @*/
private int getPassword(EditText input) {
    return Integer.valueOf(input.getText().toString()).intValue();
  }
```

— Java + JML (2.8) —

The last candidate for specifying is the `isPasswordCorrect()` method The following code is the specification and implementation of this method. Later on we try to describe it in detail.

— Java + JML (2.9) —

```
1  /*@ public normal_behaviour
2    @ requires pwd==passCode && !userHasLoggedIn;
3    @ ensures userHasLoggedIn && \result==true;
4    @ assignable  userHasLoggedIn;
5    @ also
6    @ requires pwd!=passCode && !userHasLoggedIn;
7    @ ensures !userHasLoggedIn && \result==false;
8    @ assignable \nothing;
9    @*/
10
11   public boolean isPasswordCorrect(int pwd)
12      {
13       if(pwd==passCode)
14         {
15          //tempText.setText("The Password is correct!");
16          //@ set userHasLoggedIn=true;
17           return true;  }
18       else
19        {
20         //tempText.setText("The Password is not correct!");
21         return false; }
22 }
```

— Java + JML (2.9) —

As we can see in the implementation, if the entered password by the user is correct, a message is shown and the `ghost` field `userHasLoggedIn` is assigned to `true`. Otherwise, it is assigned to `false`.
The specification of the method describes that in a normal termination, if entered password is correct, and the user is not logged in before execution of the method, after termination, the method should return `true` and also `userHasLoggedIn` should be `true`.
In the assignable clause we stated that only `userHasLoggedIn` may be changed after termination of the method.
In the other case, when the entered password is not correct and the user is not logged in earlier, the return value should be `false` and the `assignable` clause states that nothing may change after termination of the method.
Again in lines 15 and 20 the `tempText.setText(...)` statements are commented out as KeY only recently implemented support for treatment of Strings, but not yet when the presented work had been carried out. In other words, al-

though they are not commented out in the running mode of the application, we have to comment them out when we go to the verification phase in next section.

2.3.2 Verification With KeY

To verify "android applications" by the KeY system, first, the KeY system needs to know about the signature of the android APIs that we have used in our application. So, we need to create a directory containing all the used and needed android library files in our application. We name this directory as *Stubs*.
To create *Stub* files, there is a prototype generator tool: *Stubmaker*.
When we are going to use android, we should download SDK somewhere on the system. There is an "android.jar" file in this folder which can be used to generate *Stub* files. We add this file as input to this tool at the command prompt.
Further there are some options in this tool so that we can restrict generated "Stubs" only to those which are actually needed (Classes that we used in our application, not all android's classes). The "Stubs" should then be put under "myapp/Stubs/".

As the android libraries(android.jar) are compiled to bytecode, the inner and anonymous classes are compiled away into external classes. In other words, android compiles Java classes so that every inner class or anonymous class is compiled away into an external class.
For example, if there is an inner class B in class A, android creates two compiled classes which one is the main class A and another is A$B. It also creates external classes for anonymous classes as well.
So, the android.jar file not only consists of the main library classes but also the extra created classes for inner and anonymous classes.
On the other hand, the *stubmaker* does not know that A$B type of files are inner or anonymous classes of the main classes and creates instead normal Java source classes for them. This is wrong and will cause a problem for the verification and we must fix it.

The next step is to create a "myapp.key" file which is called *problem file*. That is a text file containing the path of our source files and android library classes which are used in our application(*Stubs*).
Afterwards, we load this file in the KeY system.

The following detailed steps should be taken:

1. Creating a directory for *stubs*, which contains all used and needed android library files. We name this directory *Stubs*. These library files consist of

method signatures of all classes and super classes which we used in our application. Furthermore, we do not need the actual implementation of the methods for the purpose of the verification, instead only contracts should be used if necessary.
For example, assume in the application we are using an android class: "android.app.activity". Then we need to create the *Stub*:

— Java (2.10) —

```
package android.app;
public class Activity extends <superclass> implements <interfaces>{
  ...
  public void method1 (T arg1,..., T argN);
  ...
  public void methodM (T arg1,..., T argK);
  ...
}
```

— Java (2.10) —

and put this "Stub" under "myApp/Stubs/android/app"
Afterwards, We need to create a similar "Stub" for all its super classes and interfaces that are not yet known to the KeY system.
If we use one of these methods in the implementation of the application, we should also specify the used method.

Because the *stubmaker* cannot handle inner classes correctly, after creating "Stub" files we need to change some part of these files in the following way: There are several Java files named something like

$$<classname1>\$<classname2>.java$$

these are inner classes, to fix this we have to open the file

$$<classname1>.java$$

and then to inline the code in

$$<classname1>\$<classname2>.java$$

After inlining it, we should delete the file

$$<classname1>\$<classname2>.java.$$

Further, we have to go searching through all the "Stubs" and if we find somewhere (method parameter type, field type etc.) a type name called <cl1>$<cl2>replace the name by the real name <cl1>.<cl2>

2. Creating a ”name of application”.key file. This contains path of ”Stubs” and source files. Here the source files are under ”srcKey” directory. This directory contains all files under ”src” directory and ”gen” directory.
   For example: the file ”myApp/myApp.key” would then look as the following code:

```
\bootclasspath "Stubs";
\javaSource "srcKey";
\chooseContract;
```

Creating ”Stub” Files For Android Libraries

To create ”Stub” files, there is a prototype generator tool: *Stubmaker*. Once we are going to use android, we should download SDK somewhere on the system. There is an android.jar file in this folder which can be used to generate ”Stub” files. We add this file as input to this tool in command prompt:

```
java -jar .../stubgenerator/stubmaker.jar Stubmaker -expand -d Stubs/ -seed
android.jar android.jar
```

Further there are some options in this tool so that we can restrict generated ”Stubs” only to those which are actually needed (Classes that we used in our application, not all android’s classes).

2.3.3 Verifying Enterpassword With KeY

After starting the KeY system (version 1.7.1786), we choose File→Load, then we select the ”Enterpassword.Key” file from ”select file to load proof or problem” dialogue. The KeY system loads all program files including the provided JML specifications automatically.

Afterwards, the ”Proof Obligation Browser” (see Fig. 2.3) lists all packages, classes/interfaces and methods of the project to be verified in a tree structure similar to standard file managers. In the left pane we can choose the class and the method which we are going to prove and in the right pane we can choose different aspects of the method specification to prove.
As Fig. 2.3 shows there are different options in the right pane to select. They are described briefly as follows, for more information please refer to [6, ”Proof obligations”]

- `StrongOperationContract`: It generates a formula used to prove that our contract is strong enough to ensure that all invariants of a given set of invariants INV are satisfied after return of the method if our method is

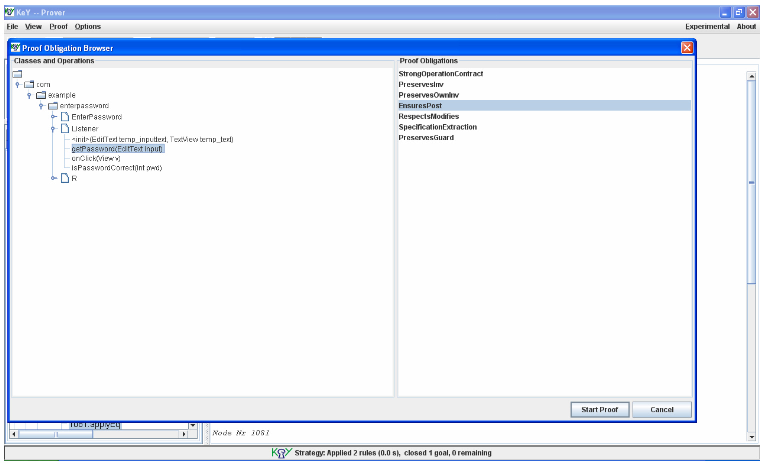


Figure 2.3: Proof Obligation Browser

called in a state where all invariants of the given set of INV are held. It tries to prove preservation of invariants without the need to symbolically execute the method body, but approximates the final state by reusing the assignable clause. If the post condition is not strong enough; we will have to use the proof obligation "preservesInv" to prove if the method preserves the invariants.

- `PreservesInv`: It generates a formula used to prove that our contract preserves validity of a given set of invariants INV. In other words, we try to prove that, if our method is called in a state where its precondition and all invariants in INV hold, then all of the invariants in INV are also valid in the method's final state.

- `PreservesOwnInv`: It is a special case of PreservesInv where set INV contains exactly all invariants of our class.

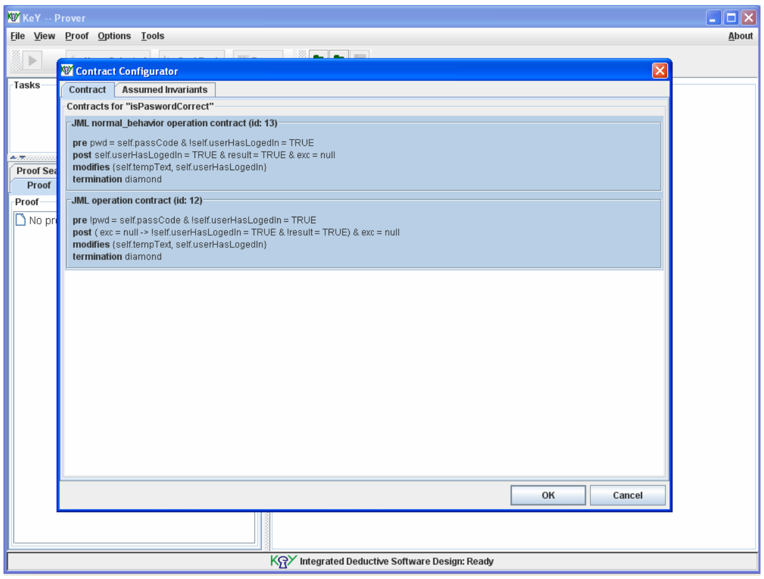
- `EnsuresPost`: It generates a formula used to prove that if our method is called in a state where the precondition and a given set of invariants INV is satisfied, then in the method's final state postconditions holds and optionally the method terminates. Furthermore the set of additional invariants is user customizable and can be selected in the "Contract Configurator" dialogue.

- `RespectsModifies`: It generates a formula used to verify the assignable/-modifies clause.

In this example, we want to verify that the "OK" button has a correct reaction when it is clicked by the user. The method `onClick()` implements this functionality. As explained before the method itself makes use of the helper methods:

`isPasswordCorrect()`and `getPassword()`. So we first verify the latter methods and use the contracts then for the verification of the `onClick()` method.
On the other hand, we need to prove the *EnsurePost*, *RespectsModifies* aspects of the method specifications `isPasswordCorrect()`, `onClick()`, `getPassword()`, to prove correctness of the `EnterPassword` class .

So, as it is illustrated in Fig. 2.3, we choose `isPasswordCorrect()` in the left pane, and `EnsuresPost` in the right pane. After clicking "Start Proof" button another dialogue is shown to the user. Fig. 2.4 shows the "Contract Configuration" dialogue.
In this dialogue we can choose the contracts which we are going to prove. Addi-



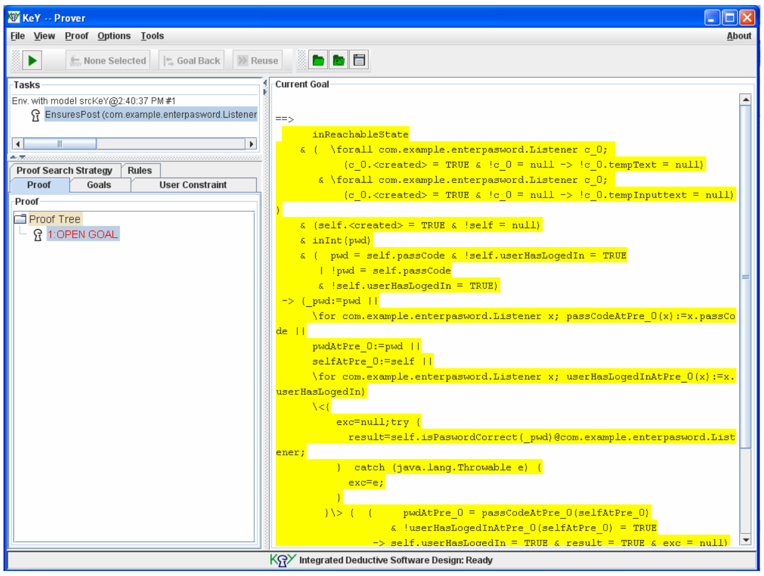
```
Contracts for "isPaswordCorrect"
JML normal_behavior operation contract (id: 13)
pre pwd = self.passCode & !self.userHasLogedIn = TRUE
post self.userHasLogedIn = TRUE & result = TRUE & exc = null
modifies {self.tempText, self.userHasLogedIn}
termination diamond
JML operation contract (id: 12)
pre !pwd = self.passCode & !self.userHasLogedIn = TRUE
post ( exc = null -> !self.userHasLogedIn = TRUE & !result = TRUE) & exc = null
modifies {self.tempText, self.userHasLogedIn}
termination diamond
```

Figure 2.4: Contract Configuration

tionally, as is mentioned earlier, a set of invariants INV can also be assumed to be proved. This set of invariants INV, can be changed by selecting or deselection desired invariants in the "Assumed Invariants" page.

Without any change we just click on "OK" button, the obligation loads as it is shown in Fig. 2.5.

As we can see in Fig. 2.5 the loaded proof obligation is an implication, with *inReachableState* acting as basic condition under which the rest of the proof obligation must be true. The predicate *inReachableState* restricts the states to those reachable by any Java computation.
For instance, *inReachableState* implies that a created object can only reference other created objects or null.
The remaining proof obligation starts with some quantifiers and updates. Thereafter, we have an implication basically saying that the (translated) requires part,



Env. with model srcKeY@2:40:37 PM #1
EnsuresPost (com.example.enterpasword.Listener

```
==>
      inReachableState
    & (  \forall com.example.enterpasword.Listener c_0;
             (c_0.<created> = TRUE & !c_0 = null -> !c_0.tempText = null)
       & \forall com.example.enterpasword.Listener c_0;
             (c_0.<created> = TRUE & !c_0 = null -> !c_0.tempInputtext = null)
)
    & (self.<created> = TRUE & !self = null)
    & inInt(pwd)
    & (  pwd = self.passCode & !self.userHasLogedIn = TRUE
       | !pwd = self.passCode
       & !self.userHasLogedIn = TRUE)
 -> {_pwd:=pwd ||
      \for com.example.enterpasword.Listener x; passCodeAtPre_0(x):=x.passCo
de ||
      pwdAtPre_0:=pwd ||
      selfAtPre_0:=self ||
      \for com.example.enterpasword.Listener x; userHasLogedInAtPre_0(x):=x.
userHasLogedIn}
      \<{
          exc=null;try {
            result=self.isPaswordCorrect(_pwd)@com.example.enterpasword.List
ener;
          }  catch (java.lang.Throwable e) {
            exc=e;
          }
        }\> (  (      pwdAtPre_0 = passCodeAtPre_0(selfAtPre_0)
                    & !userHasLogedInAtPre_0(selfAtPre_0) = TRUE
                 -> self.userHasLogedIn = TRUE & result = TRUE & exc = null)
```

Figure 2.5: Proof Obligation

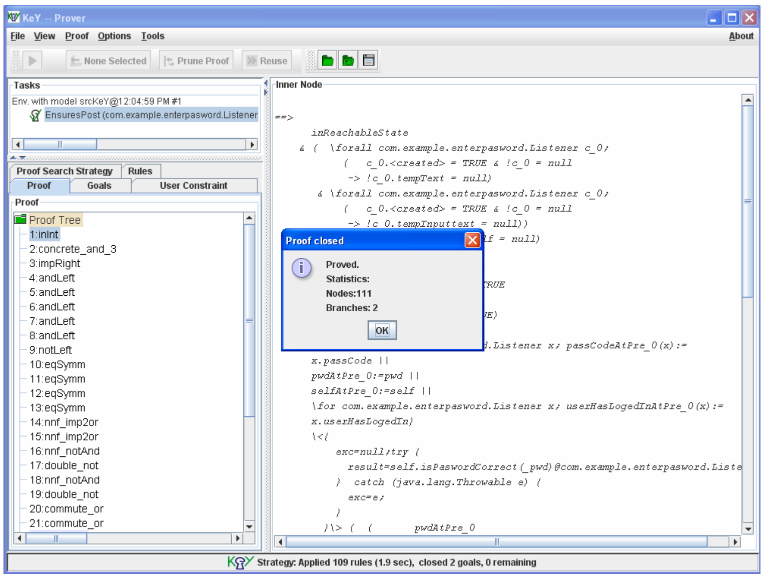
together with the (translated) class invariant, implies that the (translated) ensures part holds after the method.
(translated means: translate to finite sets of JAVA CARD DL formulas that can be submitted to the KeY prover.) Actually, being on the left side of the implication, provides additional assumptions that may, or may not, be used for establishing the right-hand side.

After loading the obligation we can prove it automatically by clicking on ▶ button either in the toolbar or by right clicking on the *Current Goal* pane and selecting *Apply rules automatically here*.
The obligation closes automatically as shown in Fig. 2.6.

`RespectsModifies` aspect of the method specification `isPasswordCorrect()`, is also proved automatically in the same way as stated above .

We have now verified that `isPasswordCorrect()` satisfies its contract and preserves the invariants of its class. Later on, when we are going to verify `onClick()` we can use the now proved contract of `isPasswordCorrect()` instead of inlining the method.

Proving `getPassword()` method is somewhat different from the `isPasswordCorrect()` method. As we can see in the implementation of the method, some android APIs such as: `getText()`, `intValue()`, `toString()` are invoked. The KeY system needs to know about these methods and we need to reason about the incorporated methods. To do this, we have two possibilities:



Env. with model srcKeY@12:04:59 PM #1
EnsuresPost (com.example.enterpasword.Listener

Proof
Proof Tree
1:inInt
2:concrete_and_3
3:impRight
4:andLeft
5:andLeft
6:andLeft
7:andLeft
8:andLeft
9:notLeft
10:eqSymm
11:eqSymm
12:eqSymm
13:eqSymm
14:nnf_imp2or
15:nnf_imp2or
16:nnf_notAnd
17:double_not
18:nnf_notAnd
19:double_not
20:commute_or
21:commute_or

```
==>
     inReachableState
   & (  \forall com.example.enterpasword.Listener c_0;
          (    c_0.<created> = TRUE & !c_0 = null
           -> !c_0.tempText = null)
      & \forall com.example.enterpasword.Listener c_0;
          (    c_0.<created> = TRUE & !c_0 = null
           -> !c_0.tempInputtext = null))
                                      lf = null)

                                      TRUE

                                      E)

                                      l.Listener x; passCodeAtPre_0(x):=
     x.passCode ||
     pwdAtPre_0:=pwd ||
     selfAtPre_0:=self ||
     \for com.example.enterpasword.Listener x; userHasLogedInAtPre_0(x):=
     x.userHasLogedIn}
     \<{
         exc=null;try {
           result=self.isPaswordCorrect(_pwd)@com.example.enterpasword.Liste
         }  catch (java.lang.Throwable e) {
           exc=e;
         }
       }\> (  (        pwdAtPre_0
```

Proved.
Statistics:
Nodes:111
Branches: 2

Strategy: Applied 109 rules (1.9 sec), closed 2 goals, 0 remaining

Figure 2.6: Proved Goal

1. Replacing the invocation of an API by the actual method body.

2. Replacing the invocation of an API by its specification.

Actually we often prefer to replace the invocation of an API by its specification (except when we are verifying the API itself). Because we usually do not have access to the implementation of the APIs and we just use them in our application.
Usually, APIs are specified and verified just when they are implemented. Afterwards, to specify any application which have used these APIs, we just need to use their specification. This means that we do not have to do the same work over and over again.
In other words, for every invocation of an API, if there is a specification, and the precondition of this method is satisfied in the current state, then one may actually replace the method call with its postcondition.

In our example, to prove correctness of the method, we should also prove the correctness of the invariants of called APIs before calling them, if they are assumed to be proved(if they are in the set of selected invariants INV).
They should also be held after termination of the method. Furthermore, the specification and implementation of our method should satisfy the contract of the used APIs.[10]

For example, as it is illustrated in the implementation of `getPassword()`, we have used `getText()` to get the entered password by the user. For the pur-

pose of verification, we need to specify this method in its *Stub* file which is the *TextView.java* class.
The contract is as follows:

—— Java + JML (2.11) ——

