NTNU
Norwegian University of
Science and Technology

Developing Android Applications with Arctis

Stephan Haugsrud

Master of Science in Communication Technology
Submission date: June 2009
Supervisor: Rolv Bræk, ITEM
Co-supervisor: Frank Alexander Kraemer, ITEM

Norwegian University of Science and Technology
Department of Telematics

Problem Description

Developing Android Applications with Arctis

Android applications for mobile phones typically consist of code and resources for user interfaces, interfaces to network resources and hardware features of the phone and code to interact with other services. Despite that a single Android application is usually not distributed, this means that there can be a high number of events that
occur at each interface. The user interface, for instance, produces events upon every action by the user, location services may update a status at any time, or applications may be notified by other services
available on the phone. An application developer faces the complex task of coordinating the behaviors that all these interfaces have.

Coordinating concurrent behavior is one of the strengths of the Arctis tool, in which applications are specified by means of UML activity diagrams. These activities are constructed by the composition of building blocks that encapsulate certain functionalities. Executable
code is generated automatically via a transformation into state machines.

The task in this thesis is to examine how Arctis can be used for the development of Android applications. The existing code generator for Java should be extended to produce Android applications, and mechanisms should be investigated and proposed to encapsulate Android services provided by the API of the phone as building blocks in Arctis. To demonstrate the effectiveness of the proposed solutions, a simple application should be developed.

Assignment given: 15. January 2009
Supervisor: Rolv Bræk, ITEM

Abstract

The focus of this thesis is the design of Android applications from building blocks in Arctis. The Arctis tool is used for modeling applications with UML activities, which already can be deployed on the Java ME and Java SE platforms. State machines and a runtime support system are generated. Creation of a generator for Arctis, enabling deployment to the Android platform, is the key element in this work. The Android platform is presented using an example application. A discussion on challenges, solutions and architectures for the design and implementation of Android applications using Arctis is presented. Then, the example application is redesigned using Arctis building blocks and deployed as a Java project using the existing code generator for Java SE. The adaptations necessary for turning the Java project into a runnable Android project is studied in detail and serves as a basis for the development of our code generator, along with the discussion. After describing our code generator, several building blocks are designed for an Android building block library. Demonstration of the code generator and the building blocks are done by designing and deploying an Android application named *TwitterFromAndroid*.

Preface

This thesis is a result of my work in the course TTM4900, from the Department of Telematics (ITEM) at the Norwegian University of Science and Technology (NTNU). The academically responsible for my subject has been Professor Rolv Bræk and my supervisor has been Frank Alexander Kraemer.

I would like to thank Frank Alexander Kraemer for valuable and continuous feedback and guidance on my work. I would also like to acknowledge Marius Bjerke for his input and Sverre Bye Grimsmo for help with LaTeX.

Trondheim June 10, 2009

Stephan Haugsrud

Contents

List of Figures

Acronyms

OS Operating System

API Application Programming Interface

SDK Software Development Kit

VM Virtual Machine

SMS Short Message Service

UI User Interface

UML Uniform Markup Language

GPS Global Positioning System

XML Extension Markup Language

jar Java Archive

JET Java Emitter Templates

JSP JavaServer Pages

JRE Java Runtime Environment

EMF Eclipse Modeling Framework

SQL Structured Query Language

NTNU Norges teknisk-naturvitenskapelige universitet

J2SDK Java 2.0 Software Development Kit

SE Standard Edition

PDE Eclipse Plug-in Development Environment

JDT Java Development Tool

GUI Graphical User Interface

Chapter 1

Introduction

Android is an open-source Operating System (OS) for mobile devices. Applications can be developed using the Android Software Development Kit (SDK) and run on Android devices without central approvement. The SDK provides access to several Android system services which includes a location service, alarm service and a notification service. A typical Android application may consist of quite an amount of User Interface (UI) code which has to be programmed manually. On the other hand, core functionality like the use of Android services can be standardized.

The use of Android services means that the applications have to be reactive. Applications have to respond to user inputs, e.g starting a service, or system input such as location updates, incoming calls or alarms going off.