```

public class TextView extends android.view.View implements
                android.view.ViewTreeObserver.OnPreDrawListener
{
 ...
//@ public   model  CharSequence   inputText;
 ...
/*@ public normal_behaviour
  @ requires true;
  @ ensures \result==inputText ;
  @*/
 public /*@pure*/ java.lang.CharSequence getText();
```

—— Java + JML (2.11) ——

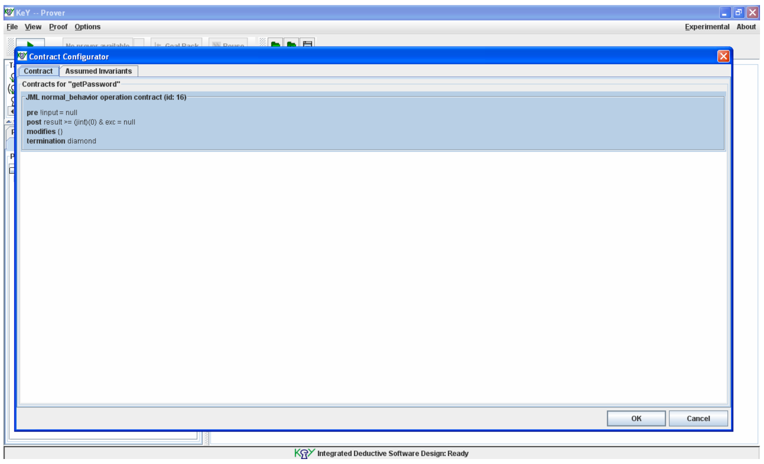
As it is described precisely in [6, "Components and Modular Proofs"], we need to ensure invariants of used components before the call.
In other words, the contract of the used APIs should remain valid under composition. The caller of the API should ensure the precondition and invariants of the API before the call, and invariants and its postcondition after the call.

Because only the public interface of the API is exposed and the internal implementation of the API is hidden, `model` fields are then a necessity to provide a specification of the API classes. This way the specification remains modular and does not need to be changed if the implementation changes. Actually, only the represents clauses might have to be adapted and it allows to add new subclasses(or classes implementing an interface) and to reuse and relate to the specification of the superclass/interface.

In the `getText()` method, just for specification purposes we declared a `model` field `inputText`. It is an abstract field. It represents the real field `mText`. Actually, the real field `mText` is set in `setText()` method. So to state that the `getText()` method should return exactly the value that has been set in `setText()` method, we use a model field.

To prevent adding more complexity to our proof obligation and because of the purpose of this example, we just add some specification as simple as possible to other used APIs. To prove correctness of the `getPassword()` method, after loading proof obligations, in Fig. 2.3 we select the `getPassword()` method in the left pane and `EnsuresPost` in right pane. After clicking "Start Proof" we see

”Contract Configuration” dialogue in Fig. 2.7.



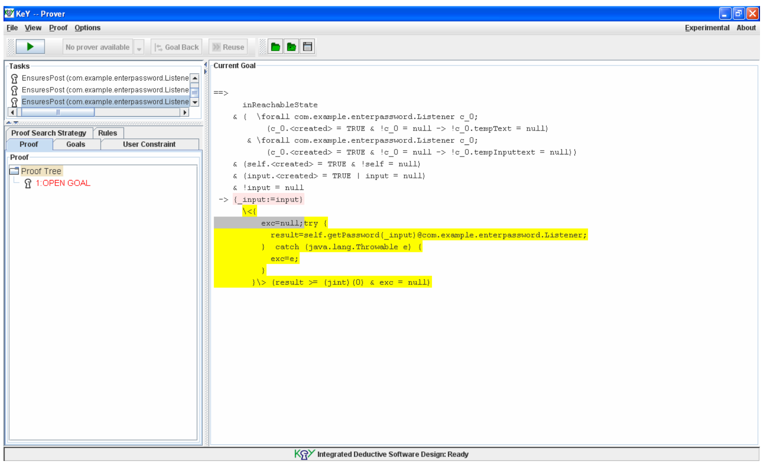
Contracts for "getPassword"

JML normal_behavior operation contract (id: 16)

```
pre !input = null
post result >= (jint)(0) & exc = null
modifies {}
termination diamond
```

Figure 2.7: Contract Configuration

In this dialogue we choose the contract and also in *Assumed Invariants* page we keep the default selection of assumed invariant without any change. The assumed invariants are the invariants of the `Listener` class and are some assumptions that appear in the sequent and help us to prove the correctness of the method.
After loading the proof obligation we see Fig. 2.8.


KeY -- Prover
File View Proof Options
Experimental About
No prover available
Goal Back
Reuse
Tasks
EnsuresPost (com.example.enterpassword.Listene
EnsuresPost (com.example.enterpassword.Listene
EnsuresPost (com.example.enterpassword.Listene
Proof Search Strategy
Rules
Proof
Goals
User Constraint
Proof
Proof Tree
1:OPEN GOAL
Current Goal
Integrated Deductive Software Design: Ready

```
==>
    inReachableState
  & (  \forall com.example.enterpassword.Listener c_0;
         (c_0.<created> = TRUE & !c_0 = null -> !c_0.tempText = null)
     & \forall com.example.enterpassword.Listener c_0;
         (c_0.<created> = TRUE & !c_0 = null -> !c_0.tempInputtext = null))
  & (self.<created> = TRUE & !self = null)
  & (input.<created> = TRUE | input = null)
  & !input = null
 -> {_input:=input}
      \<{
          exc=null;try {
            result=self.getPassword(_input)@com.example.enterpassword.Listener;
          }  catch (java.lang.Throwable e) {
            exc=e;
          }
        }\> (result >= (jint)(0) & exc = null)
```

Figure 2.8: Loaded Proof Obligation

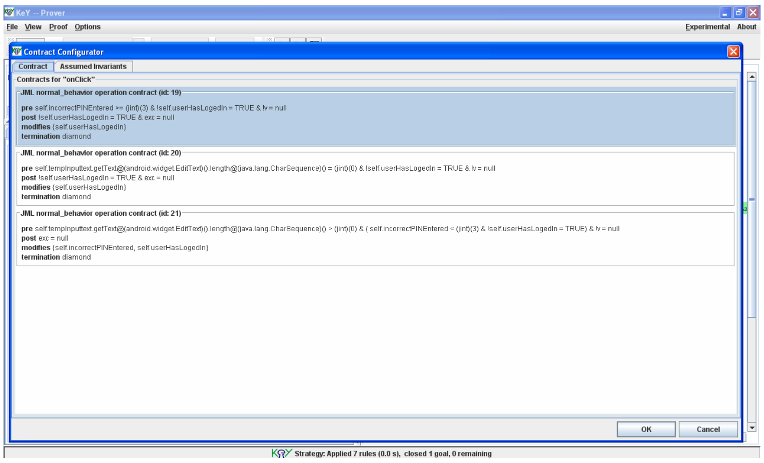
After some interaction with the KeY system and using the contracts of used APIs(deselection of some of the assumed invariants to decrease complexity), our goal closes automatically.
`RespectsModifies` aspect of the method specification is also proved automati-

cally in the same way as stated above.

We can now turn towards the verification of method `onClick()` and prove its correctness. For this we can use the now proved contract of `getPassword()` and `isPasswordCorrect()` instead of inlining the methods.

To prove correctness of the `onClick()` method, after opening Proof Obligation Browser dialogue, in Fig. 2.3 we choose the `onClick()` method in left pane and
`EnsuresPost` in the right pane. After clicking "Start Proof" we see the "ContractConfiguration" dialogue in Fig. 2.9.

In this dialogue we choose the first contract which corresponds to the third



Contracts for "onClick"

JML normal_behavior operation contract (id: 19)

```
pre self.incorrectPINEntered >= (jint)(3) & !self.userHasLogedIn = TRUE & !v = null
post !self.userHasLogedIn = TRUE & exc = null
modifies {self.userHasLogedIn}
termination diamond
```

JML normal_behavior operation contract (id: 20)

```
pre self.tempInputtext.getText@(android.widget.EditText)().length@(java.lang.CharSequence)() = (jint)(0) & !self.userHasLogedIn = TRUE & !v = null
post !self.userHasLogedIn = TRUE & exc = null
modifies {self.userHasLogedIn}
termination diamond
```

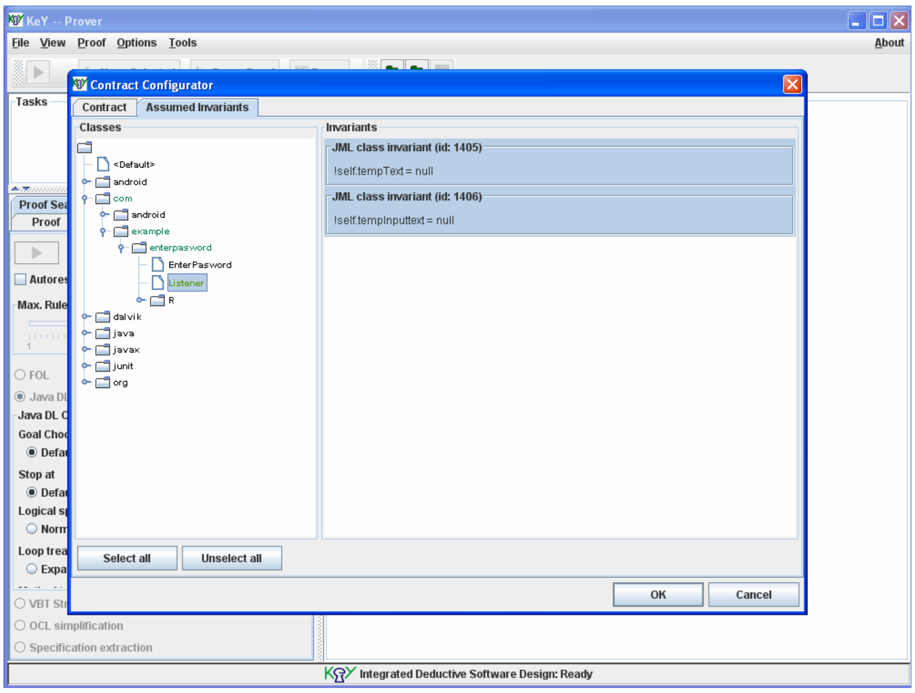
JML normal_behavior operation contract (id: 21)

```
pre self.tempInputtext.getText@(android.widget.EditText)().length@(java.lang.CharSequence)() > (jint)(0) & ( self.incorrectPINEntered < (jint)(3) & !self.userHasLogedIn = TRUE) & !v = null
post exc = null
modifies {self.incorrectPINEntered, self.userHasLogedIn}
termination diamond
```

Figure 2.9: Contract Configuration

part of the specification of the method and also on the *Assumed Invariants* page we keep the default selection of assumed invariants without any change as it is illustrated in Fig. 2.10. After loading the proof obligation, we see Fig. 2.11. This goal closes automatically by clicking on ▶ button in the toolbar as it is illustrated in Fig. 2.12.

Proving the second contract is somewhat different from the previous one because it requires user interaction to be closed. Actually after clicking on ▶ button and applying rules automatically we need to apply some rules by hand to close the goal. Because in this contract of the method we are calling other APIs we need to specify these APIs and use their contracts. One of these APIs is the `length()` method of type `CharSequence`. We have specified this API in `CharSequence` interface and also in all its implementing classes. These classes are: `AlteredCharSequence, SpannableStringBuilder, CharBuffer`. In the following code we have specified `CharSequence, AlteredCharSequence`. The other classes are specified in the same way as `AlteredCharSequence`.



Invariants

JML class invariant (id: 1405)

!self.tempText = null

JML class invariant (id: 1406)

!self.tempInputtext = null

Figure 2.10: Assumed Invariants

— Java + JML (2.12) —

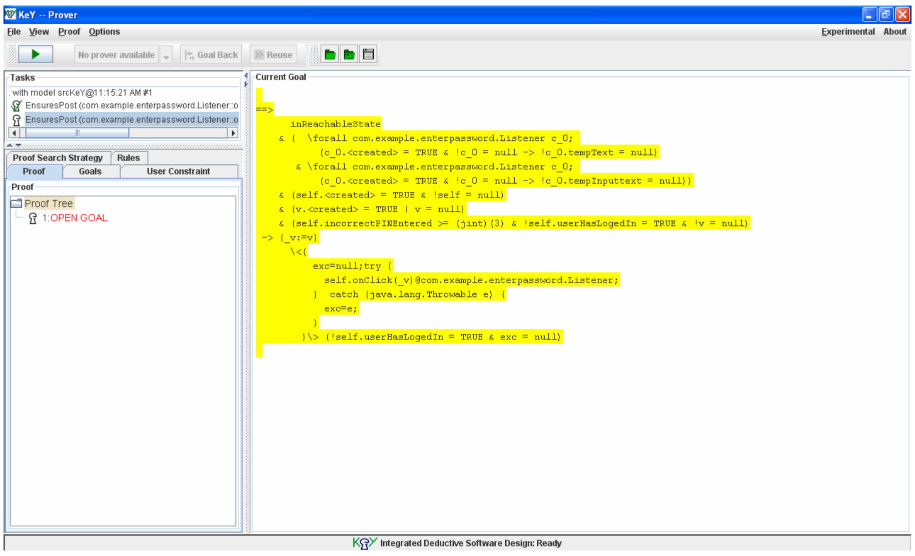
```
public interface CharSequence{
  /*@ public normal_behaviour
    @ requires true;
    @ ensures \result>=0;
    @*/
    public /*@pure*/int length();
```

— Java + JML (2.12) —

— Java + JML (2.13) —

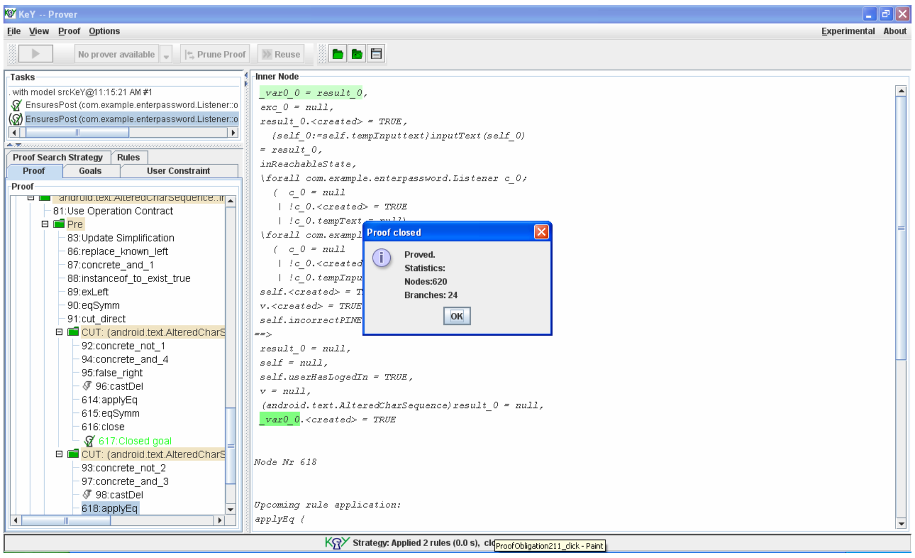
```
public class AlteredCharSequence extends java.lang.CharSequence,...
/*@ public model int len;      @*/
/*@ public invariant len >=0; @*/
...
/*@ public normal_behavior
  @ requires true;
  @ ensures \result == len ;
  @*/
public /*@ pure @*/ int length();
```

— Java + JML (2.13) —



```
==>
    inReachableState
  & (  \forall com.example.enterpassword.Listener c_0;
          (c_0.<created> = TRUE & !c_0 = null -> !c_0.tempText = null)
     & \forall com.example.enterpassword.Listener c_0;
          (c_0.<created> = TRUE & !c_0 = null -> !c_0.tempInputtext = null))
  & (self.<created> = TRUE & !self = null)
  & (v.<created> = TRUE | v = null)
  & (self.incorrectPINEntered >= (jint)(3) & !self.userHasLogedIn = TRUE & !v = null)
 -> {_v:=v}
     \<{
         exc=null;try {
           self.onClick(_v)@com.example.enterpassword.Listener;
         }  catch (java.lang.Throwable e) {
           exc=e;
         }
       }\> (!self.userHasLogedIn = TRUE & exc = null)
```

Figure 2.11: Loaded Proof Obligation



```
 _var0_0 = result_0,
 exc_0 = null,
 result_0.<created> = TRUE,
   {self_0:=self.tempInputtext}inputText(self_0)
 = result_0,
 inReachableState,
 \forall com.example.enterpassword.Listener c_0;
   (  c_0 = null
    | !c_0.<created> = TRUE
    | !c_0.tempText
 \forall com.exampl
   (  c_0 = null
    | !c_0.<created
    | !c_0.tempInpu
 self.<created> = T
 v.<created> = TRUE
 self.incorrectPINE
==>
 result_0 = null,
 self = null,
 self.userHasLogedIn = TRUE,
 v = null,
 (android.text.AlteredCharSequence)result_0 = null,
 _var0_0.<created> = TRUE


Node Nr 618


Upcoming rule application:
applyEq {
```

```
Proved.
Statistics:
Nodes:620
Branches: 24
```

Figure 2.12: Proved Goal

Now we can use their contract during verification. Further, we also can simplify the proof operation by selecting the `len >=0;` invariant in `AlteredCharSequence` class as "Assumed invariant" when applying "Use Operation Contract" rule on the `length()` method.

After clicking on ▶ button and using the "castDel" rule on some nodes of the proof tree, we reach a state that is not provable by just clicking on ▶ button or using "castDel" rule. To clear this state we should explain this proof situation in more detail.

Actually the proof situation when starting the proof for this contract (the second contract) is basically something like the following:

```
 inputText.length()==0 ->
<{
    if (inputText.length()==0) {
          var = false;
    } else {
          var = true;
    }
}> var = false
```

This formula should be provable as obviously the `else` branch is never executed. Still we cannot prove it, because when we try to evaluate the condition of the `if` statement we get first something like:

```
inputText.length()==0 ==>
<{
    b= inputText.length()==0;
    if (b) {
          var = false;
    } else {
          var = true;
    }
}> var = false
```

Followed by:

```
inputText.length()==0 ==>
<{
    int j = inputText.length();
    b = (j==0);
    if (b) {
          var = false;
    } else {
          var = true;
    }
}> var = false
```

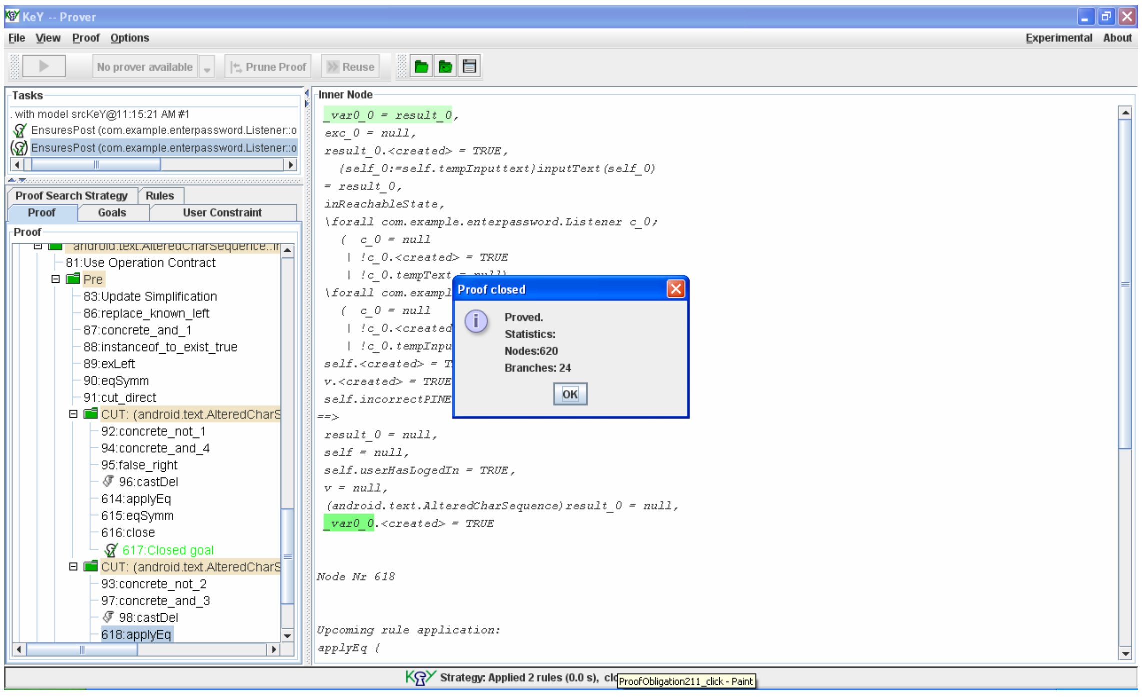
Now we symbolically execute *int j* = *inputText.length*(); using our contract we get then the following sequent:

```
inputText.length() == 0 ==>

{j:=c} (
(j>=0 && j == inputText.len) ->

<{
    b = (j==0);
    if (b) {
          var = false;
    } else {
          var = true;
    }

}> var = false
)
```

But the prove tree stops here and needs the user interaction to close the goal. To close the goal, we need to show that the result of `inputText.length()` in the specification is the same as it is in the implementation. So we use the "query" and "Use Operation Contract" rules on the precondition to simplify the sequent to something as follows:

```
d == inputText.len && d>=0),
d == 0
==>
(c>=0 && c == inputText.len) ->
{j:=c}<{
    b = (j==0);
    if (b) {
          var = false;
    } else {
          var = true;
    }
}> var = false
```

Now we can conclude that c==d and thus that c has the value 0 as well. This allows then to show that only the ”then” branch of the **if** statement can be entered and thus after some interaction with the KeY system and using the ”castDel” rule, our goal closes automatically as it is shown in Fig. 2.13 .



```
. with model srcKeY@11:15:21 AM #1
EnsuresPost (com.example.enterpassword.Listener::o
EnsuresPost (com.example.enterpassword.Listener::o
```

```
android.text.AlteredCharSequence::in
  81:Use Operation Contract
  Pre
    83:Update Simplification
    86:replace_known_left
    87:concrete_and_1
    88:instanceof_to_exist_true
    89:exLeft
    90:eqSymm
    91:cut_direct
    CUT: (android.text.AlteredCharS
      92:concrete_not_1
      94:concrete_and_4
      95:false_right
      96:castDel
      614:applyEq
      615:eqSymm
      616:close
      617:Closed goal
    CUT: (android.text.AlteredCharS
      93:concrete_not_2
      97:concrete_and_3
      98:castDel
      618:applyEq
```

```
 _var0_0 = result_0,
 exc_0 = null,
 result_0.<created> = TRUE,
   {self_0:=self.tempInputtext}inputText(self_0)
 = result_0,
 inReachableState,
 \forall com.example.enterpassword.Listener c_0;
   (  c_0 = null
    | !c_0.<created> = TRUE
    | !c_0.tempText = null)
 \forall com.exampl
   (  c_0 = null
    | !c_0.<created
    | !c_0.tempInpu
 self.<created> = T
 v.<created> = TRUE
 self.incorrectPINE
==>
 result_0 = null,
 self = null,
 self.userHasLogedIn = TRUE,
 v = null,
 (android.text.AlteredCharSequence)result_0 = null,
 _var0_0.<created> = TRUE


Node Nr 618


Upcoming rule application:
applyEq (
```

```
Proved.
Statistics:
Nodes:620
Branches: 24
```

```
Strategy: Applied 2 rules (0.0 s), cl
ProofObligation211_click - Paint
```

Figure 2.13: Proved Goal

The third contract also can be closed automatically by using the contract of the used APIs and using the "castDel" rule as described above.

Chapter 3

Specification of Android Applications

In this chapter we are going to focus on android core applications and some critical aspects of their functionalities.
We introduce the *Phone* application as a fundamental application and also its critical APIs, along with their specifications in JML.
Afterwards, we focus on the *Screen manager* and *Contact* applications, describing their important APIs and their specification in JML.

3.1 Specifying the Phone Application

The Phone application is one of the most often used applications in android and provides functionalities such as: Dialer, Bluetooth, Ring Manager, and so on.
The Phone application is a part of all produced mobile devices running android and provides (access to) essential functionalities.
In this section we are going to focus on Emergency Dialer as a special case of Dialer functionality.

As the Phone application is used for critical use cases like: emergency calls and financial transactions, possible bugs in these parts can lead to financial losses, security or safety critical issues.
For example, every mobile phone provides an Emergency Dialer functionality. To verify that a dialed emergency number is actually handed over to the carrier is an important safety property, while verifying that only emergency numbers are

dispatched is a central security feature. The following is an image of the Phone application.

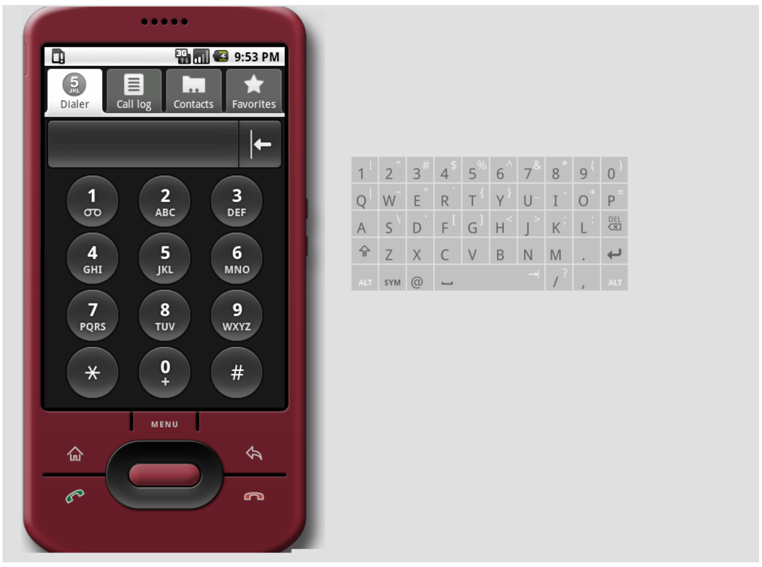


Figure 3.1: Phone Application(Dialer)

3.1.1 First Walk through Specifying Emergency Dialer

The `EmergencyDialer` is a special dialer that is used only to dial emergency calls.
It must be possible to dial emergency numbers even when the phone is locked, no SIM card is available, or the mobile phone is on Airplane mode, or an emergency number is entered instead of the PIN. Therefore the android OS provides the activity `EmergencyDialer`, which is the first activity for dialing an emergency number.
On the other hand, it should only dispatch emergency numbers provided by the *RIL* (Radio Interface Layer) and SIM card.
A RIL is a layer in the android operating system which provides an interface to the hardware's radio and modem on a mobile phone. The android provides a Radio Interface Layer (RIL) between android telephony services (android.telephony) and the radio hardware[11].

The mobile phone and SIM card have a preprogrammed list of emergency numbers.
"Most GSM(Global System for Mobile Communications) mobile phones have 112, 999 and 911 as pre-programmed emergency numbers that are always available. The SIM card issued by the operator can contain additional country-specific emergency numbers that can be used even when roaming abroad. The GSM

network can also update the list of well-known emergency numbers when the phone registers to it".[12]

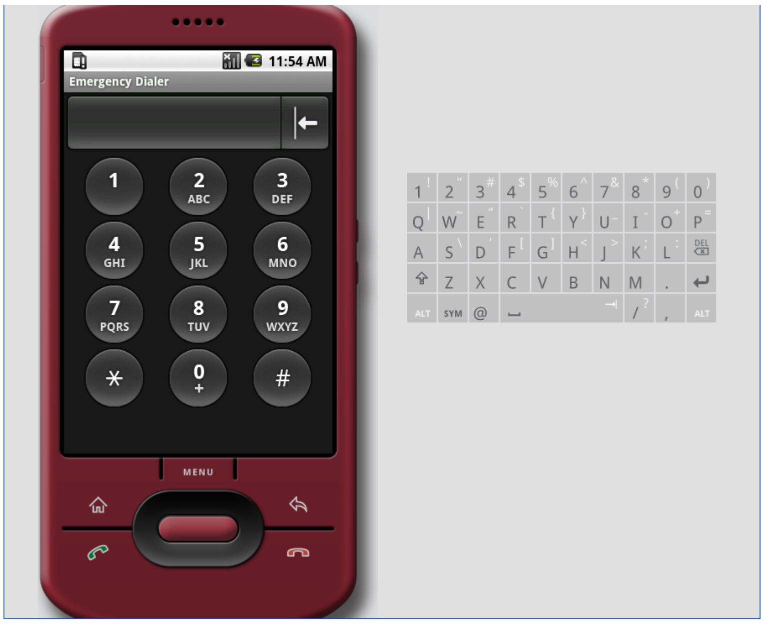


Figure 3.2: Emergency Dialer

EmergencyDialer Activity

The `EmergencyDialer.java` class is the first activity for dialing an emergency number. It is a special case of the *TwelveKeyDialer* that:

- Allows only emergency calls to be dialed.
- Disallows *voicemail* functionality. Voice Mail is a computer based means of communication. All messages are sent in the original voice of the sender using a standard telephone.
  In a single call to Voice Mail, the user can send messages to others, pick up messages sent to her by subscribers or outside callers, reply to messages, and send a copy of a message to someone else. Moreover there is a personal password that protects the Voice Mail.
  We also may use Voice Mail to send messages to other individual subscribers, or groups of subscribers, without calling them directly.
- Handles *keyguard* access correctly among all of its methods. The Keyguard is a mechanism to control access to the cell phone by providing a password or pin code or so on, to login. In Emergency Dialer calls, the Keyguard is ignored to call emergency numbers even if the user is not authorized. After ending the emergency call, the Keyguard is activated again.

The following is the sequence diagram of the launched components involved in dialing an emergency number.

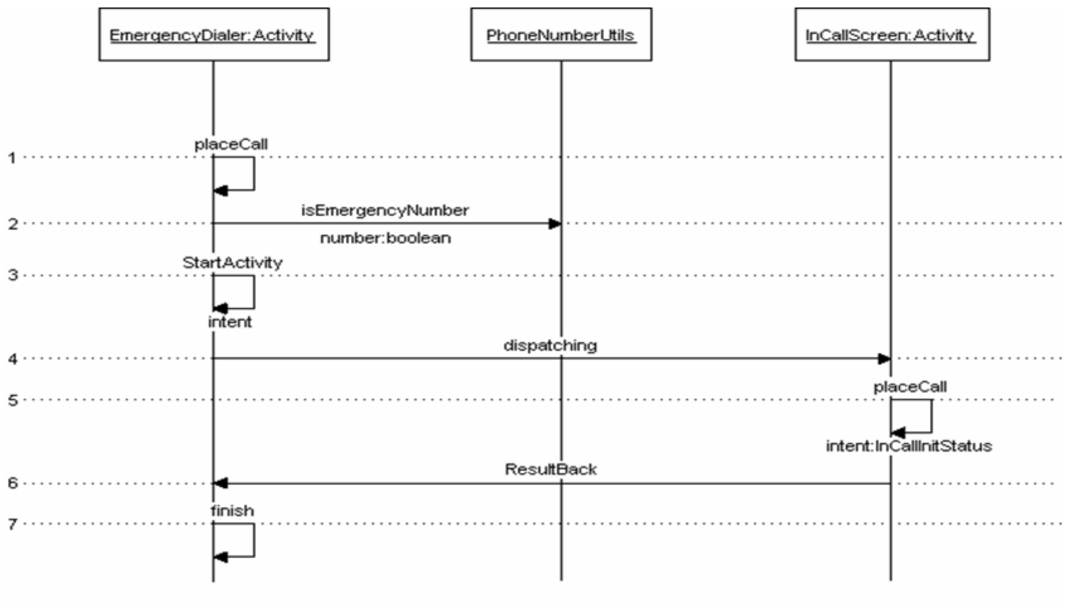


Figure 3.3: Emergency Sequence Diagram

In explanation of the above sequence diagram, first, `EmergencyDialer` activity is called by the OS to handle the emergency calls. Further, In this class, `placeCall()` method is called. It is the first method which is responsible to handle the emergency calls. To ensure that the input number is actually an emergency number, `placeCall()` method calls `isEmergencyNumber()` method in `PhoneNumberUtils` class. This method checks a given number against the list of emergency numbers and returns `false` if it is not an emergency number. In this case the `EmergencyDialer` activity erases the number and shows an alert dialogue. Otherwise, it creates an Intent with the given number. Afterwards, the activity dispatches the Intent to the responsible activity: `InCallScreen`. It is done by calling `startActivity` method with the specified Intent.
In `InCallScreen` activity, the emergency number actually is handed over to the carrier by the `placeCall()` method. After dispatching the number, the result of the action (failure or success) can be sent back to the caller activity: `EmergencyDialer`.
Finally the `finish()` method in `EmergencyDialer` is called to end the activity.

The following is Java and JML code for the `placeCall()` method in `EmergencyDialer` class as the first handler for an emergency call.

— Java + JML (3.1) —

```
/*@ public model  String  number;
  @ public ghost boolean isCallDispatched;
  @ invariant number==mDigits.getText().toString();
  @*/
/*@ public normal_behaviour
  @ requires !DBG &&
  @          PhoneNumberUtils.isEmergencyNumber(number)  ;
  @ ensures     isCallDispatched;
  @ assignable  isCallDispatched,mFinished,
  @             intent.callDispatched,\object_creation(Intent);
  @ also
  @ requires !DBG &&
  @ !(PhoneNumberUtils.isEmergencyNumber(number));
  @ ensures   !isCallDispatched;
  @ assignable isCallDispatched;
  @*/
 void placeCall() {
 //@ set isCallDispatched = false;
  final String number =
                 mDigits.getText().toString();
  if (PhoneNumberUtils.isEmergencyNumber(number))
      {  ...
      // place the call if it is a valid number
      Intent intent = new
                   Intent(Intent.ACTION_CALL_EMERGENCY);
      intent.setData(Uri.fromParts("tel", number,null));
     intent.setFlags(Intent.FLAG_ACTIVITY_NEW_TASK);
   //@ set intent.callDispatched = false;
     startActivity(intent);
   //@ set isCallDispatched =
                         intent.callDispatched;
     finish();
      }
   else {
        // erase the number and shows an alert dialogue.
                ...
        }
}
```

— Java + JML (3.1) —

First of all the `normal_behavior` keyword indicates that this specification describes the case in which `placeCall()` returns normally without throwing an exception.
The `requires` clause in line 6 states that, the entered number should be an emergency number. The number is checked by a function in `PhoneNumberUtils` class to check if it is an emergency number. The other precondition of this method is that *DBG* should be `false` which means we are not in debug mode.

In the `ensures` clause in line 8, we state that by fulfilling the requirements, just the emergency numbers should be handed over to the carrier. We show this by introducing a ghost variable which is set to `false` before calling the corresponded activity which is `InCallScreen`. This activity is responsible to actually dispatch the emergency number to the network. The declared ghost field `isCallDispatched` is used to check if dispatching the number to the network has been success or not. It is set to `false` at the beginning in line 18. There is another ghost field `callDispatched` that is introduced as an auxiliary field in the Intent class. See listing (3.2).

— Java (3.2) —

```java
public class Intent implements Parcelable {
...
//@ public ghost boolean callDispatched;
....
```

— Java (3.2) —

Because we use an Intent to dispatch the emergency number to the corresponding activity `InCallScreen`, we add a ghost field to the Intent object to lead to access to the result of dispatching. This field is set to `false` before dispatching, it will be set again in (`InCallScreen)` according to situation of dispatching. After activity `InCallScreen` is finished and the success or failure case is recorded by setting the ghost field `callDispatched`, the ghost field `isCallDispatched` is now updated accordingly (line 30).
Finally the method finish() is invoked to end the activity (line 32).

As explained in the preceding paragraphs the fields `isCallDispatched, intent.callDispatched`, are potentially changed and so they have to be included into the assignable set, further the field `mFinished` also is changed in the base class: Activity after calling `finish()` method in line 32. Their meaning is not of importance for our purposes and therefore not further explained.

— Java (3.3) —

```
public class Activity extends ContextThemeWrapper
        implements LayoutInflater.Factory,
        Window.Callback, KeyEvent.Callback,
        OnCreateContextMenuListener, ComponentCallbacks {
...
public void finish()
 {
     ...
        mFinished = true;
     ...
}
...
}
```

— Java (3.3) —

As it is described in section 2.1.7, `\object_creation(Intent)` is a KeY keyword and we should introduce any new created object in this way.

The second part starting with `also` is where the other case is described. Just like the previous case, this situation also has a `requires` and an `ensures` clause describing the situation that the entered number is not an emergency number. In this case the number should not be handed over to the carrier and `isCallDispatched` should be `false`. `assignable` clause in line 15 states that just `isCallDispatched` can be changed after termination of the method.

As we can see in line 29, `startActivity(intent)` is called. Actually, after calling `startActivity(intent)`, as it is shown in the sequence diagram 3.3, the corresponding activity in respect to the input parameter intent is launched. The launched activity is `InCallScreen`. It is where the emergency number actually is handed over to the carrier.
In this class, the `placeCall(Intent intent)` method is responsible for dispatching the dialed number. So, the correctness of the functionality of this method is desired. In the next step we explain the `placeCall` method in `InCallScreen` and its specification in JML.

InCallScreen Activity

As already mentioned in the previous section, the `InCallScreen` activity is the place where the number is dispatched to the carrier by calling its `placeCall(Intent intent)` method. Actually, the method makes a call to whomever the `intent` parameter tells us to and the `intent` parameter is the Intent we were launched with.
The `intent` here is passed through `startActivity` in `placeCall(Intent intent)` method of EmergencyDialer class see section 3.1.1.

── Java + JML (3.4) ──

```
 private InCallInitStatus placeCall(Intent intent) {
...
}
```

── Java + JML (3.4) ──

The method returns `InCallInitStatus.SUCCESS` if it successfully initiated an outgoing call. If there was some kind of failure, return one of the other InCallInitStatus codes indicating what went wrong.
We want to be sure that it actually dispatches the emergency numbers to the carrier successfully.
The specification of the `placeCall()` method describes more details, as follows:

── Java + JML (3.5) ──

```
1 /*@ public model InCallInitStatus okToCallState;
2   @ public ghost boolean isRadioOn;
3   @ represents okToCallState<-checkIfOkToInitiateOutgoingCall();@*/
```

── Java + JML (3.5) ──

The above JML specification is added directly after the header of `InCallScreen` class declaration. As we can see, first a model field is declared in line 1. This field is used to specify the method in listing(3.6). We need to specify the relation between the concrete field and the model field. The model field okToCallState exactly corresponds to the result of `checkIfOkToInitiateOutgoingCall()` method. It means whenever we refer to `okToCallState` in a specification, we mean the result of `checkIfOkToInitiateOutgoingCall()` in the implementation.
We describe the `placeCall()` method precisely in listing(3.10).

— JML (3.6) —

```
 4 /*@ public normal_behaviour
 5   @ requires
 6   @      !DBG  &&
 7   @      intent.getAction() == Intent.ACTION_CALL_EMERGENCY   &&
 8   @      intent.getFlags()  == Intent.FLAG_ACTIVITY_NEW_TASK  &&
 9   @      intent.getData().getScheme().equals("tel")           &&
10   @      !intent.callDispatched                               &&
11   @      PhoneNumberUtils.isEmergencyNumber(
12   @               intent.getData().getSchemeSpecificPart() )  &&
13   @      (okToCallStatus == InCallInitStatus.SUCCESS );
14   @ ensures intent.callDispatched && \result==
15                                         InCallInitStatus.SUCCESS;
16   @ assignable intent.callDispatched,(mDialer.*);
```

— JML (3.6) —

The specification of the `placeCall()` method starting in line 4 (listing 3.6) describes the method contract in a normal termination. Actually, we specify two normal behavior and one exceptional behavior cases for the method. The first normal behavior covers the case that the input intent corresponds to an emergency number and the radio is on. The second case is when the input intent corresponds to an emergency number but the radio is off. In the first case the `requires` clause states that the fields of the input parameter `intent` should correspond to an emergency call request. Moreover the requested number should be an emergency number and it is checked in line 11 by calling `isEmergencyNumber` method of `PhoneNumberUtils` class (listing 3.11).
We need to be sure that `intent.callDispatched` is `false` in line 10, before dispatching the number. Because it is already assigned to `false` in `EmergencyDialer` class listing(3.1) and it will be set to `true` after dispatching .
The `ensures` clause in line 14, states that, by fulfilling these requirements, after termination of the method, `intent.callDispatched` should be `true`, because it is set to `true` in line 78 listing(3.9) after dispatching the emergency number successfully.
Also, the return value of the method should be `InCallInitStatus.SUCCESS` that means we successfully initiated an outgoing call.

JML (3.7)

```
@ also
@
@ requires
@        !DBG                                          &&
@        okToCallStatus == InCallInitStatus.POWER_OFF    &&
@        PhoneNumberUtils.isEmergencyNumber(
                 intent.getData().getSchemeSpecificPart() )  &&
@        !isRadioOn;
@
@ ensures
@        isRadioOn ==>(\result==InCallInitStatus.SUCCESS  &&
                           mFinished && intent.callDispatched );
@ assignable isRadioOn,mFinished,intent.callDispatched;
```

JML (3.7)

The listing(3.7) covers the second normal behavior case. It presents the situation that radio is explicitly powered off and the number is an emergency number. As it is mentioned earlier an emergency number should be dialed even if the radio is off. In this case, the system should try to power on the radio and try to dial the number. To do this, the system calls another activity `EmergencyCallHandler` to do whatever else is needed. The called activity is responsible to turn on the radio and callback `placeCall()` in `InCallScreen` again.
So, for now, the method finishes the `InCallScreen` (since it is expecting a callback when the responsible activity dictates it) and just returns the success state. To handle this situation we introduce a `ghost` field: `isRadioOn` in line 2 listing(3.5). It is assigned to `false` in line 75 listing(3.9) once the radio is off and the number is an emergency number. After calling the new activity, we check the status of the radio in line 78 listing(3.9), and `isRadioOn` is assigned again.
The specification states that, if the radio is off and the number is an emergency number, after termination of the method, it should be on and the number should be dialed.
The return value should be `InCallInitStatus.SUCCESS` and `InCallScreen` activity should be closed `(mFinished==true)`.
In the assignable clause we state that just `isRadioOn,mFinished,` `intent.callDispatched` are allowed to be changed after termination of the method.

— JML (3.8) —

```
30 @ also
31 @
32 @ public exceptional_behaviour
33 @ requires
34 @      !DBG                                             &&
35 @      intent.getData().getScheme().equals("voicemail")  &&
36 @      (intent.getData().getSchemeSpecificPart()  == null
37 @       || TextUtils.isEmpty(intent.getData().
38 @                                 getSchemeSpecificPart() ));
39 @
40 @ signals (Exception e) e instanceof
41 @            PhoneUtils.VoiceMailNumberMissingException &&
42 @                     \result==okToCallStatus ;
43 @ assignable \nothing;  */
```

— JML (3.8) —

The last specification part in line 32 listing(3.8), specifies exceptional behavior case of the method. It is when the intent contains a voicemail URI, but there's no voicemail number configured on the device. It can happen when the call status is not in an acceptable state, so it may effect the way the voicemail number to be retrieved in line 56 listing(3.9).
In this case it has specified the `exception` that can be thrown within a `signal` clause. The exception is of type `(PhoneUtils.VoiceMailNumberMissingException)`.

The following code shows the implementation of the `placeCall(Intent intent)` method.

— Java + JML (3.9) —