Development of reactive applications might provide a challenge due to a possible high degree of concurrency. Arctis, an Eclipse based tool used for designing systems and services, uses state machines and runtime support as a means to handle concurrency. The state machines are generated automatically from building blocks.

Use of Arctis building blocks in developing Java SE applications have been shown in Arctis system designs for the *Uno* card game [16] and Treasure Hunt [15]. The building block approach could be useful also for Android applications.

The goal of this work is to use Arctis building blocks and a code generator when developing Android applications, as can be seen in figure 1.1. An Arctis library containing building blocks that access Android services or perform other tasks are used in developing system designs for Android applications. From that, a code generator will be used to generate Android executable code for the designed application.

Arctis currently has a deployment option for Java SE, enabling implementation of Arctis system designs into executable Java code. The executable Java code is capable of running on an Android device, but as the generated project is not yet an Android project, adaptations are needed. These will be documented. To avoid having to make manual adaptations every time an Android application

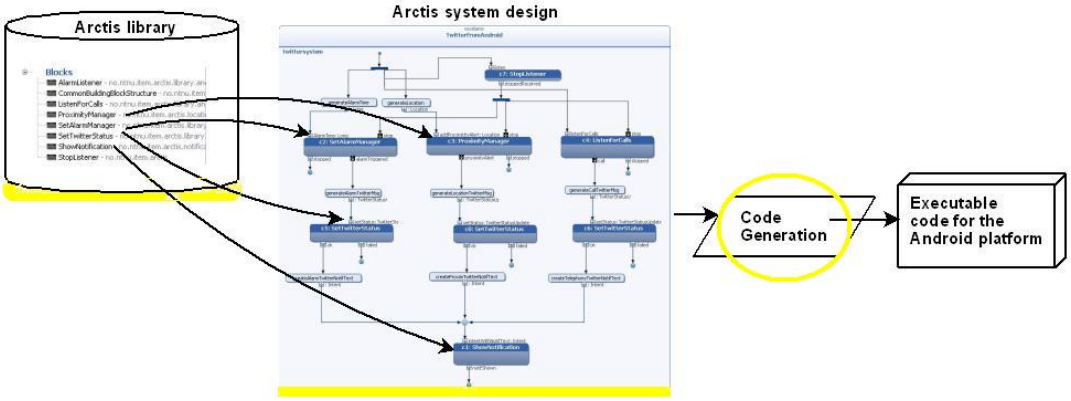

Figure 1.1: Application development with Arctis for Android

is to be generated, we will use the documented adaptations in creation of a code generator, *Java SE for Android*. Upon developing the generator, a discussion on the solution architecture, including possible challenges and problems, will be done. The resulting generator is used in development of example applications used for demonstration in this work.

Designing building blocks for every single service provided by the Android platform is out of the scope of this thesis. Hence, it is important to develop a solution which focus on a general structure. This will allow developers to design new building blocks and implement them using the Android code generator.

Graphical user interfaces are not of focus, as all processes are supposed to be run in the background and the goal is that the designed applications should react to input from either the Android device's services or input through remote calls from a connection.

1.1 Contribution

The highlighted parts of figure 1.1 represents our contribution. The Arctis editor will be equipped with an Android building block library which contain all blocks developed for use by Android applications. Two example applications are designed using Uniform Markup Language (UML) activities, existing building blocks and building blocks developed for use by Android. State machines and model checking are done as for every application designed in Arctis. A new code generator is developed, using parts from the currently existing *Java SE* generator, and adding functionality for generation of Android projects. A new deployment option for Arctis is added, *Java SE for Android*, which uses our Android generator in creation of an Android project. This project will be able to run on the Android emulator or mobile device.

1.2 Outline

This work is outlined as follows:

Chapter 2 provides an example driven description of the Android platform.

Chapter 3 presents the tools used in this work.

Chapter 4 discuss the goals of Android application design in Arctis. The architecture, problems and challenges of the solution is also described.

Chapter 5 illustrates what changes are needed when adapting a Java SE generated project into an Android project. This will be used as a ground stone for our Java SE for Android generator.

Chapter 6 describes the Android code generator in detail.

Chapter 7 provides a look at all Arctis building blocks for Android applications developed in this work.

Chapter 8 presents an example Android application built using the developed building blocks.

Chapter 9 concludes the work and suggest future work.

Chapter 2

Android

2.1 Introduction

Android is a complete, open and free mobile platform initiated by the *Open Handset Alliance* [27]. Android comes with a set of core functionalities which can be accessed through standard Application Programming Interface (API)'s. All applications on the mobile device are created equal and can be replaced or extended. Applications running on Android can be run in parallel. The below subsections describe the basics of the Android system, based on [8].

2.1.1 Platform

The Android platform includes an operating system, middle ware and key applications. Any developer can construct new applications for the platform using the Android SDK. All applications are written in Java and is run on the Dalvik Virtual Machine (VM) [31, 9].

2.1.2 Architechture

Figure 2.1 depicts the Android architecture. A bottom-up explanation of the architecture follows.

Linux Kernel lies on the bottom of the Android Architecture and acts as an abstraction layer between the hardware and the rest of the stack. The kernel provides drivers for the different parts of the mobile phone, as well as handling the power management. Also, security, memory management, process management and network stack are handled by the Linux kernel.

Libraries provides a set of C/C++ libraries which are used by various components of the Android system. The Media Framework libraries are used to store and playback video as well as images. *SQLite* is a lightweight relationship database available for all applications. SGL are the underlying 2D graphics

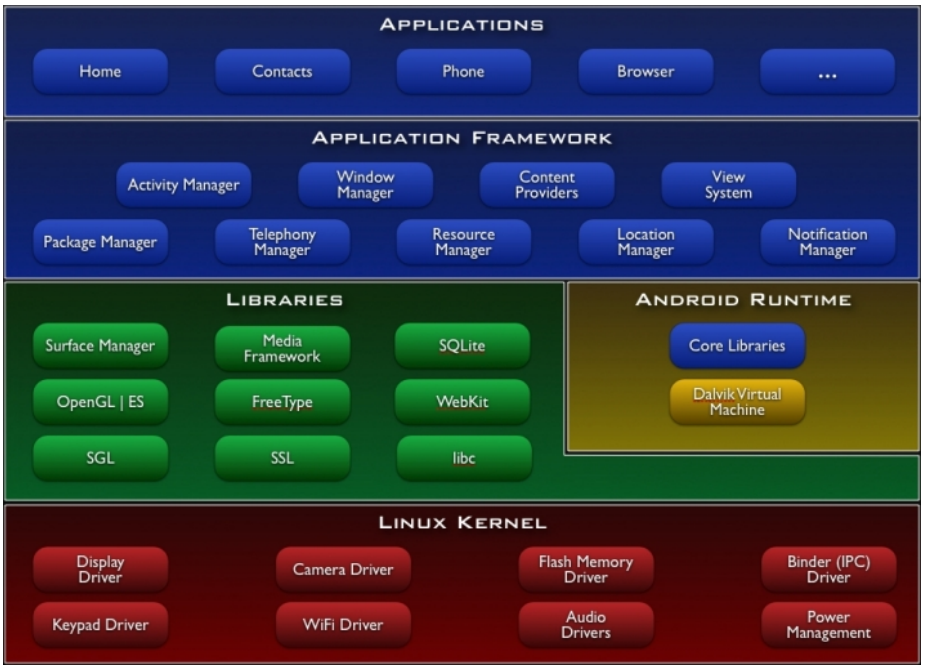

Figure 2.1: Android Architecture, from [8]

engine, while the 3D libraries are based on an *OpenGL ES* implementation. *LibWebCore* is a web browser engine.

All of these libraries and their capabilities are available for the developer through the Android application framework.

Android Runtime includes core libraries and the Dalvik VM, [31]. The core libraries provide most of the basic functionality available in the core libraries of the Java programming language. The Dalvik VM is written as to reduce memory footprint and enable a device to run multiple VM's effectively.

The Application Framework architecture allows for reuse of components. Any application can make use of the capabilities of a component and also publish its own capabilities. Every application has underlying components, including:

- **Views** consisting of i.e buttons, lists, text boxes and a web browser, all used to build an application.

- **An Activity Manager** that controls navigation and manages the life cycle of an application.

- **A Notification Manager** that enables all applications to have notifications displayed as alerts in the status bar.