```
44  private InCallInitStatus placeCall(Intent intent) {
45 // Check the current ServiceState to make sure it is
46 // OK to even try making a call.
47  InCallInitStatus okToCallStatus =
48                        checkIfOkToInitiateOutgoingCall();
49    try {
50          number = getInitialNumber(intent);
51        } catch (PhoneUtils.VoiceMailNumberMissingException ex) {
52
53            if (okToCallStatus != InCallInitStatus.SUCCESS)
54                return okToCallStatus;
55
56            return InCallInitStatus.VOICEMAIL_NUMBER_MISSING;
57        }
```

```
 ...
boolean isEmergencyNumber =
                       PhoneNumberUtils.isEmergencyNumber(number);
boolean isEmergencyIntent =
                         Intent.ACTION_CALL_EMERGENCY.equals(
                                                 intent.getAction());
if (isEmergencyNumber && !isEmergencyIntent) {
         ...
          return InCallInitStatus.CALL_FAILED;
   } else if (!isEmergencyNumber && isEmergencyIntent) {
          ...
         return InCallInitStatus.CALL_FAILED;
   }
 ...
if (okToCallStatus != InCallInitStatus.SUCCESS) {
  if (isEmergencyNumber && (okToCallStatus ==
                               InCallInitStatus.POWER_OFF)){
       //@ set isRadioOn = false;
       startActivity(intent.setClassName(this,
                    EmergencyCallHandler.class.getName()));
       /*@ set isRadioOn==( checkIfOkToInitiateOutgoingCall()
                      ==ServiceState.STATE_POWER_OFF)*@/
       finish();
       return InCallInitStatus.SUCCESS;
     } else {
              return okToCallStatus;}
     ...
//@ set intent.callDispatched=false;
int callStatus = PhoneUtils.placeCall(mPhone, number,
                                                 intent.getData());
switch (callStatus) {
         case PhoneUtils.CALL_STATUS_FAILED:
             ...
             return InCallInitStatus.CALL_FAILED;
         case PhoneUtils.CALL_STATUS_DIALED:
             //@ set intent.callDispatched=true;
             return InCallInitStatus.SUCCESS;
           ... }
}
```

Java + JML (3.9)

As we described before in section 2.3.3, for the purpose of verification we need to specify all called methods in `placeCall()`. So we first specify the `checkIfOkToInitiateOutgoingCall` method, afterwards, the `isEmergencyNumber` API in JML.

The `checkIfOkToInitiateOutgoingCall` method checks the current ServiceState to make sure it is OK to try making an outgoing call to the specified number. It returns `InCallInitStatus.SUCCESS` if it is OK to try calling the specified number. If not, like if the radio is powered off or we have no signal, it returns one of the other `InCallInitStatus` codes indicating what the problem is.
Specification and implementation is as follows:

— Java + JML (3.10) —

```
/*@ public model int state;
  @ represents state<-mPhone.getServiceState().getState();
  @*/

/*@ public normal_behaviour
  @ requires state==ServiceState.STATE_IN_SERVICE;
  @ ensures \result==InCallInitStatus.SUCCESS;
  @ assignable \nothing;
  @ also
  @
  @ requires state!=ServiceState.STATE_IN_SERVICE;
  @ ensures \result!=InCallInitStatus.SUCCESS;
  @ assignable \nothing;
  @ also
  @
  @ public exceptional_behaviour
  @ requires state!=ServiceState.STATE_IN_SERVICE &&
  @          state!=ServiceState.STATE_POWER_OFF  &&
  @          state!=ServiceState.STATE_OUT_OF_SERVICE &&
  @          state!=InCallInitStatus.EMERGENCY_ONLY;
  @ signals (Exception e) e instanceof  IllegalStateException;
  @ assignable \nothing;
*@/
private /*@ pure @*/InCallInitStatus checkIfOkToInitiateOutgoingCall()
  {
        int state = mPhone.getServiceState().getState();
        switch (state) {
```

```
28              case ServiceState.STATE_IN_SERVICE:
29               // Normal operation.  It is OK to make outgoing calls.
30                  return InCallInitStatus.SUCCESS;
31
32              case ServiceState.STATE_POWER_OFF:
33                  // Radio is explicitly powered off.
34                  return InCallInitStatus.POWER_OFF;
35
36              case ServiceState.STATE_OUT_OF_SERVICE:
37              case ServiceState.STATE_EMERGENCY_ONLY:
38               // The phone is registered, but locked. Only emergency
39               // numbers are allowed.
40                  return InCallInitStatus.EMERGENCY_ONLY;
41              default:
42                  throw new IllegalStateException(
43                                 "Unexpected␣ServiceState:␣" + state);
44          }
45      }
```

Java + JML (3.10)

As the implementation shows, the method checks the current `ServiceState` and if it is in `STATE_IN_SERVICE` mode it is OK to try making an outgoing call to the specified number. So just in this mode the result value of the method is SUCCESS. We have described this situation in the specification in lines 5 to 13. If the current `ServiceState` is none of the permitted states, an exception will be thrown. This is described in lines 16 to 22.

As we can see in the listings (3.1), (3.5) and (3.6), the `isEmergencyNumber` API, is called to determine if the number entered by the user is an emergency one.
The following is the specification and the implementation of the method.
This API checks a given number against the list of emergency numbers provided by the RIL and SIM card. If the number is in the list of emergency numbers listed in the RIL / SIM, then it returns `true`, otherwise `false`. łabel-list:PhoneNumberUtils

Java + JML (3.11)

```
/*@ public model static non_null java.lang.String[]
  @                                    emergencynumbers;

  @ invariant(\exists int i,j ; j>=0 && i>=0 &&
  @       i<emergencynumbers.length && j<emergencynumbers.length ;
  @       emergencynumbers[i].equals("911")&&
7    @      emergencynumbers[j].equals("112") );
8    @ invariant(\forall int i;i>=0 && i<getEmergencyNumbers().length;
9    @            (\exists int j;j>=0 && j<emergencynumbers.length ;
10   @             emergencynumbers[j]==getEmergencyNumbers()[i]));
11   @*/
12
13 /*@ public model String[] getEmergencyNumbers() {
14   @ String numbers = SystemProperties.get("ro.ril.ecclist");
15   @ String[] _emergency =
16   @             new String[numbers.split(",").length();
17   @ for (int i=0;i< numbers.split(",").length;i++)
18   @       _emergency[i]=numbers.split(",")[i];
19   @
20   @ return _emergency;}
```

Java + JML (3.11)

As we can see in line 1 we have declared a `model` field: `emergencynumbers`. This field is an array of `String`. It saves a list of all emergency numbers and contains all numbers that are in RIL/SIM, plus "912", "112" which are two international emergency numbers.
In line 4 to 11 we have declared two invariants. These invariants are used to describe the included members of the `emergencynumbers` array. Later on, we use this array to specify the `isEmergencyNumber` API.
We also declared a `model` method: `getEmergencyNumbers`. This method (from line 13 to 20) is used just for the purpose of specification and retrieves the list of emergency numbers from RIL/SIM.

Java + JML (3.12)

```
21 /*@ public normal_behaviour
22   @ requires number !=null;
23   @ ensures \result==((\exists int i;i>=0 && i< emergencynumbers.length ;
24   @                        number.equals(emergencynumbers[i])));
25   @ assignable \nothing;
26   @ also
27   @ requires number==null;
28   @ ensures \result==false;
29   @ assignable \nothing;*/
30  public static boolean isEmergencyNumber(
31                String/*@ nullable @*/ number){...}
32
```

Java + JML (3.12)

The specification of the method starts in line 21. In line 22 we specify the first situation, when the input number is not null. In this case, we expect a result `true` if the input number is in the `emergencynumbers` array, otherwise we expect a result `false`.
In line 27 we describe the second situation when the input value is null. In this case the return value always should be `false`.
In line 31, we have added `/*@ nullable @*/` to input value. This is because, in JML all fields are assumed `non_null` by default, so if we have such a field which may have a `null` data, we should state this by adding `/*@ nullable @*/` before the name of the field.

3.2 Specifying the Screen manager Application

Most cell phones in the market have a way to control the access to the cell phone. For this purpose, they have the capability to set a kind of password on the device so that the user needs to enter the password to activate the cell phone.
Android cell phones also have this capability to lock the screen and request a password to unlock.

When the cell phone turns on or when it is on standby mode (when we do not use it for a while), to activate it again, the device requests for a password. If the entered password is a correct password then the screen would be unlocked, otherwise it would remain locked.

In android these functions are accomplished by the Screen manager application which is run automatically by the OS when the device is turned on.
Fig. 3.4 shows the lock screen.

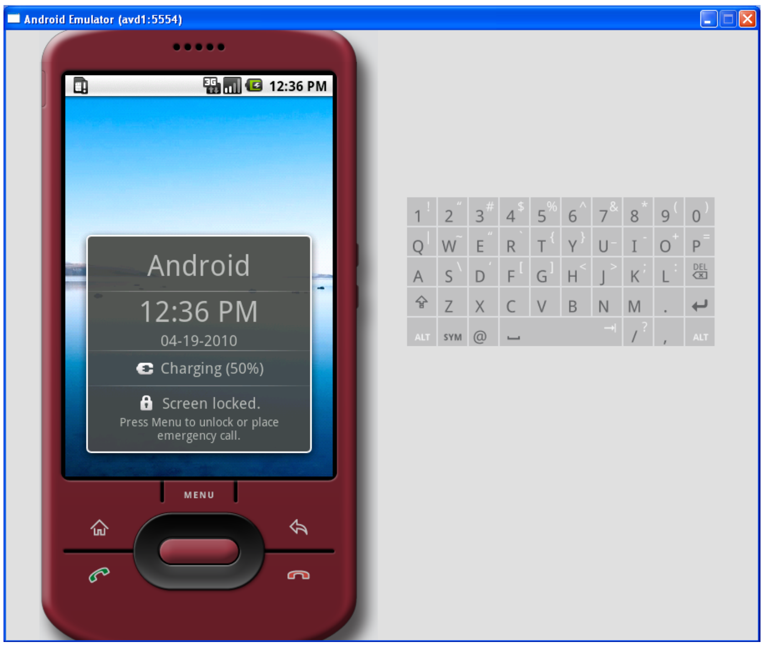

Android Emulator (avd1:5554)
12:36 PM
Android
12:36 PM
04-19-2010
Charging (50%)
Screen locked.
Press Menu to unlock or place emergency call.
MENU

Figure 3.4: Lock Screen

In android, unlocking the screen can be done in three different ways based on the user's choice:

1. drawing a pattern
2. entering a sim pin
3. entering an account's login and password

Fig. 3.5 shows the unlock screen expecting a pattern to be drawn by the user.

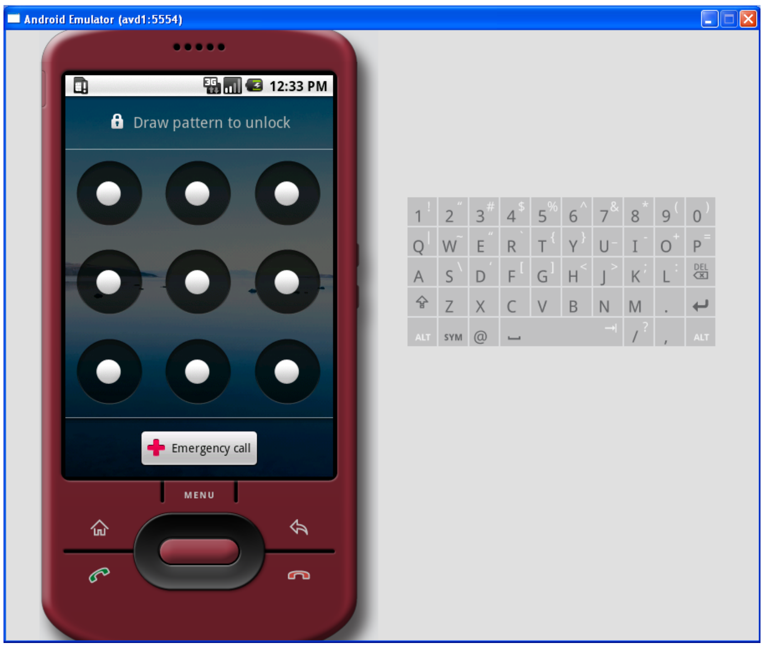


Figure 3.5: Unlock Screen

Since any defects in the unlock screen functionality can lead to a blocked cell phone so ensuring correctness of its functionality is very critical for android.
In the following section, first, we try to describe the sequence of used APIs to lock and unlock the screen. Afterwards we specify the important APIs formally using JML.

3.2.1 Lock And Unlock Screen Functionalities

Whenever the cell phone is turned on by the user or it is on standby mode(when a specified timeout after a user activity is finished) and the user is going to perform some other activities, the Lock screen pops up. As shown in Fig. 3.4, it shows general information about the device depending on its state. The user needs to press *Menu*. Afterwards, the Unlock screen(base on the user choice) is shown to the user expecting a correct pattern as shown in Fig. 3.5.
In this scenario, the system needs to start the `lock screen` activity automatically, when it is needed. So, android uses a *service*: `KEYGUARD_SERVICE` to accomplish this.
As it is described in Section 1.4, a service is an application component that runs in the background, not interacting with the user, for an indefinite period of time. This service runs automatically by the android and starts the `unlock screen` activity when it is needed.
Actually, there is also another scenario to call `unlock screen` it is when the

screen is unlocked temporarily for some reason and should be locked again after a certain action. For example, to receive an incoming call while the device is locked, the system needs to disable the keyguard the entire time the activity `InCallScreen` is active. later on, the application should re-enable the keyguard again.
In this situation, after disabling the keyguard (temporary) the input would be in *restricted* mode. In this mode if the application needs to navigate to something that is not safe to view without getting past the keyguard (e.g., pressing Home key), the system will bring up the unlock screen of the keyguard.

The input is always restricted when the keyguard is showing, or when the keyguard was suppressed by an application that disabled the keyguard or the device has not been provisioned yet(the device has not gone through the setup wizard and it is not ready to use).
When the system is in such a mode, certain keys, such as the Home key and the right soft keys, do not work.

The applications always check if the key input is in restricted mode. In this situation, it should then bring up the unlock screen of the keyguard again.
This is done by a API `exitKeyguardSecurely` in `KeyguardManager` class.
If the user gets past the keyguard, the result of this API is successful, and any other application requested by the user or other applications can be accomplished.
Another example is `EmergencyDialer` activity. As we described earlier in section 3.1.1, an emergency number should be dialed even if the cell phone is locked and the keygaurd is not passed by the user. So, in this activity also the key input is in restricted mode and some keys do not work.

In Fig. 3.6 the sequence diagram shows the involved APIs in order to unlock screen in the second scenario. As we can see in the sequence diagram, `exitKeyguardSecurely` is the first API called by the application that needs to get past the keyguard to navigate to something that is not safe to view without getting past the keyguard. After passing through some other classes, the final class which should verify the password depending on the user selection, is one of the following classes: `SimUnlockScreen, AccountUnlockScreen, UnlockScreen`.
In the next section we specify the `UnlockScreen` class.

Specification Of UnlockScreen

The `UnlockScreen` class is responsible to verify a pattern mode password and returns the result to the caller. In this class, the API `onPatternDetected` detects the pattern. It checks the drawn pattern with the correct pattern pre-declared by the user. If it was a correct pattern, the method would return the `true` result to the caller. Otherwise, it shows a message regarding the wrong pattern and also it increases the failed attempts since the last timeout by one.

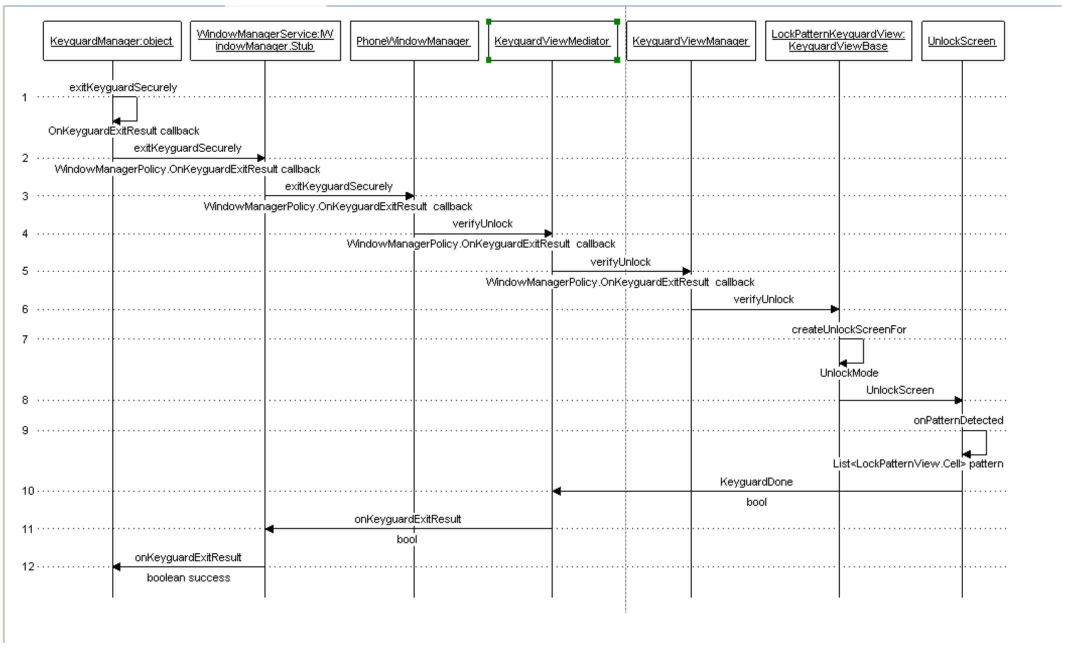


Figure 3.6: Unlock Screen sequence diagram

This field(failed attempts since the last timeout) is the maximum number of incorrect attempts before the user is prevented from trying again for. It is by default 5. It also increases the total failed attempts. If the user has more than 5 failed attempts, the footer shows a message regarding a forgotten lock pattern. The following codes show the specification and implementation of `Unlock screen` in a *pattern* mode:

— Java + JML (3.13) —

```
class UnlockScreen extends LinearLayoutWithDefaultTouchRecipient
      implements KeyguardScreen,
             KeyguardUpdateMonitor.ConfigurationChangeCallback {
 /*@ invariant mFailedPatternAttemptsSinceLastTimeout>=0;
   @ invariant mTotalFailedPatternAttempts>=0;
   @ public ghost boolean keyguarddone;
  */
```

— Java + JML (3.13) —

In the above code, there are two defined invariants for this class. As it is described earlier, `mFailedPatternAttemptsSinceLastTimeout` field is used to count the failed attempts since the last timeout. The default number is 5. This means that after 5 failed attempts, the user is prevented from trying again for some seconds. On the other hand, the `mTotalFailedPatternAttempts` field is to count the total failed attempts, so that after a specified failed attempt, the system shows a message regarding the forgotten pattern on footer.

— JML (3.14) —

```
/*@ public normal_behaviour
  @ requires mLockPatternUtils.checkPattern(pattern);
  @ ensures mCallback.authenticationOK;
  @ assignable mCallback.*,
  @ mUnlockHeader.mCharWrapper.mChars,
  @ mUnlockIcon.*,
  @ mLockPatternView.mPatternDisplayMode,
                          mLockPatternView.mAnimatingPeriodStart,
  @ mLockPatternView.mInProgressX,mLockPatternView.mInProgressY,
  @ mLockPatternView.mPatternDrawLookup[*][*],
  @ mLockPatternView.mPrivateFlags,
  @ \object_creation(RandomAccessFile),\object_creation(byte[]),
  @ \object_creation(IllegalStateException);
  @
  @ also
  @
  @ requires !mLockPatternUtils.checkPattern(pattern)         &&
                      pattern.size() >=
  @                 LockPatternUtils.MIN_PATTERN_REGISTER_FAIL &&
  @ mFailedPatternAttemptsSinceLastTimeout <
  @              LockPatternUtils.FAILED_ATTEMPTS_BEFORE_TIMEOUT;
  @ ensures !mCallback.authenticationOK &&
  @ mTotalFailedPatternAttempts==
  @              \old(mTotalFailedPatternAttempts)+1 &&
  @ mFailedPatternAttemptsSinceLastTimeout==
  @              \old(mFailedPatternAttemptsSinceLastTimeout)+1;
  @ assignable mCallback.authenticationOK,
  @            mFailedPatternAttemptsSinceLastTimeout,
  @ mTotalFailedPatternAttempts,
  @ mLockPatternView.mPatternDisplayMode,
  @                       mLockPatternView.mAnimatingPeriodStart,
  @ mLockPatternView.mInProgressX,mLockPatternView.mInProgressY,
  @ mLockPatternView.mPatternDrawLookup[*][*],
  @ mLockPatternView.mPrivateFlags,
  @ mUnlockHeader.mCharWrapper.mChars,
  @ mCallback.*, mUnlockIcon.*,
  @ \object_creation(RandomAccessFile),\object_creation(byte[]);
  @
  @ also
```