- **A Resource Manager** providing access to non-code resources such as localized strings, graphics, and layout files.

- **Content Providers** that enables applications to share their own, and access data from other applications.
- **Applications**, written in Java and provided as default within Android. These core applications includes Short Message Service (SMS) program, calendar, browser, maps, email client and other.

2.2 The *HelloLocationWorld* Application

HelloLocationWorld is an application where the user will be notified when closing in on a selected location. To achieve this, the default Android location service will be used, along with customized screens, alerts and notifications. Below follows an illustration, see figure 2.2, and a description on how the application works.

On start, the application will provide a welcome screen to the user, displaying four buttons. The first three buttons have the name of a location on the Norges teknisk-naturvitenskapelige universitet (NTNU) campus, while the last one stops the service. After the user has selected a preferred location, an alert will pop up, displaying information on the service started. The welcome screen with the alert is depicted in *Step 1* of figure 2.2. The "'Stop Service"' button will, correspondingly, stop the started service and display an alert with this information.

An Android service is now running in the background, constantly checking for location updates, regardless of what other applications the user is currently running on the Android device. Figure 2.2 *Step 2* shows the Android device's desktop background, as a result of the *Home* button being pressed.

As the user moves within the set limit of the selected location (movement is simulated on the emulator console [7]) a notification is displayed at the top of the screen, as seen in figure 2.2 *Step 3*. The user may then extend the notification view to get more information on the notification, *Step 4*.

The following sections will describe the different elements of an Android application and how they work. References to elements used in the *HelloLocationWorld* application is used whenever possible.

2.3 Components in an Android Application

An android application may consist of four kinds of elements:

- **Activity, described in section 2.4**
- **Broadcast Intent Receiver, described in section 2.5**
- **Service, described in section 2.6**
- **Content Provider, used when applications want to share their data. Not relevant for this work.**

Figure 2.2: The *HelloLocationWorld* application

All Android applications are written using some combination of the listed elements. The elements composing an application have to be declared in a manifest file, *AndroidManifest.xml*, which is described in section 2.9.

2.4 Activity

This element is common in Android applications. An activity is a single class extending the base class *Activity*. Almost all activities interacts with the user. Because of this, the base class comes with a method *setContentView(View)* in which you can place your own UI. It is important to note that one activity usually is related to a single screen in an application, meaning every new UI is related to its own class. In our example application we operate with a single UI, depicted in figure 2.2 *Step 1*, but many applications consists of multiple screens. In a multi screen application the different activity classes are called due to user or system input.

Android allows only one activity running at a time and the activity base class has several life cycle methods. Regardless of running a single screen or multi screen application, each activity class will need to consider the activity life cycle. At all times an activity could be put on hold due to i.e an incoming phone call or when a new activity is started. A description of the Android activity life cycle follows belows.

2.4.1 Activity Lifecycle

The Android system uses an activity stack to manage activities. A new activity is put on top of the stack and comes to the foreground, while the previous ones remain below and will not come to the foreground before the newly created one exits.

Figure 2.3 depicts the life cycle of an Android activity. *onCreate()* is always called whenever an activity is started for the first time, and this method is mandatory for every activity.

The **entire lifetime** of an activity yields from *onCreate()* is called until *onDestroy()* is called. Usually, as in the *HelloLocationWorld* application, the UI is set in *onCrate()* along with the global state of the activity. *onDestroy()*is called when the activity should be shut down, as when we push the *Stop Service* button in the activity of the example application.

In between the entire lifetime we have the **visible lifetime** of the activity. This is the time between *onStart()* and *onStop()* is called. During this time the activity is visible to the user, although it might not be a foreground activity. This means the activity might not interact with the user at all times, but may maintain resources that are needed to show the activity to the user. An example resource is a *Broadcast Receiver* monitoring changes which impact the UI. The receiver, see 2.5, is registered in *onStart()* and closed in *onStop()*. This cycle