```
40   @
41   @ requires !mLockPatternUtils.checkPattern(pattern) &&
42   @ mFailedPatternAttemptsSinceLastTimeout >=
43   @         LockPatternUtils.FAILED_ATTEMPTS_BEFORE_TIMEOUT;
44   @ ensures !mCallback.authenticationOK &&
45   @          mFailedPatternAttemptsSinceLastTimeout==0;
46   @ assignable mCountdownTimer,mCallback.authenticationOK,
47   @ mFailedPatternAttemptsSinceLastTimeout,
48   @ mLockPatternView.mPatternDisplayMode,
49   @ mLockPatternView.mAnimatingPeriodStart,
50   @ mLockPatternView.mInProgressX,mLockPatternView.mInProgressY,
51   @ mLockPatternView.mPatternDrawLookup[*][*],
52   @ mLockPatternView.mPrivateFlags,
53   @ mUnlockHeader.mCharWrapper.mChars,
54   @ mUnlockIcon.*,
55   @ mFooterForgotPattern.*,
56   @ mFooterNormal.*,
57   @ \object_creation(mCountdownTimer),
58   @ \object_creation(RandomAccessFile),
59   @ \object_creation(byte[]); */
```

JML (3.14)

The `requires` clause in line 2 represents the first condition when the `checkPattern` method compares the drawn pattern with a correct one and returns `true`. It means that the drawn pattern is correct. In this case in line 3 (listing 3.14) `mCallback.authenticationOK` would be `true`. The field `authenticationOK` is a boolean auxiliary field declared in interface `KeyguardViewCallback` as follows:

Java + JML (3.15)

```
public interface KeyguardViewCallback {
...
//@ public instance model boolean  authenticationOK;
...
/*@ public normal_behaviour
  @ requires true;
  @ ensures authenticationOK==authenticated;
  @ assignable authenticationOK;*/
    void keyguardDone(boolean authenticated);
```

Java + JML (3.15)

As we can see in diagram 3.6, the `onPatternDetected` method calls back the `keyguardDone` method of `KeyguardViewMediator` class to return the result of

pattern matching. On the other hand this class implements `KeyguardViewCallback` interface. We need to be sure that the result of pattern matching is returned back correctly to the caller.
The specification of `keyguardDone` method in listing(3.15) states that `authenticationOK` should always be equal to `authenticated` parameter.

The `keyguardDone` method in `KeyguardViewMediator` class is implemented as follows:

— Java + JML (3.16) —

```
1  public class KeyguardViewMediator implements
2                                  KeyguardViewCallback,..
3   {
4  public void keyguardDone(boolean authenticated) {
5          ...
6   if (authenticated) {
7       mUpdateMonitor.clearFailedAttempts();
8     }
9   if (mExitSecureCallback != null) {
10        mExitSecureCallback.onKeyguardExitResult(authenticated);
11             ...
12        }
13  }
```

— Java + JML (3.16) —

In listing(3.14), in the first situation we expect the `mCallback.authenticationOK` ghost field to be `true`. It is stated in the `ensures` clause in line 3 listing(3.14). In this state, failed attempts will be reset to zero in the `keyguardDone` method in line 7 listing(3.16).
The `assignable` clause represents all fields that may change after running the method. This means the fields of the current class and also fields of other classes which may change through calling a method of them in this method.

The other condition starts in line 17 listing(3.14) after the `also` keyword. It describes the situation that the drawn pattern is wrong and the `checkPattern` method returns `false`. In this condition, `mCallback.authenticationOK` field should be `false`, also the two fields `mFailedPatternAttemptsSinceLastTimeout` and
`mTotalFailedPatternAttempts` will be increased by one.
The last situation describes the state that the number of failed attempts hits to the maximum allowed attempts before the time out.
So `mFailedPatternAttemptsSinceLastTimeout` field will reset to zero and also `mCallback.authenticationOK` will be `false`.

— Java + JML (3.17) —

```
public void onPatternDetected(List<LockPatternView.Cell>
                                                    pattern) {
if (mLockPatternUtils.checkPattern(pattern)) {
    mLockPatternView.setDisplayMode(
                         LockPatternView.DisplayMode.Correct);
    mUnlockIcon.setVisibility(View.GONE);
    mUnlockHeader.setText("");
    mCallback.keyguardDone(true);

   }
else {
     mCallback.pokeWakelock(UNLOCK_PATTERN_WAKE_INTERVAL_MS);
     mLockPatternView.setDisplayMode(
                        LockPatternView.DisplayMode.Wrong);
if (pattern.size() >=
              LockPatternUtils.MIN_PATTERN_REGISTER_FAIL) {
     mTotalFailedPatternAttempts++;
     mFailedPatternAttemptsSinceLastTimeout++;
     mCallback.reportFailedPatternAttempt();
   }
if (mFailedPatternAttemptsSinceLastTimeout >=
          LockPatternUtils.FAILED_ATTEMPTS_BEFORE_TIMEOUT) {
     long deadline = mLockPatternUtils.
                               setLockoutAttemptDeadline();
     handleAttemptLockout(deadline);
     return;
   }
 mUnlockIcon.setVisibility(View.VISIBLE);
 mUnlockHeader.setText(R.string.lockscreen_pattern_wrong);
 mLockPatternView.postDelayed(
 mCancelPatternRunnable,
 PATTERN_CLEAR_TIMEOUT_MS);
    }
   }
}
```

— Java + JML (3.17) —

3.3 Specifying the Contact Application

The *Contact* application is one of the most important and useful applications in a cell phone.
A *Contact* application is like a phone book that the user can add some contact information to it, such as: name, telephone number, address, email address, and so on. The user can use these pieces of information to call people or send them an email, etc.
The following figure shows a launched Contact application.

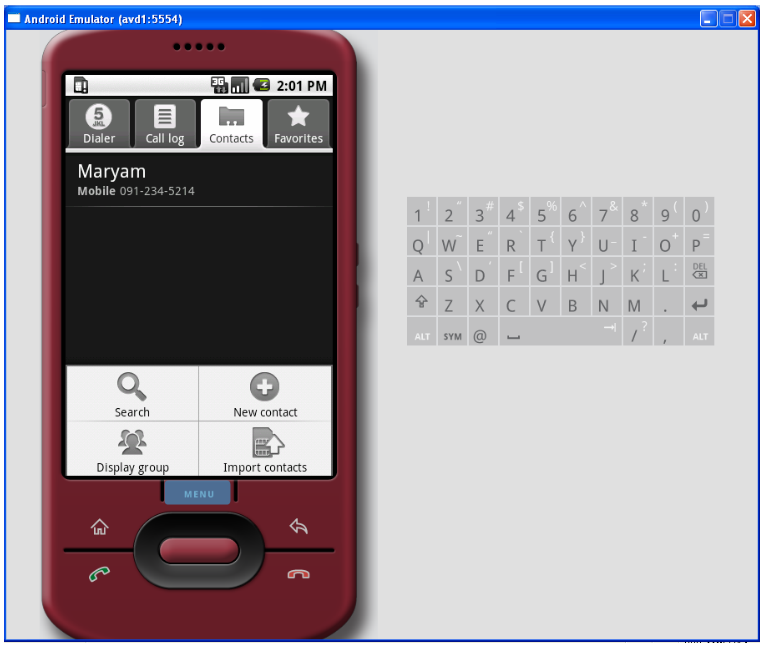


Figure 3.7: Contact Application

Because of the importance of the functions of *Add Contact* and *Edit Contact* in this application, the correctness of *New Contact* and *Edit Contact* activities is our interest.
We continue with explaining these activities and their functionality, afterwards we try to specify them in JML.

As we can see in Fig. 3.7, to create a new contact we should select the *New Contact* option in the *Menu* list. Then we see the next activity to enter the necessary information (see Fig. 3.8).

On the other hand, if the user selects an available contact from the list, an activity with saved information regarding the selected contact is shown to the user. The user is able to select the *Edit* option to edit information, see Fig. 3.9. In this case also the user encounters an new activity to edit old information.

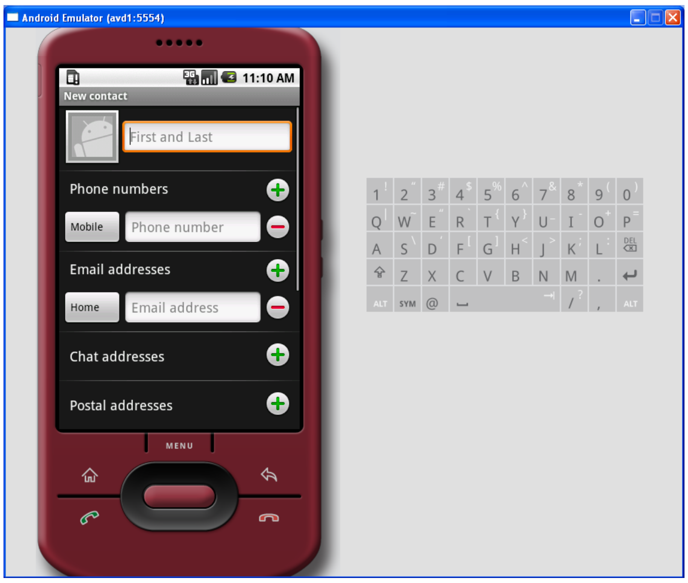
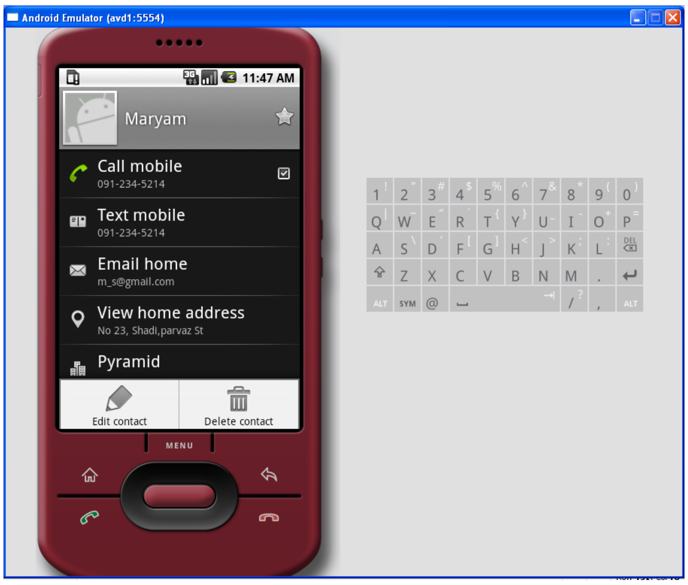


Figure 3.8: New Contact



Maryam
Call mobile
091-234-5214
Text mobile
091-234-5214
Email home
m_s@gmail.com
View home address
No 23, Shadi,parvaz St
Pyramid
Edit contact
Delete contact

Figure 3.9: Edit Contact

At the end, in both cases, the user selects the *Done* button to save data properly. So, our goal is to prove the correctness of the *save* functionality. To do this, we first explain the involved APIs and next, specify formally some of them in the next section.

Specification Of Involved APIs

The diagram in Fig. 3.10 indicates the sequence of involved activities.

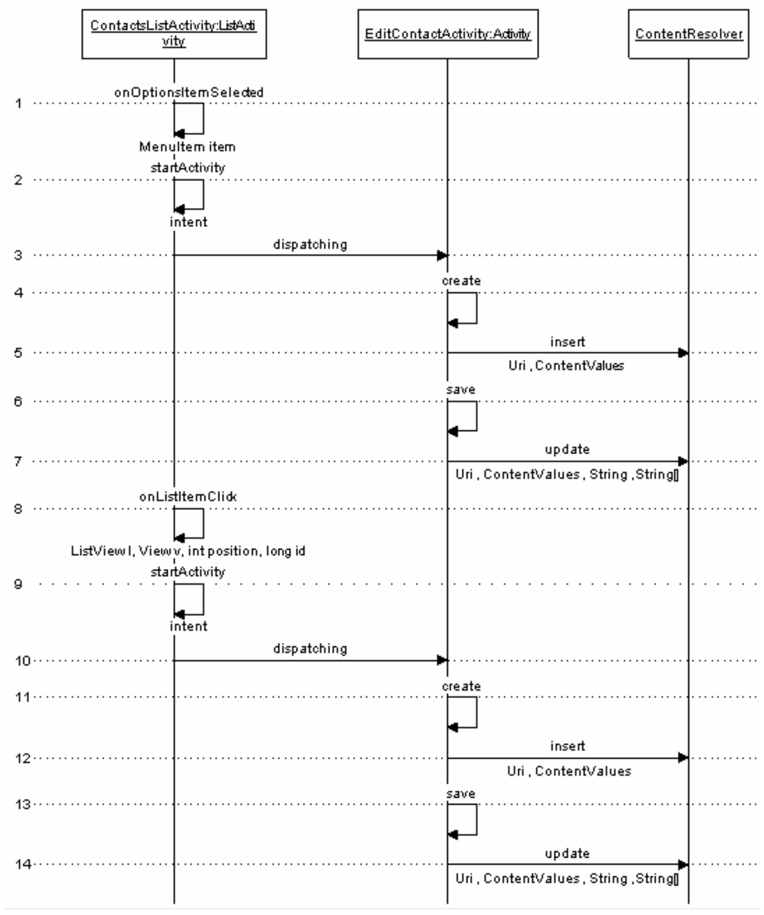


Figure 3.10: Contacts Diagram

EditContactActivity Class As we can see in the diagram(Fig. 3.10) after selection of *New Contact* from the menu, either selecting one item in the list of contacts, the `EditContact` activity will be shown to the user, and the user can create a new contact, or edit data of an existent one. In this class, our goal is to specify the correctness of the methods `create` and `save` which are responsible to create a new contact or edit an existing one.
Before focusing on these methods we proceed to describe how the database in android is organized and how it communicates with it, and also which android component is used for this communication.

As we described in section 1.4, to communicate with databases in android, we always need a content provider component. Actually, this component is re-

sponsible to control a specific type of data and make that type of data available to other applications. This data can be a table in the database or a file system. To do this, every content provider implements a set of methods so that every application can use them to save or retrieve data of the type it controls. Further, Other applications call these methods by using a ContentResolver object and call its methods instead of calling ContentProvider methods. A ContentResolver provides applications with access to the content model and acts as a wrapper for the content providers. Also it can communicate to any content provider.
So, to have a connection with a database we need to acquire the ContentResolver of the application and later on by calling its methods we can access different methods of different content providers.

The following list shows implementation and specification of the `create` method. This method takes the entered data and saves it to a new contact.

— Java + JML (3.18) —

```
/*@ public model int count;
  @ invariant count<=getAllEntries().length;
  @ public ghost URI insertedContactUri;
  @ represents count<-ContactEntryAdapter.
  @                             countEntries(mSections, false);
  @*/

/*@ public model nullable EditEntry[] getAllEntries() {
  @ EditEntry[] entry = new EditEntry[count];
  @ for (int i=0;i<count;i++)
  @       entry[i]=ContactEntryAdapter.getEntry(mSections, i, false);
  @ return entry};
  @*/

/*@ public normal_behavior
  @ requires (\forall int i;i<count && i>=0;
  @                                getAllEntries()[i]!=null &&
  @                                getAllEntries()!=null   &&
  @            (\exists  EditEntry entry ;
  @                entry==getAllEntries()[i]&&
  @                !People.CUSTOM_RINGTONE.equals(entry.column) &&
  @                !People.SEND_TO_VOICEMAIL.equals(entry.column)&&
  @                 entry.getData()!=null));
  @ ensures mResolver.query(insertedContactUri,CONTACT_PROJECTION,
  @                             null,null,null)!=null &&
  @                            mResultCode==RESULT_OK;
  @ assignable  mResolver.*,mPhoto.*,mUri.*,mUri,
  @ mResultCode,\object_creation(ContentValues),
  @ \object_creation(ByteArrayOutputStream),
  @ \object_creation(String),
  @ \object_creation(Intent),
  @ \object_creation(ByteArrayOutputStream);
  @ also
  @ requires (\forall int i;i<count &&
  @              i>=0;getAllEntries()[i]==null &&
  @             !People.CUSTOM_RINGTONE.equals(entry.column) &&
  @             !People.SEND_TO_VOICEMAIL.equals(entry.column)&& );
  @ ensures  mResultCode==RESULT_CANCELED;
  @ assignable mResolver.*,mPhoto.*,mResultCode,
  @           \object_creation(ContentValues),
  @           \object_creation(ByteArrayOutputStream),
  @           \object_creation(String);
  @*/

 private void create() {
  ContentValues values = new ContentValues();
  String data;
  int numValues = 0;
  // Create the contact itself
  final String name = mNameView.getText().toString();
  if (name != null && TextUtils.isGraphic(name)) {
       numValues++;
     }
   values.put(People.NAME, name);
   values.put(People.PHONETIC_NAME,
                      mPhoneticNameView.getText().toString());
        // Add the contact to the My Contacts group
   Uri contactUri = People.createPersonInMyContactsGroup(
                                            mResolver,values);
     /*@ set insertedContactUri=contactUri;*/
 ...
  if (numValues == 0) {
          mResolver.delete(contactUri, null, null);
          setResult(RESULT_CANCELED);
      }
 else {
```

```
67          mUri = contactUri;
68          setResult(RESULT_OK, resultIntent);
69           ...
70        }
71  ...
72  }
```

Java + JML (3.18)

Before describing the specification of the method, we need to know about the operation of the method. We proceed to describe its functionality.

Actually, to create a new contact and adding it to the contact list, first, we need to acquire the ContentResolver of the application. By this object we can communicate with the ContentProvider which is responsible to control *Contact* table in the database.
By sending a specific URI(Uniform Resource Identifier) which is corresponding to the *Contact* table to the Content Resolver we can access the desired table, and later on we can call `insert` or `update` methods on this table.

As we have already mentioned in the previous part, to specify the correctness of this method, we need to be assured that after creating a new contact and adding it to the contact list, it is actually added to the *Contact* table. We assure this by a query on the *Contact* table and looking for the just added contact. If the added contact was actually there, so, the correctness of the method is assured.

To specify the method first, in line 1 we have declared a model field `count`. It is representing the count of entries in all sections of a new contact, see Fig. 3.8. We use this field to count all entries in all sections visible to the user e.g(Name, Phone Number, Email Number,...).
To count entries we call a query in the `ContactEntryAdapter` class. This class is an adapter which provides access to data items of a contact in the database. This class is also responsible for making a view for each item in the data set.
In line 8 to 13, we have described a `model` method. This method is an auxiliary method to help us in the specification purpose. This method returns an array of all entries in all sections. We use this method in specification of the `create` and `save` methods to check if the entries in Fig. 3.8 have data or they are empty. If at least on of the entries(except some of them which have default value) has data, we should have a new added row in *contact* table corresponding to the data of the entries.

In line 16 in the `requires` clause, we state that the user needs to enter at least one entry(e.g., name or phone number). In the `ensures` clause we state that if the requirement is fulfilled, after returning method, the created contact

with provided information should be added to the database . We check this by calling a query on the "mResolver" object. This object is a ContentResolver which provides access to the content model.
As is mentioned earlier, by calling the methods of ContentResolver we actually call the methods of the ContentProvider corresponding to a specific table. Further, `query` is one of the ContentResolver methods which we can use to access a specific row or a specific part of a data set. By using this method plus information regarding the added contact we can access that row of the specific *contact* table.
The first parameter of this method is the URI corresponding to the added contact to the *contact* table. This field is set in line 60, when a new contact is created. The second field is a static `String` array which consists of a list of all fields of a Contact. The query retrieves a cursor over the result set if there is any row corresponding to the specified URI, otherwise it returns null.
Also the result code which propagates back to the originating activity should be `RESULT_OK`. That means the operation has been successful.
In line 34 we stated the other situation, when all entries are empty and the user did not enter any data. So after finishing the method we expect the return value to be `RESULT_CANCELED`.

The next method which we are going to specify is the `Save` method. This method saves the various fields to the existing contact.
Actually, if the user selects a contact in the list, the contact's data is shown to edit. Afterwards, if the user selects the *Done* button to save the changed information, the `save` method is called. The implementation and specification of the method is illustrated in the following list:

— Java + JML (3.19) ————————————————

```
// invariant count<=getAllEntries().length;
/*@ public normal_behavior
 @ requires \forall int i;i<count && i>=0 ;
 @                        (\exists EditEntry entry ;
 @                               entry==getAllEntries()[i] &&
 @                                entry.getData()!=null )  &&
 @          !People.CUSTOM_RINGTONE.equals(entry.column) &&
 @          !People.SEND_TO_VOICEMAIL.equals(entry.column);
 @ ensures (\forall int i;i<count && i>=0;
 @           (mResolver.query(getAllEntries()[i].uri,
 @                         CONTACT_PROJECTION,null,null,null).
 @                         getString(getAllEntries()[i].column))
 @                          .equals(getAllEntries()[i].getData())
 @                && mResultCode==RESULT_OK;
```

```
15   @ assignable \fields_of(mResolver),
16   @     \fields_of(mPhoto),mResultCode,
17   @     \object_creation(ContentValues),\object_creation(Intent),
18   @     \object_creation(ByteArrayOutputStream);
19   @ also
20   @ requires (\forall int i;i<count && i>=0 ;
21   @               getAllEntries()[i].getData()==null )&&
22   @           !People.CUSTOM_RINGTONE.equals(entry.column) &&
23   @           !People.SEND_TO_VOICEMAIL.equals(entry.column);
24   @ ensures mResultCode==RESULT_CANCELED && mUri==null;
25   @ assignable \fields_of(mResolver),\fields_of(mPhoto),
26   @        mResultCode,\fields_of(mUri),
27   @        \object_creation(ContentValues),
28   @        \object_creation(ByteArrayOutputStream);
29   @*/
30  private void save() {
31    ...
32   mResolver.update(entry.uri, values, null, null);
33    ...
34   if (!People.CUSTOM_RINGTONE.equals(entry.column) &&
35             !People.SEND_TO_VOICEMAIL.equals(entry.column)) {
36                        numValues++;
37                    }
38   ...
39   if (numValues == 0) {
40          // The contact is completely empty, delete it
41          mResolver.delete(mUri, null, null);
42          mUri = null;
43          setResult(RESULT_CANCELED);
44        } else {
45      ...
46   setResult(RESULT_OK, new Intent().setData(mUri));
47  }
```

Java + JML (3.19)

To ensure correctness of the Save method, we have to ensure that the changed data in the `Edit Contact` activity after clicking "Done" button is saved correctly in the database.

To specify this method, first in line 3 we express the precondition of the method in the first case, when at least one of the entries (except Ringtone, Incoming Calls which have default value) is not null.

To be convinced that the method has a correct behavior, we need to call query

for all entries with their specific URI. Later on we compare the result of the query which is the data of the given URI(each of the entries of a contact in the database) with the data of the entry.
We expect them to be equal if the precondition is fulfilled , after running the method. This is illustrated in the `ensures` clause in line 9.
Additionally, in this case, the result of the activity which will propagate back to the originating activity should be `RESULT_OK`.
In the other case, when all entries are empty(except Ringtone,Incoming Calls which have default value), `mResultCode` and `mUri` should be, `RESULT_CANCELED` and null respectively. As they are set in the line 42 to 43.
The assignable clause in lines 15, 25 states that all fields may change during execution of the method.

Chapter 4

Conclusion and Future Work

Today, the mankind life is really dependent on computers and complex softwares. Most businesses run on computers and many people use them heavily in day-to-day routines.
Today computer softwares are used for every imaginable purpose, anywhere from banking to ordering products to looking up recipes.
One of the important usage of computer softwares is in sensitive and critical places such as: car engines, phone devices, aircraft's instruments and so on.
Due to its great complexity, it is inevitable that modern softwares will suffer from the presence of numerous errors. These software bugs are frequent sources of security vulnerabilities, and in safety-critical systems, they are not simply expensive annoyances but can endanger lives.
The growing demand for high availability and reliability of computer systems has led to a formal verification of such systems.
The android platform for mobile devices is one of these critical systems which malfunction in its functionalities can lead to serious financial losses, security or even safety critical issues.

The thesis focused on the specification of the critical android applications, but was carried out with respect to a possible later verification using the KeY system. In addition we concentrated on potentially safety and security critical features of the different applications like that emergency calls send via the EmergencyDialer are eventually dispatched to the carrier.
A functional complete specification for the Android API was not in reach for this thesis for several reasons:

- Android libraries make extensive use of callback methods and asynchron method invocations. JML and the current KeY support for these features are not yet sufficient resp. supported. Enriching the passed result objects by ghost field carrying intermediate results for state information allowed partially to overcome these restrictions on the specification side.

- Specification of third party libraries requires the use of model fields. The KeY version with which the thesis was carried out provided only limited support for model fields and we worked around using ghost fields where possible. The upcoming KeY version using an explicit heap model will provide full support for model fields and allow to actually verify the specifications developed in this thesis.

- The API documentation of the android libraries is rudimentary at best and a major part of the work was put into understanding the functionality of the different involved methods as well as how they interact, e.g., the ScreenManager was particularly hard to comprehend.

At the moment the specified classes are about 1700 lines in 16 classes, namely: `Listener,AlteredCharSequence,CharSequence,CharBuffer, SpannableStringBuilder,EmergencyDialer,InCallScreen,Activity,Intent, PhoneNumberUtils,UnlockScreen,KeyguardViewCallback,SimUnlockScreen, EditContactActivity,AccountUnlockScreen,Instrumentation.`
From the verification of a sample android application, we gained the following insights:

- Performance Issue: Nearly the complete android API needs to be loaded to be able to access the necessary JML invariants and contracts. The loading of the JML enriched skeleton classes currently slows down KeY's startup time to several minutes until the proof obligation browser window is displayed. Further, the amount of memory required to hold the associated data structures is considerable and leads to practical problems on machines with less memory.

- Interactive Steps: To close the proof, it was necessary to comprehend the proof situation at one open proof goal and to apply the necessary of rule applications by hand until the strategies could take over again. Understanding the proof situation required a certain familiarity/expertise with KeY not obvious to a normal Java developer.

Despite these bottlenecks we want to emphasize that we were able to complete the proof using KeY indicating that current state of-the-art provers can be applied to bigger problems and provide a reasonable automation even if the later needs still to be improved. That is in particular interesting as the application itself was relatively small, but the attached API rather large.

In later stages of this thesis support for Java Strings and also the explicit heap version of KeY came available. We expect that if both would have been available at earlier stage of the master thesis, more work on the verification side and even more complete and elegant specifications could have been achieved. As future work it will be interesting to verify the specified safety and security properties using these new developments.

Bibliography

[1] Android developers. What is android? http://developer.android.com/guide/basics/what-is-android.html. [cited at p. 2, 10]

[2] Android developers. Application fundamentals. http://developer.android.com/guide/topics/fundamentals.html. [cited at p. 2, 5, 6, 8, 9]

[3] WIKIPEDIA. Dalvik virtual machine. http://en.wikipedia.org/wiki/Dalvik_virtual_machine, February 2010. [cited at p. 4]

[4] Gray T.Leavens, Erik Poll, Curtis Clifton, Yoonisk Cheon, Clyde Ruby, David Cok, Peter Müller, Joseph Kiniry, and Patrice Chalin. *JML Reference Manual*, draft revision 1.200 edition, February 2007. [cited at p. i, 13, 17]

[5] IBM. Getting started with JML. http://www.ibm.com/developerworks/java/library/j-jml.html, March 2003. [cited at p. 16]

[6] Bernhard Beckert, Reiner Hähnle, and Peter H.Schmitt. *Verification of Object-Oriented Software: The KeY Approach*, volume vol.4334 of LNCS. Springer-Verlag, 2007. [cited at p. 18, 19, 20, 30, 35]

[7] Christian Engel, Christoph Gladisch, Vladimir Klebanov, and Philipp Rümmer. Integrating verification and testing of object-oriented software. In *Tests and Proofs*, April 2008. [cited at p. 18, 20]

[8] Christian Engel, Andreas Roth, Abian Blome, Richard Bubel, and Simon Greiner. Key quicktour for JML. http://www.key-project.org/case_studies/, March 2009. [cited at p. 19, 20]

[9] The KeY Project. Key 1.4.0. http://www.key-project.org/download/key.html, March 2009. [cited at p. 21]

[10] Andreas Roth. Specification and verification of object-oriented components. In *PhD thesis*. Fakultät für Informatik der Universität Karlsruhe, 2006. [cited at p. 34]

[11] WIKIPEDIA. Radio interface layer. http://en.wikipedia.org/wiki/Radio_Interface_Layer, November 2009. [cited at p. 46]

[12] WIKIPEDIA. Emergency telephone number. http://en.wikipedia.org/wiki/Emergency_telephone_number. [cited at p. 47]

Appendices

Appendix A

Specified Classes with JML

— Listener class —

```
package com.example.enterpassword;
import com.example.enterpassword.R;
import android.text.Editable;
import android.view.View;
import android.view.View.OnClickListener;
import android.widget.EditText;
import android.widget.TextView;


public class Listener implements OnClickListener
{
//@ public ghost boolean  userHasLogedIn=false;
private /*@ spec_public  @*/ int passCode=1234;
private /*@ spec_public  @*/ TextView tempText;
private /*@ spec_public  @*/ EditText tempInputtext;
private /*@ spec_public  @*/ int incorrectPINEntered=0;
public Listener(EditText temp_inputtext,TextView temp_text)
{
this.tempInputtext=temp_inputtext;
this.tempText=temp_text;
}

/*@ private normal_behavior
@ requires true;
@ ensures \result >= 0;
@ assignable \nothing;
@*/
private int getPassword(EditText input) {
return Integer.valueOf(input.getText().toString()).intValue();
}


/*@ public normal_behaviour
@
@ requires tempInputtext.getText().length()>0 &&
incorrectPINEntered<3 && !userHasLogedIn;
@ ensures true ;
@ assignable userHasLogedIn,incorrectPINEntered;
@
@ also
@ public normal_behavior
@
@ requires tempInputtext.getText().length()==0  && !userHasLogedIn ;
@ ensures !userHasLogedIn ;
@ assignable \nothing;
@
@ also
@ public normal_behavior
@ requires incorrectPINEntered>=3 && !userHasLogedIn ;
@ ensures !userHasLogedIn ;
@ assignable \nothing;
@ */

public void onClick(View v)
{
if( tempInputtext.getText().length()==0 )
{
tempText.setText("Please␣Enter␣The␣Password!");
return;
}
int intPass;
if (incorrectPINEntered < 3) {
intPass=getPassword(tempInputtext);
if (isPasswordCorrect(intPass)) {
incorrectPINEntered = 0;
} else {
```

```
incorrectPINEntered++;
}
}
else
{
tempText.setText("You␣have␣entered␣the␣password
incorrectly␣more␣than␣3␣times!");
return;
}
}

/*@ public normal_behaviour
@
@ requires pwd==passCode && !userHasLogedIn;
@ ensures userHasLogedIn && \result==true;
@ assignable userHasLogedIn;
@
@ also
@ public normal_behavior
@
@ requires pwd!=passCode && !userHasLogedIn;
@ ensures !userHasLogedIn && \result==false;
@ assignable userHasLogedIn;
@ */
public boolean isPasswordCorrect(int pwd)
{
if(pwd==passCode)
{
tempText.setText("The␣Password␣is␣correct!");
//@set userHasLogedIn=true;
return true;
}

else
{
tempText.setText("The␣Password␣is␣not␣correct!");
//@ set userHasLogedIn=false;
return false;
}
}
```

```
}
```

—————————————————— Listener class ——

—— AlteredCharSequence class ——————————————————

```
package android.text;

public class AlteredCharSequence extends java.lang.Object implements
ava.lang.CharSequence, android.text.GetChars
{

/*@ public model int len; @*/
//@ public invariant len >=0;
...
/*@ public normal_behavior
@ requires true;
@ ensures \result == len;
@*/
public /*@ pure @*/ int length();
}
```

—————————————————— AlteredCharSequence class ——

—— CharSequence class ——————————————————

```
package java.lang;

public interface CharSequence
{
/*@ public normal_behaviour
@ requires true;
@ ensures \result>=0;
@ assignable \nothing;
@*/
public /*@pure*/int length();

/*@ public normal_behaviour
@ requires true;
@ ensures true;
@ assignable \nothing;
@*/
public java.lang.String toString();
}
```

—————————————————— CharSequence class ——

—— SpannableStringBuilder class ——

```
package android.text;

public class SpannableStringBuilder extends java.lang.Object
implements java.lang.CharSequence,
android.text.GetChars, android.text.Spannable,
android.text.Editable, java.lang.Appendable
{
/*@ public model int len; @*/
//@ public invariant len >=0;

/*@ public normal_behavior
@ requires true;
@ ensures \result == len ;
@*/
public /*@ pure @*/ int length();
```

—— SpannableStringBuilder class ——

—— CharBuffer class ——

```
package java.nio;

public abstract class CharBuffer extends java.nio.Buffer implements
java.lang.Comparable, java.lang.CharSequence,
java.lang.Appendable, java.lang.Readable
{
/*@ public model int len; @*/
//@ public invariant len >=0;

/*@ public normal_behavior
@ requires true;
@ ensures \result == len;
@*/
public /*@ pure @*/ int length();
```

—— CharBuffer class ——

—— EmergencyDialer class ——

```
//package com.android.phone;
package com.android.phone;


public class EmergencyDialer extends Activity
```

```
implements View.OnClickListener,
View.OnLongClickListener,
View.OnKeyListener,
TextWatcher {

/*@ public model  String  number;
@ public ghost boolean isCallDispatched;
@ invariant number==mDigits.getText().toString();
@*/

private /*@ nullable*/ ToneGenerator mToneGenerator;
private /*@nullable*/ Object mToneGeneratorLock = new Object();

/*@
@ public normal_behaviour
@ requires !DBG && keyCode==KEYCODE_CALL &&
@                              !(TextUtils.isEmpty(number))
@        &&PhoneNumberUtils.isEmergencyNumber(number)
@        && (number!=null ||TextUtils.isGraphic(number);
@ ensures iscalldispatched==true && \result==true;
@ assignable iscalldispatched,mFinished,intent.call_dispatched,
@        \object_creation(Intent),\object_creation(String);
@ also
@ requires !DBG && keyCode==KEYCODE_CALL && !(TextUtils.isEmpty(number)) &&
@!PhoneNumberUtils.isEmergencyNumber(number);
@ ensures iscalldispatched==false && \result==true;
@ assignable iscalldispatched,\object_creation(String),
@                      \fields_of(mDigits.getText());
@ also
@ requires !DBG && keyCode==KEYCODE_CALL && (TextUtils.isEmpty(number))
@ ensures \result==true;
@ assignable mFinished;

*/
@Override
public boolean onKeyDown(int keyCode, KeyEvent event) {
switch (keyCode) {
case KeyEvent.KEYCODE_CALL: {
if (TextUtils.isEmpty(mDigits.getText().toString())) {
// if we are adding a call from the InCallScreen and the phone
// number entered is empty, we just close the dialer to expose
// the InCallScreen under it.
finish();
} else {
// otherwise, we place the call.
placeCall();
}
return true;
}
}
return super.onKeyDown(keyCode, event);
}

private void keyPressed(int keyCode) {
KeyEvent event = new KeyEvent(KeyEvent.ACTION_DOWN, keyCode);
mDigits.onKeyDown(keyCode, event);
}

/*@ public normal_behaviour
@ requires !DBG && keyCode==KeyEvent.KEYCODE_ENTER &&
@ view.getId()== R.id.digits &&!(TextUtils.isEmpty(number)) &&
@ PhoneNumberUtils.isEmergencyNumber(number) &&
@     (number!=null ||TextUtils.isGraphic(number);
@ ensures  iscalldispatched==true && \result==true;
@ assignable iscalldispatched,mFinished,intent.call_dispatched,
@           \object_creation(Intent),\object_creation(String);
@ also
@ requires !DBG && keyCode==KeyEvent.KEYCODE_ENTER &&
@                                  view.getId()== R.id.digits
@ &&!(TextUtils.isEmpty(number)) &&
@!PhoneNumberUtils.isEmergencyNumber(number);
@ ensures iscalldispatched==false && \result==true;
@ assignable iscalldispatched,\object_creation(String),
@                                \fields_of(mDigits.getText());
@ also
@ requires  !DBG && view.getId() != R.id.digits;
@ ensures \result==false;
@ assignable \nothing;
*/
public boolean onKey(View view, int keyCode, KeyEvent event) {
switch (view.getId()) {
case R.id.digits:
```

```
if (keyCode == KeyEvent.KEYCODE_ENTER) {
placeCall();
return true;
}
break;
}
return false;
}

/*@ public normal_behaviour
@ requires  !DBG && view.getId()== R.id.digits &&
@           !(TextUtils.isEmpty(number)) &&
@ PhoneNumberUtils.isEmergencyNumber(number)&&
@       (number!=null ||TextUtils.isGraphic(number);
@ ensures  iscalldispatched==true ;
@ assignable iscalldispatched,mFinished,intent.call_dispatched,
@ \object_creation(Intent),\object_creation(String);
@ also
@ requires  !DBG && view.getId()== R.id.digits &&
@           !(TextUtils.isEmpty(number)) &&
@!PhoneNumberUtils.isEmergencyNumber(number);
@ ensures iscalldispatched==false ;
@ assignable iscalldispatched,\object_creation(String),
@                       \fields_of(mDigits.getText());
@*/
public void onClick(View view) {
final Editable digits = mDigits.getText();

switch (view.getId()) {
case R.id.one: {
playTone(ToneGenerator.TONE_DTMF_1);
keyPressed(KeyEvent.KEYCODE_1);
return;
}
case R.id.two: {
playTone(ToneGenerator.TONE_DTMF_2);
keyPressed(KeyEvent.KEYCODE_2);
return;
}
case R.id.three: {
playTone(ToneGenerator.TONE_DTMF_3);
```

```
keyPressed(KeyEvent.KEYCODE_3);
return;
}
case R.id.four: {
playTone(ToneGenerator.TONE_DTMF_4);
keyPressed(KeyEvent.KEYCODE_4);
return;
}
case R.id.five: {
playTone(ToneGenerator.TONE_DTMF_5);
keyPressed(KeyEvent.KEYCODE_5);
return;
}
case R.id.six: {
playTone(ToneGenerator.TONE_DTMF_6);
keyPressed(KeyEvent.KEYCODE_6);
return;
}
case R.id.seven: {
playTone(ToneGenerator.TONE_DTMF_7);
keyPressed(KeyEvent.KEYCODE_7);
return;
}
case R.id.eight: {
playTone(ToneGenerator.TONE_DTMF_8);
keyPressed(KeyEvent.KEYCODE_8);
return;
}
case R.id.nine: {
playTone(ToneGenerator.TONE_DTMF_9);
keyPressed(KeyEvent.KEYCODE_9);
return;
}
case R.id.zero: {
playTone(ToneGenerator.TONE_DTMF_0);
keyPressed(KeyEvent.KEYCODE_0);
return;
}
case R.id.pound: {
playTone(ToneGenerator.TONE_DTMF_P);
keyPressed(KeyEvent.KEYCODE_POUND);
```

```
return;
}
case R.id.star: {
playTone(ToneGenerator.TONE_DTMF_S);
keyPressed(KeyEvent.KEYCODE_STAR);
return;
}
case R.id.digits: {
placeCall();
return;
}
case R.id.backspace: {
keyPressed(KeyEvent.KEYCODE_DEL);
return;
}
}
}


/*@ public normal_behaviour
@ requires !DBG &&
@          PhoneNumberUtils.isEmergencyNumber(number)  ;
@ ensures     isCallDispatched;
@ assignable  isCallDispatched,mFinished,
@             intent.callDispatched,\object_creation(Intent);
@ also
@ requires !DBG &&
@ !(PhoneNumberUtils.isEmergencyNumber(number));
@ ensures   !isCallDispatched;
@ assignable isCallDispatched;
@*/

void placeCall() {
//@ set iscalldispatched = false;
final String number = mDigits.getText().toString();
if (PhoneNumberUtils.isEmergencyNumber(number)) {
if (DBG) Log.d(LOG_TAG, "placing␣call␣to␣" + number);

// place the call if it is a valid number
if (number == null || !TextUtils.isGraphic(number)) {
// There is no number entered.
```

```
playTone(ToneGenerator.TONE_PROP_NACK);
return;
}
Intent intent = new Intent(Intent.ACTION_CALL_EMERGENCY);
intent.setData(Uri.fromParts("tel", number, null));
intent.setFlags(Intent.FLAG_ACTIVITY_NEW_TASK);
//@ set intent.call_dispatched = false;
startActivity(intent);
//@ set iscalldispatched = intent.call_dispatched;


finish();
} else {
if (DBG) Log.d(LOG_TAG, "rejecting␣bad␣requested␣number␣" + number);

// erase the number and throw up an alert dialogue.
mDigits.getText().delete(0, mDigits.getText().length());
displayErrorBadNumber(number);
}
}
}
```

EmergencyDialer class

InCallScreen class

```
package com.android.phone;
public class InCallScreen extends Activity
implements View.OnClickListener, View.OnTouchListener,
CallerInfoAsyncQuery.OnQueryCompleteListener {

/*@ public model InCallInitStatus okToCallState;
@ public ghost boolean isRadioOn;
@ represents okToCallState<-checkIfOkToInitiateOutgoingCall();
@ public model int state;
@ represents state<-mPhone.getServiceState().getState();
@*/

/*@ public normal_behaviour
@ requires
@     !DBG  &&
@     intent.getAction() == Intent.ACTION_CALL_EMERGENCY   &&
@     intent.getFlags()  == Intent.FLAG_ACTIVITY_NEW_TASK  &&
@     intent.getData().getScheme().equals("tel")           &&
@     !intent.callDispatched                              &&
@     PhoneNumberUtils.isEmergencyNumber(
@              intent.getData().getSchemeSpecificPart() )  &&
@     (okToCallStatus == InCallInitStatus.SUCCESS );
@ ensures intent.callDispatched && \result==
InCallInitStatus.SUCCESS;
@ assignable intent.callDispatched,(mDialer.*);
@
@ also
@
@ requires
@        !DBG                                           &&
@        okToCallStatus == InCallInitStatus.POWER_OFF    &&
@        PhoneNumberUtils.isEmergencyNumber(
intent.getData().getSchemeSpecificPart() )  &&
@        !isRadioOn;
@
@ ensures
@        isRadioOn ==>(\result==InCallInitStatus.SUCCESS  &&
mFinished && intent.callDispatched );
@ assignable isRadioOn,mFinished,intent.callDispatched;
@
@ also
@
@ public exceptional_behaviour
@ requires
@      !DBG                                             &&
@      intent.getData().getScheme().equals("voicemail")  &&
@      (intent.getData().getSchemeSpecificPart()  == null
@       || TextUtils.isEmpty(intent.getData().
@                               getSchemeSpecificPart() ));
@
@ signals (Exception e) e instanceof
@            PhoneUtils.VoiceMailNumberMissingException &&
@                    \result==okToCallStatus ;
@ assignable \\nothing;


private InCallInitStatus placeCall(Intent intent) {

if (VDBG) log("placeCall()...␣␣intent␣=␣" + intent);
```

```
String number;