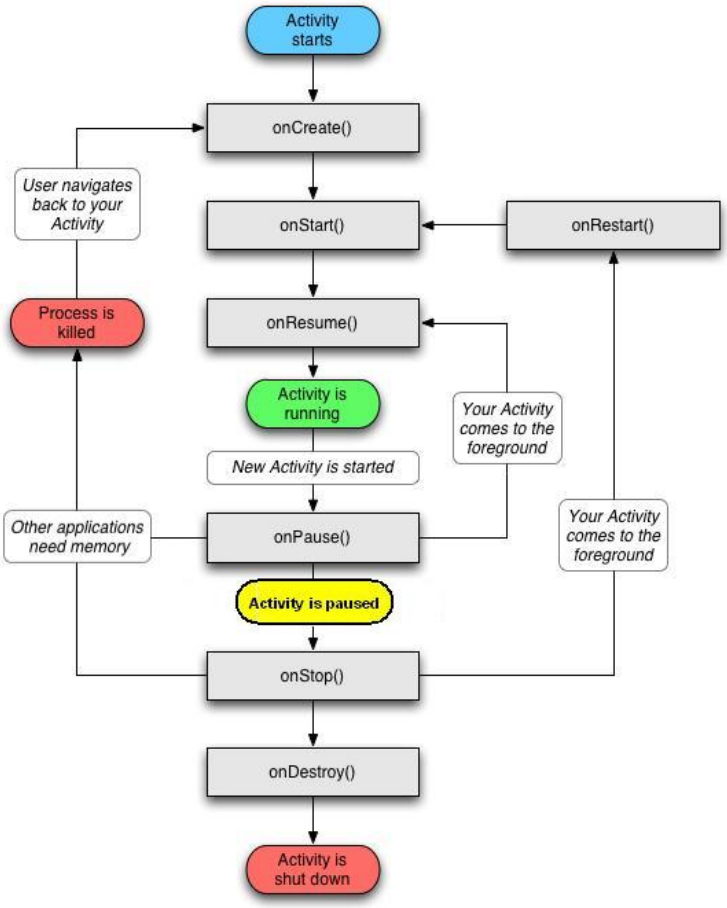

Figure 2.3: The activity life cycle, from [5]

could repeat itself several times during the entire lifetime of the activity as seen in figure 2.3.

Lastly there is the **foreground lifetime** which represents the time the activity is interacting with the user and is in front of all other activities. The foreground lifetime yields from *onResume()* is called until *onPause()* is called. *onPause()* is frequently called as applications start new activities, services and broadcasts. Additionally, onPause() is called when the mobile device goes to sleep or receives an incoming call. *onPause()* is typically used for handling of unsaved data.

The Android activity has essentially four states, which are:

- *running*, where the activity is active and runs in the in the foreground.
- *paused*, where the activity has lost focus, but is still alive and visible.
- *stopped*, meaning the activity maintains state information, but is obscured by another activity and might be killed if the system needs extra memory.
- *terminated*, where the activity is either killed or shut down by the system. The activity is dropped from memory and will need to be completely recovered in a new startup.

2.4.2 Activity Life Cycle of *HelloLocationWorld*

This subsection provides a simulation of mobile device inputs and an explanation on how *HelloLocationWorld's* activity reacts to the given inputs. The simulation is done as a means to give the reader a better understanding of the life cycle of an Android activity, depicted in figure 2.3.

- Application is started.

  Methods called: *onCreate()* -> *onStart()* -> *onResume()*.

  This is the default path an activity follows from startup until it reaches the running state.

- Button is clicked, indicating which location we want to track.

  Methods called: None.

  As described in 2.2, there is no new activity started after a location button click, and hence the current activity is neither paused, stopped or destroyed. If a new activity was to be started after the button click, the current activity would have been paused.

- An incoming call is received.

  Methods called: *onPause()* -> *onStop()* -> (after hangup) *onRestart()* -> *onStart()* -> *onResume()*.

  When a call arrives the activity immediately exits the foreground, but both *onPause()* and *onStop()* is first called. During these methods all

state necessary for correctly resuming the pre-call state need to be saved. In *HelloLocationWorld* no state information is required, but in a similar case where the user would need to type in the destination, the case would have been different. The *onPause()* or *onStop()* method should then save the currently written text. As an incoming call may arrive at all times, this is of importance for every Android activity in all Android applications. *onRestart()*, *onStart()* or *onResume()* should accordingly check for previously saved state before returning the activity to the foreground.