// Check the current ServiceState to make sure it's OK
// to even try making a call.
InCallInitStatus okToCallStatus = checkIfOkToInitiateOutgoingCall();

try {
number = getInitialNumber(intent);
} catch (PhoneUtils.VoiceMailNumberMissingException ex) {
// If the call status is NOT in an acceptable state, it
// may effect the way the voicemail number is being
// retrieved.  Mask the VoiceMailNumberMissingException
// with the underlying issue of the phone state.
if (okToCallStatus != InCallInitStatus.SUCCESS) {
if (DBG) log("Voicemail␣number␣not␣reachable␣in␣current␣SIM␣card␣state.");
return okToCallStatus;
}
if (DBG) log("VoiceMailNumberMissingException␣from␣getInitialNumber()");
return InCallInitStatus.VOICEMAIL_NUMBER_MISSING;
}

if (number == null) {
Log.w(LOG_TAG, "placeCall:␣couldn't␣get␣a
␣␣␣␣␣␣␣phone␣number␣from␣Intent␣" + intent);
return InCallInitStatus.NO_PHONE_NUMBER_SUPPLIED;
}

boolean isEmergencyNumber =
       PhoneNumberUtils.isEmergencyNumber(number);
boolean isEmergencyIntent =
       Intent.ACTION_CALL_EMERGENCY.equals(intent.getAction());

if (isEmergencyNumber && !isEmergencyIntent) {
Log.e(LOG_TAG, "Non-CALL_EMERGENCY␣Intent␣" + intent
+ "␣attempted␣to␣call␣emergency␣number␣" + number
+ ".");
return InCallInitStatus.CALL_FAILED;
} else if (!isEmergencyNumber && isEmergencyIntent) {
Log.e(LOG_TAG, "Received␣CALL_EMERGENCY␣Intent␣" + intent
+ "␣with␣non-emergency␣number␣" + number
+ "␣--␣failing␣call.");
return InCallInitStatus.CALL_FAILED;
}

// need to make sure that the state is adjusted if we are ONLY
// allowed to dial emergency numbers AND we encounter an
// emergency number request.
if (isEmergencyNumber && okToCallStatus ==
           InCallInitStatus.EMERGENCY_ONLY) {
okToCallStatus = InCallInitStatus.SUCCESS;
if (DBG) log("Emergency␣number␣detected,␣changing␣state␣to:␣" +
        okToCallStatus);
}

if (okToCallStatus != InCallInitStatus.SUCCESS) {
// If this is an emergency call, we call the emergency call
// handler activity to turn on the radio and do whatever else
// is needed. For now, we finish the InCallScreen (since were
// expecting a callback when the emergency call handler dictates
// it) and just return the success state.
if (isEmergencyNumber && (okToCallStatus ==
                  InCallInitStatus.POWER_OFF)) {
// @ set isradioon = false;
startActivity(intent.setClassName(this,
               EmergencyCallHandler.class.getName()));
/*@ set isRadioOn==( checkIfOkToInitiateOutgoingCall()
==ServiceState.STATE_POWER_OFF)*@/

if (DBG) log("placeCall:␣starting␣EmergencyCallHandler,␣finishing...");
finish();
return InCallInitStatus.SUCCESS;
} else {
return okToCallStatus;
}
}

// We have a valid number, so try to actually place a call:
//make sure we pass along the URI as a reference to the contact.
//@ set intent.call_dispatched=true;
int callStatus = PhoneUtils.placeCall(mPhone, number, intent.getData());
```

```
switch (callStatus) {
case PhoneUtils.CALL_STATUS_DIALED:
if (VDBG) log("placeCall:␣PhoneUtils.placeCall()
␣␣␣␣␣␣␣␣succeeded␣for␣regular␣call␣'"
+ number + "'.");

// Any time we initiate a call, force the DTMF dialpad to
// close.  (We want to make sure the user can see the regular
// in-call UI while the new call is dialing, and when it
// first gets connected.)
mDialer.closeDialer(false);  // no "closing" animation

// Also, in case a previous call was already active (i.e. if
// we just did "Add call"), clear out the "history" of DTMF
// digits you typed, to make sure it doesn't persist from the
// previous call to the new call.
// TODO: it would be more precise to do this when the actual
// phone state change happens (i.e. when a new foreground
// call appears and the previous call moves to the
// background), but the InCallScreen doesn't keep enough
// state right now to notice that specific transition in
// onPhoneStateChanged().
mDialer.clearDigits();

//@ set intent.call_dispatched=true;

return InCallInitStatus.SUCCESS;



case PhoneUtils.CALL_STATUS_DIALED_MMI:
if (DBG) log("placeCall:␣specified␣number␣was␣an␣MMI␣code:
//␣␣␣␣'" + number + "'.");
// The passed-in number was an MMI code, not a regular phone number!
// This isn't really a failure; the Dialer may have deliberately
// fired a ACTION_CALL intent to dial an MMI code, like for a
// USSD call.
//
// Presumably an MMI_INITIATE message will come in shortly
// (and we'll bring up the "MMI Started" dialogue), or else
// an MMI_COMPLETE will come in (which will take us to a
// different Activity; see PhoneUtils.displayMMIComplete()).
return InCallInitStatus.DIALED_MMI;
case PhoneUtils.CALL_STATUS_FAILED:
Log.w(LOG_TAG, "placeCall:␣PhoneUtils.placeCall()␣FAILED␣for␣number␣'"
+ number + "'.");
// We couldn't successfully place the call; there was some
// failure in the telephony layer.
return InCallInitStatus.CALL_FAILED;
default:
Log.w(LOG_TAG, "placeCall:␣unknown␣callStatus␣" + callStatus
+ "␣from␣PhoneUtils.placeCall()␣for␣number␣'" + number + "'.");
return InCallInitStatus.SUCCESS;  // Try to continue anyway...
}
}
/*@ public normal_behaviour
@ requires state==ServiceState.STATE_IN_SERVICE;
@ ensures \result==InCallInitStatus.SUCCESS;
@ assignable \nothing;
@ also
@
@ requires state!=ServiceState.STATE_IN_SERVICE;
@ ensures \result!=InCallInitStatus.SUCCESS;
@ assignable \nothing;
@ also
@
@ public exceptional_behaviour
@ requires state!=ServiceState.STATE_IN_SERVICE &&
@          state!=ServiceState.STATE_POWER_OFF  &&
@          state!=ServiceState.STATE_OUT_OF_SERVICE &&
@          state!=InCallInitStatus.EMERGENCY_ONLY;
@ signals (Exception e) e instanceof  IllegalStateException;
@ assignable \nothing;*/

private /*@ pure*/ InCallInitStatus checkIfOkToInitiateOutgoingCall() {
// Watch out: do NOT use PhoneStateIntentReceiver.getServiceState() here;
// that's not guaranteed to be fresh.  To synchronously get the
// CURRENT service state, ask the Phone object directly:
int state = mPhone.getServiceState().getState();
if (VDBG) log("checkIfOkToInitiateOutgoingCall:␣ServiceState␣=␣" + state);

switch (state) {
```

```
case ServiceState.STATE_IN_SERVICE:
// Normal operation.  It's OK to make outgoing calls.
return InCallInitStatus.SUCCESS;


case ServiceState.STATE_POWER_OFF:
// Radio is explictly powered off.
return InCallInitStatus.POWER_OFF;

case ServiceState.STATE_OUT_OF_SERVICE:
case ServiceState.STATE_EMERGENCY_ONLY:
// The phone is registered, but locked. Only emergency
// numbers are allowed.
return InCallInitStatus.EMERGENCY_ONLY;
default:
throw new IllegalStateException("Unexpected␣ServiceState:␣" + state);
}
}
```

InCallScreen class

PhoneNumberUtils class

```
package android.telephony;
public class PhoneNumberUtils
{
//@ public model static int MIN_MATCH;
//@ invariant MIN_MATCH==5;
/*@ public model static non_null java.lang.String[]
@                                    emergencynumbers;

@ invariant(\exists int i,j ; j>=0 && i>=0 &&
@      i<emergencynumbers.length && j<emergencynumbers.length ;
@      emergencynumbers[i].equals("911")&&
@      emergencynumbers[j].equals("112") );
@ invariant(\forall int i;i>=0 && i<getEmergencyNumbers().length;
@            (\exists int j;j>=0 && j<emergencynumbers.length ;
@             emergencynumbers[j]==getEmergencyNumbers()[i]));
@*/

/*@ public model String[] getEmergencyNumbers() {
@ String numbers = SystemProperties.get("ro.ril.ecclist");
@ String[] _emergency =
@             new String[numbers.split(",").length()];
@ for (int i=0;i< numbers.split(",").length;i++)
@       _emergency[i]=numbers.split(",")[i];
@
@ return _emergency;}


/*@public normal_behaviour
@requires true;
@ensures \result==(c >= '0' && c <= '9');
@assignable \nothing;
@*/

/** True if c is ISO-LATIN characters 0-9 */
public static boolean
isISODigit (char c) {
return c >= '0' && c <= '9';
}
/*@public normal_behaviour
@requires true;
@ensures \result==((c >= '0' && c <= '9') || c == '*' || c == '#');
@assignable \nothing;
@*/

/** True if c is ISO-LATIN characters 0-9, *, # */
public final static boolean
is12Key(char c) {
return (c >= '0' && c <= '9') || c == '*' || c == '#';
}
/*@public normal_behaviour
@requires true;
@ensures \result==((c >= '0' && c <= '9') || c == '*' ||
 c == '#' || c == '+' || c == WILD);
@assignable \nothing;
@*/

/** True if c is ISO-LATIN characters 0-9, *, # , +, WILD  */
public final static boolean
isDialable(char c) {
return (c >= '0' && c <= '9') || c == '*' || c == '#' ||
         c == '+' || c == WILD;
}
```

```
/*@public normal_behaviour
@requires true;
@ensures \result==((c >= '0' && c <= '9') || c == '*' || c == '#'
 || c == '+');
@assignable \nothing;
@*/

/** True if c is ISO-LATIN characters 0-9, *, # , + (no WILD)  */
public final static boolean
isReallyDialable(char c) {
return (c >= '0' && c <= '9') || c == '*' || c == '#' || c == '+';
}

/*@public normal_behaviour
@requires true;
@ensures \result==((c >= '0' && c <= '9') || c == '*' || c == '#'
 || c == '+'
@|| c == WILD || c == WAIT || c == PAUSE);
@assignable \nothing;
@*/

/** True if c is ISO-LATIN characters 0-9, *, # , +,
    WILD, WAIT, PAUSE   */
public final static boolean
isNonSeparator(char c) {
return (c >= '0' && c <= '9') || c == '*' || c == '#' || c == '+'
|| c == WILD || c == WAIT || c == PAUSE;
}

/*@public normal_behaviour
@requires true;
@ensures \result==(c == PAUSE || c == WAIT);
@assignable \nothing;
@*/

/*@public normal_behaviour
@requires (a.length()!=0 && b.length()!=0 && a.length()==b.length()&&
@           a.length()<=MIN_MATCH);
@ensures \result==(a.equals(b));
@assignable \nothing;
```

```
@also
@requires (a.length()!=0 && b.length()!=0 &&
@ (a.length() >=MIN_MATCH && b.length() >=MIN_MATCH));
@ensures \result==(\exists String s;s.length()>=MIN_MATCH ;
@          a.lastIndexOf(s)>=0
@                   && b.lastIndexOf(s)>=0);
@assignable \nothing;
@also
@requires (a.length()==0 || b.length()==0);
@ensures \result==false;
@assignable \nothing;
@*/

public static boolean compare(String a, String b) {
int ia, ib;
int matched;

if (a == null || b == null) return a == b;

if (a.length() == 0 || b.length() == 0) {
return false;
}

ia = indexOfLastNetworkChar (a);
ib = indexOfLastNetworkChar (b);
matched = 0;

while (ia >= 0 && ib >=0) {
char ca, cb;
boolean skipCmp = false;

ca = a.charAt(ia);

if (!isDialable(ca)) {
ia--;
skipCmp = true;
}

cb = b.charAt(ib);

if (!isDialable(cb)) {
```

```
ib--;
skipCmp = true;
}

if (!skipCmp) {
if (cb != ca && ca != WILD && cb != WILD) {
break;
}
ia--; ib--; matched++;
}
}

if (matched < MIN_MATCH) {
int aLen = a.length();

// if the input strings match, but their lengths < MIN_MATCH,
// treat them as equal.
if (aLen == b.length() && aLen == matched) {
return true;
}
return false;
}

// At least one string has matched completely;
if (matched >= MIN_MATCH && (ia < 0 || ib < 0)) {
return true;
}

/*
* Now, what remains must be one of the following for a
* match:
*
*  - a '+' on one and a '00' or a '011' on the other
*  - a '0' on one and a (+,00)<country code> on the other
*     (for this, a '0' and a '00' prefix would have succeeded above)
*/

if (matchIntlPrefix(a, ia + 1)
&& matchIntlPrefix (b, ib +1)
) {
return true;
```

```
}

if (matchTrunkPrefix(a, ia + 1)
&& matchIntlPrefixAndCC(b, ib +1)
) {
return true;
}

if (matchTrunkPrefix(b, ib + 1)
&& matchIntlPrefixAndCC(a, ia +1)
) {
return true;
}

return false;
}

/*@ public normal_behaviour
@ requires number !=null;
@ ensures \result==((\exists int i;i>=0 && i< emergencynumbers ;
@                     number.equals(emergencynumbers[i])));
@ assignable \nothing;
@ also
@ requires number==null;
@ ensures \result==false;
@ assignable \nothing;*/
*/


public static boolean isEmergencyNumber(String/*@ nullable*/ number) {
// Strip the separators from the number before comparing it
// to the list.
number = extractNetworkPortion(number);

// retrieve the list of emergency numbers
String numbers = SystemProperties.get("ro.ril.ecclist");

if (!TextUtils.isEmpty(numbers)) {
// searches through the comma-separated list for a match,
// return true if one is found.
for (String emergencyNum : numbers.split(",")) {
```

```
if (emergencyNum.equals(number)) {
return true;
}
}
// no matches found against the list!
return false;
}

//no ecclist system property, so use our own list.
return (number.equals("112") || number.equals("911"));
}
```

——————————————————————— PhoneNumberUtils class ——

—— UnlockScreen class ———————————————————————

```
package com.android.internal.policy.impl;
class UnlockScreen extends LinearLayoutWithDefaultTouchRecepient
   implements KeyguardScreen, KeyguardUpdateMonitor.
       ConfigurationChangeCallback {

/*@ invariant mFailedPatternAttemptsSinceLastTimeout>=0;
@ invariant mTotalFailedPatternAttempts>=0;
@ public ghost boolean keyguarddone;
@*/


/*@ public normal_behaviour
@ requires mLockPatternUtils.checkPattern(pattern);
@ ensures mCallback.authenticationOK;
@ assignable mCallback.authenticationOK,mCallback.*,
@ mUnlockHeader.mCharWrapper.mChars,
@ mUnlockIcon.*,
@ mLockPatternView.mPatternDisplayMode,
@                       mLockPatternView.mAnimatingPeriodStart,
@ mLockPatternView.mInProgressX,mLockPatternView.mInProgressY,
@ mLockPatternView.mPatternDrawLookup[*][*],
@ mLockPatternView.mPrivateFlags,
@ \object_creation(RandomAccessFile),\object_creation(byte[]),
@ \object_creation(IllegalStateException);
@
@ also
@
@ requires !mLockPatternUtils.checkPattern(pattern)         &&
@                     pattern.size() >=
@                 LockPatternUtils.MIN_PATTERN_REGISTER_FAIL &&
@ mFailedPatternAttemptsSinceLastTimeout <
@              LockPatternUtils.FAILED_ATTEMPTS_BEFORE_TIMEOUT;
@ ensures !mCallback.authenticationOK &&
@ mTotalFailedPatternAttempts==
@              \old(mTotalFailedPatternAttempts)+1 &&
@ mFailedPatternAttemptsSinceLastTimeout==
@              \old(mFailedPatternAttemptsSinceLastTimeout)+1;
@ assignable mCallback.authenticationOK,
@            mFailedPatternAttemptsSinceLastTimeout,
@ mTotalFailedPatternAttempts,
@ mLockPatternView.mPatternDisplayMode,
@                       mLockPatternView.mAnimatingPeriodStart,
@ mLockPatternView.mInProgressX,mLockPatternView.mInProgressY,
@ mLockPatternView.mPatternDrawLookup[*][*],
@ mLockPatternView.mPrivateFlags,
@ mUnlockHeader.mCharWrapper.mChars,
@ mCallback.*, mUnlockIcon.*,
@ \object_creation(RandomAccessFile),\object_creation(byte[]);
@
@ also
@
@ requires !mLockPatternUtils.checkPattern(pattern) &&
@ mFailedPatternAttemptsSinceLastTimeout >=
@         LockPatternUtils.FAILED_ATTEMPTS_BEFORE_TIMEOUT;
@ ensures !mCallback.authenticationOK &&
@          mFailedPatternAttemptsSinceLastTimeout==0;
@ assignable mCountdownTimer,mCallback.authenticationOK,
@ mFailedPatternAttemptsSinceLastTimeout,
@ mLockPatternView.mPatternDisplayMode,
@ mLockPatternView.mAnimatingPeriodStart,
@ mLockPatternView.mInProgressX,mLockPatternView.mInProgressY,
@ mLockPatternView.mPatternDrawLookup[*][*],
@ mLockPatternView.mPrivateFlags,
@ mUnlockHeader.mCharWrapper.mChars,
@ mUnlockIcon.*,
@ mFooterForgotPattern.*,
@ mFooterNormal.*,
@ \object_creation(mCountdownTimer),
@ \object_creation(RandomAccessFile),
@ \object_creation(byte[]); */
public void onPatternDetected(List<LockPatternView.Cell>  pattern) {
if (mLockPatternUtils.checkPattern(pattern)) {
mLockPatternView.setDisplayMode(LockPatternView.DisplayMode.Correct);
mUnlockIcon.setVisibility(View.GONE);
mUnlockHeader.setText("");

mCallback.keyguardDone(true);
} else {

mCallback.pokeWakelock(UNLOCK_PATTERN_WAKE_INTERVAL_MS);
mLockPatternView.setDisplayMode(LockPatternView.DisplayMode.Wrong);
if (pattern.size() >= LockPatternUtils.MIN_PATTERN_REGISTER_FAIL) {
mTotalFailedPatternAttempts++;
mFailedPatternAttemptsSinceLastTimeout++;
mCallback.reportFailedPatternAttempt();
}
if (mFailedPatternAttemptsSinceLastTimeout >=
 LockPatternUtils.FAILED_ATTEMPTS_BEFORE_TIMEOUT) {
long deadline = mLockPatternUtils.setLockoutAttemptDeadline();
handleAttemptLockout(deadline);
return;
}
mUnlockIcon.setVisibility(View.VISIBLE);
mUnlockHeader.setText(R.string.lockscreen_pattern_wrong);
mLockPatternView.postDelayed(
mCancelPatternRunnable,
PATTERN_CLEAR_TIMEOUT_MS);
}
}
}
```

UnlockScreen class

KeyguardViewCallback interface

```
package com.android.internal.policy.impl;
public interface KeyguardViewCallback {
//@ public instance model boolean  authenticationOK;

/*@ public normal_behaviour
@ requires true;
@ ensures authenticationOK==authenticated;
@ assignable authenticationOK;*/
void keyguardDone(boolean authenticated);
}
```

——————————————— KeyguardViewCallback interface ——

—— EditContactActivity class ———————————————

```
package com.android.contacts;
public final class EditContactActivity extends Activity implements
 View.OnClickListener,TextWatcher, View.OnFocusChangeListener {
/*@ public model int count;
@ invariant count<=getAllEntries().length;
@ public ghost URI insertedContactUri;
@ represents count<-ContactEntryAdapter.
@                             countEntries(mSections, false);
@*/

/*@ public model nullable EditEntry[] getAllEntries() {
@ EditEntry[] entry = new EditEntry[count];
@ for (int i=0;i<count;i++)
@       entry[i]=ContactEntryAdapter.getEntry(mSections, i, false);
@ return entry};
@*/

/*@ public normal_behavior
@ requires (\forall int i;i<count && i>=0;
@                                getAllEntries()[i]!=null &&
@                                getAllEntries()!=null   &&
@            (\exists  EditEntry entry ;
@                entry==getAllEntries()[i]&&
@                !People.CUSTOM_RINGTONE.equals(entry.column) &&
@                !People.SEND_TO_VOICEMAIL.equals(entry.column)&&
@                 entry.getData()!=null));
@ ensures mResolver.query(insertedContactUri,CONTACT_PROJECTION,
@                             null,null,null)!=null &&
@                            mResultCode==RESULT_OK;
@ assignable  mResolver.*,mPhoto.*,mUri.*,mUri,
@ mResultCode,\object_creation(ContentValues),
@ \object_creation(ByteArrayOutputStream),
@ \object_creation(String),
@ \object_creation(Intent),
@ \object_creation(ByteArrayOutputStream);
@ also
@ requires (\forall int i;i<count &&
@              i>=0;getAllEntries()[i]==null &&
@             !People.CUSTOM_RINGTONE.equals(entry.column) &&
@             !People.SEND_TO_VOICEMAIL.equals(entry.column)&& );
@ ensures  mResultCode==RESULT_CANCELED;
@ assignable mResolver.*,mPhoto.*,mResultCode,
@           \object_creation(ContentValues),
@           \object_creation(ByteArrayOutputStream),
@           \object_creation(String);
@*/

private void create() {

ContentValues values = new ContentValues();
String data;
int numValues = 0;

// Create the contact itself
final String name = mNameView.getText().toString();
if (name != null && TextUtils.isGraphic(name)) {
numValues++;
}
values.put(People.NAME, name);
values.put(People.PHONETIC_NAME, mPhoneticNameView.getText().
            toString());

// Add the contact to the My Contacts group
Uri contactUri = People.createPersonInMyContactsGroup(mResolver,
                  values);
/*@ set insertedContactUri=contactUri;*/
// Add the contact to the group that is being displayed in the contact list
SharedPreferences prefs = PreferenceManager.getDefaultSharedPreferences(this);
int displayType = prefs.getInt(ContactsListActivity.PREF_DISPLAY_TYPE,
ContactsListActivity.DISPLAY_TYPE_UNKNOWN);
if (displayType == ContactsListActivity.DISPLAY_TYPE_USER_GROUP) {
String displayGroup = prefs.getString(ContactsListActivity.PREF_DISPLAY_INFO,
null);
if (!TextUtils.isEmpty(displayGroup)) {
People.addToGroup(mResolver, ContentUris.parseId(contactUri), displayGroup);
}
} else {
// Check to see if we're not syncing everything and if
//so if My Contacts is synced.
// If it isn't then the created contact can end up not
// in any groups that are
// currently synced and end up getting removed from the phone,
// which is really bad.
boolean syncingEverything = !"0".equals(
      Contacts.Settings.getSetting(mResolver, null,
Contacts.Settings.SYNC_EVERYTHING));
if (!syncingEverything) {
boolean syncingMyContacts = false;
Cursor c = mResolver.query(Groups.CONTENT_URI, new String[] { Groups.SHOULD_SYNC },
Groups.SYSTEM_ID + "=?", new String[] { Groups.GROUP_MY_CONTACTS }, null);
if (c != null) {
try {
if (c.moveToFirst()) {
syncingMyContacts = !"0".equals(c.getString(0));
}
} finally {
c.close();
}
}

if (!syncingMyContacts) {
// Not syncing My Contacts, so find a group that is being synced and stick
// the contact in there. We sort the list so at least all contacts
// will appear in the same group.
c = mResolver.query(Groups.CONTENT_URI, new String[] { Groups._ID },
Groups.SHOULD_SYNC + "!=0", null, Groups.DEFAULT_SORT_ORDER);
if (c != null) {
try {
if (c.moveToFirst()) {
People.addToGroup(mResolver, ContentUris.parseId(contactUri),
c.getLong(0));
}
} finally {
c.close();
}
}
}
}
}
```

```
// Handle the photo
if (mPhoto != null) {
ByteArrayOutputStream stream = new ByteArrayOutputStream();
mPhoto.compress(Bitmap.CompressFormat.JPEG, 75, stream);
Contacts.People.setPhotoData(getContentResolver(), contactUri,
   stream.toByteArray());
}

// Create the contact methods
int entryCount = ContactEntryAdapter.countEntries(mSections, false);
for (int i = 0; i < entryCount; i++) {
EditEntry entry = ContactEntryAdapter.getEntry(mSections, i, false);
if (entry.kind != EditEntry.KIND_CONTACT) {
values.clear();
if (entry.toValues(values)) {
// Only create the entry if there is data
entry.uri = mResolver.insert(
Uri.withAppendedPath(contactUri, entry.contentDirectory), values);
entry.id = ContentUris.parseId(entry.uri);
if (!People.CUSTOM_RINGTONE.equals(entry.column) &&
 !People.SEND_TO_VOICEMAIL.equals(entry.column))
{
numValues++;
}
}
} else {
// Update the contact with any straggling data, like notes
data = entry.getData();
values.clear();
if (data != null && TextUtils.isGraphic(data)) {
values.put(entry.column, data);
mResolver.update(contactUri, values, null, null);
if (!People.CUSTOM_RINGTONE.equals(entry.column) &&
!People.SEND_TO_VOICEMAIL.equals(entry.column)) {
numValues++;
}
}
}
}
```

```
if (numValues == 0) {
mResolver.delete(contactUri, null, null);
setResult(RESULT_CANCELED);
} else {
mUri = contactUri;
Intent resultIntent = new Intent()
.setData(mUri)
.putExtra(Intent.EXTRA_SHORTCUT_NAME, name);
setResult(RESULT_OK, resultIntent);
Toast.makeText(this, R.string.contactCreatedToast,
  Toast.LENGTH_SHORT).show();
}
}

/*@ public normal_behavior
@ requires \forall int i;i<count && i>=0 ;
@                        (\exists EditEntry entry ;
@                               entry==getAllEntries()[i] &&
@                                entry.getData()!=null )  &&
@          !People.CUSTOM_RINGTONE.equals(entry.column) &&
@          !People.SEND_TO_VOICEMAIL.equals(entry.column);
@ ensures (\forall int i;i<count && i>=0;
@           (mResolver.query(getAllEntries()[i].uri,
@                         CONTACT_PROJECTION,null,null,null).
@                         getString(getAllEntries()[i].column))
@                          .equal(getAllEntries()[i].getData())
@                && mResultCode==RESULT_OK;
@ assignable \fields_of(mResolver),
@     \fields_of(mPhoto),mResultCode,
@     \object_creation(ContentValues),\object_creation(Intent),
@     \object_creation(ByteArrayOutputStream);
@ also
@ requires (\forall int i;i<count && i>=0 ;
@                getAllEntries()[i].getData()==null )&&
@           !People.CUSTOM_RINGTONE.equals(entry.column) &&
@           !People.SEND_TO_VOICEMAIL.equals(entry.column);
@ ensures mResultCode==RESULT_CANCELED && mUri==null;
@ assignable \fields_of(mResolver),\fields_of(mPhoto),
@        mResultCode,\fields_of(mUri),
@        \object_creation(ContentValues),
@        \object_creation(ByteArrayOutputStream);
```

```
@*/

private void save() {
ContentValues values = new ContentValues();
String data;
int numValues = 0;

// Handle the name and send to voicemail specially
final String name = mNameView.getText().toString();
if (name != null && TextUtils.isGraphic(name)) {
numValues++;
}
mResolver.acquireProvider();
values.put(People.NAME, name);
values.put(People.PHONETIC_NAME, mPhoneticNameView.getText().toString());
mResolver.update(mUri, values, null, null);

if (mPhotoChanged) {
// Only write the photo if it's changed, since we don't initially load mPhoto
if (mPhoto != null) {
ByteArrayOutputStream stream = new ByteArrayOutputStream();
mPhoto.compress(Bitmap.CompressFormat.JPEG, 75, stream);
Contacts.People.setPhotoData(mResolver, mUri, stream.toByteArray());
} else {
Contacts.People.setPhotoData(mResomUrilver, mUri, null);
}
}

int entryCount = ContactEntryAdapter.countEntries(mSections, false);
for (int i = 0; i < entryCount; i++) {
EditEntry entry = ContactEntryAdapter.getEntry(mSections, i, false);
int kind = entry.kind;
data = entry.getData();
boolean empty = data == null || !TextUtils.isGraphic(data);
if (kind == EditEntry.KIND_CONTACT) {
values.clear();
if (!empty) {
values.put(entry.column, data);
mResolver.update(entry.uri, values, null, null);
if (!People.CUSTOM_RINGTONE.equals(entry.column) &&
!People.SEND_TO_VOICEMAIL.equals(entry.column)) {
```

```
numValues++;
}
} else {
values.put(entry.column, (String) null);
mResolver.update(entry.uri, values, null, null);
}
} else {
if (!empty) {
values.clear();
entry.toValues(values);
if (entry.id != 0) {
mResolver.update(entry.uri, values, null, null);
} else {
mResolver.insert(entry.uri, values);
}
if (!People.CUSTOM_RINGTONE.equals(entry.column) &&
!People.SEND_TO_VOICEMAIL.equals(entry.column)) {
numValues++;
}
} else if (entry.id != 0) {
mResolver.delete(entry.uri, null, null);
}
}
}

if (numValues == 0) {
// The contact is completely empty, delete it
mResolver.delete(mUri, null, null);
mUri = null;
setResult(RESULT_CANCELED);
} else {
// Add the entry to the my contacts group if it isn't there already
People.addToMyContactsGroup(mResolver, ContentUris.parseId(mUri));
setResult(RESULT_OK, new Intent().setData(mUri));

// Only notify user if we actually changed contact
if (mContactChanged || mPhotoChanged) {
Toast.makeText(this, R.string.contactSavedToast,
    Toast.LENGTH_SHORT).show();
}
}
```

```
}
```

——————————————————————— EditContactActivity class ——

—— SimUnlockScreen class ———————————————————————

```
package com.android.internal.policy.impl;
public class SimUnlockScreen extends LinearLayout implements
      KeyguardScreen, View.OnClickListener,
KeyguardUpdateMonitor.ConfigurationChangeCallback {

//@ public ghost boolean checked;
private final int[] mEnteredPin = {0, 0, 0, 0, 0, 0, 0, 0};
//@ invariant (\forall int i;i>=0 && i< mEnteredPin.legth ;mEnteredPin[i]=0);
/*@ public normal_behaviour
@ requires mEnteredDigits>4 &&
@(ITelephony.Stub.asInterface(ServiceManager.checkService("phone")).
@     supplyPin(mPinText.getText().toString()));
@ ensures checked;
@ assignable mCallback.*,checked,\object_creation(CheckSimPin),
@ \object_creation(Progressdialogue),
@ mUpdateMonitor.mSimState,mSimUnlockProgressdialogue.*,
@ \object_creation(String);
@ also
@ requires mEnteredDigits<4 || @!
@   (ITelephony.Stub.asInterface(ServiceManager.checkService("phone")).
@   supplyPin(mPinText.getText().toString()));
@ ensures !checked,mEnteredDigits==0;
@ assignable @mHeaderText.mCharWrapper.mChars,
@   mPinText.mCharWrapper.mChars,
@ mEnteredDigits,mCallback.*,checked,\object_creation(String);
*/
private void checkPin() {
//@ set checked=false;
// make sure that the pin is at least 4 digits long.
if (mEnteredDigits < 4) {
// otherwise, display a message to the user, and don't submit.
mHeaderText.setText(R.string.invalidPin);
mPinText.setText("");
mEnteredDigits = 0;
mCallback.pokeWakelock();
return;
}
getSimUnlockProgressdialogue().show();
new CheckSimPin(mPinText.getText().toString()) {
void onSimLockChangedResponse(boolean success) {
if (mSimUnlockProgressdialogue != null) {
mSimUnlockProgressdialogue.hide();
}
if (success) {
// before closing the keyguard, report back that
// the sim is unlocked so it knows right away
//@ set checked=true;
mUpdateMonitor.reportSimPinUnlocked();
mCallback.goToUnlockScreen();
} else {
//*@ set checked=false;
mHeaderText.setText(R.string.keyguard_password_wrong_pin_code);
mPinText.setText("");
mEnteredDigits = 0;
mCallback.pokeWakelock();
}
}
}.start();
}
```

SimUnlockScreen class

AccountUnlockScreen class

```
package com.android.internal.policy.impl;
public class AccountUnlockScreen extends RelativeLayout implements
       KeyguardScreen,
View.OnClickListener, ServiceConnection, TextWatcher {

/*@ public model nullable String account;
@ represents account<-findIntendedAccount(mLogin.getText().toString());
@ public ghost boolean keyguarddone;
*/
/*@ public normal_behavior
@ requires  v == mOk && checkPassword() ;
@ ensures keyguarddone ;
@ assignable \object_creation(FileOutputStream),
@  \object_creation(PrintWriter),
@ keyguarddone,
@ \object_creation(android.accounts.IAccountsService.Stub.Proxy),
@\object_creation(String),\object_creation(Intent),
@\object_creation(RandomAccessFile);
@ also
@ requires  v == mOk && !checkPassword()
@ ensures !keyguarddone ;
@ assignable @\object_creation(FileOutputStream),
@ \object_creation(PrintWriter),
@ mInstructions.mCharWrapper.mChars,mPassword.mCharWrapper.mChars
@ \object_creation(android.accounts.IAccountsService.Stub.Proxy),
@\object_creation(String),\object_creation(Intent),
@\object_creation(RandomAccessFile),keyguarddone;

*/
public void onClick(View v) {
// set keyguarddone=false;
mCallback.pokeWakelock();
if (v == mOk) {
if (checkPassword()) {
// clear out forgotten password
mLockPatternUtils.setPermanentlyLocked(false);

// launch the 'choose lock pattern' activity so
// the user can pick a new one if they want to
Intent intent = new Intent();
intent.setClassName(LOCK_PATTERN_PACKAGE, LOCK_PATTERN_CLASS);
intent.setFlags(Intent.FLAG_ACTIVITY_NEW_TASK);
mContext.startActivity(intent);

// close the keyguard
// set keyguarddone=true;
mCallback.keyguardDone(true);
} else {
mInstructions.setText(R.string.lockscreen_glogin_invalid_input);
mPassword.setText("");
// set keyguarddone=true;
}
}

if (v == mEmergencyCall) {
mCallback.takeEmergencyCallAction();
}
}
```

```
@Override
public boolean dispatchKeyEvent(KeyEvent event) {
if (event.getAction() == KeyEvent.ACTION_DOWN
&& event.getKeyCode() == KeyEvent.KEYCODE_BACK) {
mCallback.goToLockScreen();
return true;
}
return super.dispatchKeyEvent(event);
}
/*@ public normal_behaviour
@ requires  account!=null
@ ensures \result==mAccountsService.shouldUnlock(account,
@           mPassword.getText().toString());
@ assignable \object_creation(FileOutputStream),
@ \object_creation(PrintWriter),
@ \object_creation(android.accounts.IAccountsService.Stub.Proxy),
@ \object_creation(String);
@ also
@ public normal_behaviour
@ requires  account==null ;
@ ensures \result==false;
@ assignable @\object_creation(FileOutputStream),
@ \object_creation(PrintWriter),
@\object_creation(android.accounts.IAccountsService.Stub.Proxy),
@\object_creation(String);
@ also
@ public exceptional_behaviour
@ requires  \nothing ;
@ signals (Exception e) e instanceof RemoteException && \result==false;
@ assignable \object_creation(RemoteException),
@\object_creation(FileOutputStream),
@\object_creation(PrintWriter),
@ \object_creation(android.accounts.IAccountsService.Stub.Proxy),
@\object_creation(String);
*/

private /*@pure*/ boolean checkPassword() {
final String login = mLogin.getText().toString();
final String password = mPassword.getText().toString();
try {
```

```
String account = findIntendedAccount(login);
if (account == null) {
return false;
}
return mAccountsService.shouldUnlock(account, password);
} catch (RemoteException e) {
return false;
}
}
```

——————————————————— AccountUnlockScreen class ——

—— Activity class ———————————————————

```
public class Activity extends ContextThemeWrapper
implements LayoutInflater.Factory,
Window.Callback, KeyEvent.Callback,
OnCreateContextMenuListener, ComponentCallbacks {
/*@ public  model non_null  boolean[] state={mResumed,
                            mStopped,mFinished,mStartedActivity};
@ invariant (\exists int i; i>=0 && i<state.length; state[i]==true  &&
@   (\forall int j ;j>=0 && j<state.length;j!=i && state[j]==false ));
*/

/*@ public normal_behaviour
@ requires intent.getAction() ==Intent.ACTION_CALL_EMERGENCY &&
@    intent.getFlags()==Intent.FLAG_ACTIVITY_NEW_TASK
@      && requestCode==-1;
@ ensures Integer.valueOf(activityResult)>=
@      IActivityManager.START_SUCCESS;
@ assignable intent.call_dispatched ,
@     mInstrumentation.mActivityMonitors.mHints,
@ activityResult;
@
@ also
@
@ public exceptional_behaviour
@ requires (intent.getAction() == Intent.ACTION_CALL_EMEnRGENCY &&
@ intent.getFlags()==Intent.FLAG_ACTIVITY_NEW_TASK && requestCode==-1);
@ ensures Integer.valueOf(activityResult)==
@      IActivityManager.START_CLASS_NOT_FOUND;
@ signal(Exception e)e instanceof android.content.ActivityNotFoundException;
@ assignable activityResult;
*/
```

```
//*@ String ghost activityResult;
public void startActivityForResult(Intent intent, int requestCode) {
if (mParent == null) {
Instrumentation.ActivityResult ar =
mInstrumentation.execStartActivity(
this, mMainThread.getApplicationThread(), mToken, this,
intent, requestCode/*@, activityResult*/);

if (ar != null) {
mMainThread.sendActivityResult(
mToken, mEmbeddedID, requestCode, ar.getResultCode(),
ar.getResultData());
}
if (requestCode >= 0) {
// If this start is requesting a result, we can avoid making
// the activity visible until the result is received.  Setting
// this code during onCreate(Bundle savedInstanceState) or onResume() will keep the
// activity hidden during this time, to avoid flickering.
// This can only be done when a result is requested because
// that guarantees we will get information back when the
// activity is finished, no matter what happens to it.
mStartedActivity = true;
}
} else {
mParent.startActivityFromChild(this, intent, requestCode);
}
}
/*@ public normal_behaviour
@ requires intent.getAction() ==Intent.ACTION_CALL_EMERGENCY &&
@ intent.getFlags()==Intent.FLAG_ACTIVITY_NEW_TASK ;
@ ensures Integer.valueOf(activityResult)>=
@   IActivityManager.START_SUCCESS;
@ assignable intent.call_dispatched ,
@    mInstrumentation.mActivityMonitors.mHints,
@ activityResult;
@
@ also
@
@ public exceptional_behaviour
@ requires (intent.getAction() == Intent.ACTION_CALL_EMEnRGENCY &&
@ intent.getFlags()==Intent.FLAG_ACTIVITY_NEW_TASK );
@ ensures Integer.valueOf(activityResult)==
@    IActivityManager.START_CLASS_NOT_FOUND;
@ signal(Exception e)e instanceof android.content.ActivityNotFoundException;
@ assignable activityResult;
*/
public void startActivity(Intent intent) {
startActivityForResult(intent, -1);
}
/*@ public normal_behavior
@ requires resultCode==(RESULT_CANCELED || RESULT_OK
@ ||RESULT_FIRST_USER);
@ ensures mResultCode==resultCode && mResultData==null;
@ assignable mResultCode,mResultData;

*/
public final void setResult(int resultCode) {
synchronized (this) {
mResultCode = resultCode;
mResultData = null;
}
}
/*@ public normal_behavior
@ requires resultCode==(RESULT_CANCELED || RESULT_OK
@ ||RESULT_FIRST_USER);
@ ensures mResultCode==resultCode && mResultData==data;
@ assignable mResultCode,mResultData;

*/
public final void setResult(int resultCode, /*@ nullable*/Intent data) {
synchronized (this) {
mResultCode = resultCode;
mResultData = data;
}
}
/*@ public normal_behaviour
@ require mParent==null;
@ ensures mFinished;
@ assignable mFinished;
@ also
@ require mParent!=null;
@ ensures mParent.mFinished;
@ assignable mParent.mFinished;

*/
public void finish() {
if (mParent == null) {
int resultCode;
Intent /*@ nullable*/ resultData;
synchronized (this) {
resultCode = mResultCode;
resultData = mResultData;
}
if (Config.LOGV) Log.v(TAG, "Finishing␣self:␣token=" + mToken);
try {
if (ActivityManagerNative.getDefault()
.finishActivity(mToken, resultCode, resultData)) {
mFinished = true;
}
} catch (RemoteException e) {
// Empty
}
} else {
mParent.finishFromChild(this);
}
}
```

Activity class

Instrumentation class

```
package android.app;
public class Instrumentation {
public ActivityResult execStartActivity(
Context who, IBinder contextThread, IBinder token, Activity target,
Intent intent, int requestCode /*@, String ghost activityResult*/) {
IApplicationThread whoThread = (IApplicationThread) contextThread;
if (mActivityMonitors != null) {
synchronized (mSync) {
final int N = mActivityMonitors.size();
for (int i=0; i<N; i++) {
final ActivityMonitor am = mActivityMonitors.get(i);
if (am.match(who, null, intent)) {
am.mHits++;
if (am.isBlocking()) {
```

```
return requestCode >= 0 ? am.getResult() : null;
}
break;
}
}
}
}
try {
int result = ActivityManagerNative.getDefault()
.startActivity(whoThread, intent,
intent.resolveTypeIfNeeded(who.getContentResolver()),
null, 0, token, target != null ? target.mEmbeddedID : null,
requestCode, false, false);


//@set activityResult=Integer.toString(result);
checkStartActivityResult(result, intent);
} catch (RemoteException e) {
}
return null;
}
```

Instrumentation class

Intent class

```
package android.content;
public class Intent implements Parcelable {
//@ public ghost boolean callDispatched;
}
```

Intent class