- The *home* button is pushed, returning the mobile device to the main desktop screen, see figure 2.2 *Step 2*.

  Methods called: *onPause() -> onStop()*.

  Pushing the *home* button means we exit our current screen and application. When removing an activity from the foreground, *onPause()* is called. Making the activity non-visible triggers the *onStop()* method.

- A location update is done and the mobile device is within the correct area of our preferred location. A notification is received and opened, returning focus to the activity.

  Methods called: *onStart() -> onResume()*.

  Our activity has been non-visible for an amount of time, but it has never been destroyed. As focus is shifted back to our activity *onStart()* is called, jumping back in to the visible lifetime. As we also get our activity to the foreground *onResume()* is called.

- *Stop Service* is pushed, ending the activity and the application, before returning to the main screen.

  Methods called: *onPause() -> onResume() -> onDestroy()*.

  We finish our activity. The default finish operations are called in the given sequence. After onDestroy() has been called the entire lifetime of our activity ends.

2.5 Broadcast Receiver

Most Android applications are started by the user, but some applications should be started as a result of an external event. In these cases, a *broadcast receiver* is used. An application does not have to be running for its *broadcast receiver* to be called. When the receiver is triggered, the system will start the application.

In addition to starting an application, the *broadcast receiver* can e.g send notifications or have text appearing on screen to alert the user that something has happened, but they do not display an UI.

The lifetime of the *broadcast receiver* is the duration of the *onReceive(Context, Intent)* call. During this time the process executing the *Broadcast receiver* is considered to be a foreground process. When *onReceive* finishes, the system regards the object as no longer being active. As a consequence of this, one cannot

use a receiver with asynchronous operations, such as showing a dialog or start an Android service.

An example use of a receiver is alerting the user with an on screen message when the clock turns midnight. To achieve this, an application with a registered *broadcast receiver* in its *Manifest.xml* file, see 2.9, is created. On startup, the application uses the default Android service *Alarm Service* to set an alarm which is to be triggered at midnight. The alarm service is told to respond to the receiver when triggered. The code for the receiver class is given below. *Toast* is a Android feature used for showing on screen messages. When the clock turns midnight a call to the receiver's *onReceive* method is done, displaying the desired message.

```
public class AlarmAlerter extends BroadcastReceiver {
public void onReceive(Context context, Intent intent) {
Toast.makeText(context, "It is midnight!", Toast.LENGTH_LONG).show();
}}
```

As seen above, a *broadcast receiver* class could be written quite simple. Instead of the *Toast* message one could use a notification manager to display a notification, see section 2.10. Other triggers for a receiver could be when the phone rings, when a new data network is available or a location update is done.

2.6 Service

As described in 2.4, only one Android activity can be run at the same time. Applications might want classes which can run code continually, i.e listening for incoming location updates, incoming SMS or phone calls. To enable this, Android provides the *Service* extension. A service is code which runs without a UI. There are two categories of services used in Android applications, namely default Android services and user defined services. The default services are predefined in the Android API and includes among others location-, alarm-, telephone-, wifi-, and notification services. A complete list can be found at [3]. Basically, the default services provide access to all native functions of the mobile device. As they are part of the API, they cannot be altered and the source code is not available. User defined services, on the other hand, is written by the developers. These services can be composed by using a mix of the default services and written code.

A typical use of a default service is exemplified in *HelloLocationWorld*, see section 2.2, where we want to receive an alert after reaching a targeted location. To achieve this, the default service (LOCATION_SERVICE) is used to poll the location provider for location updates and check whether or not we are within the correct distance of our target location. The service will run even though we exit the application's activities and perform other tasks on the mobile device. When either reaching the correct location or manually canceling the service, will the service stop.

2.6.1 Service Life Cycle

The Android service life cycle, depicted in figure 2.4, have four essential life cycle methods.

- **onCreate()**, called by the system when the service is created or called upon for the first time by *startService()*.
- **onDestroy()**, called by the system to notify a service that it is being removed.
- **onStart()**, called each time a service is called upon with *startService()*.
- **stopSelf()**, stops the service, if it was previously started.

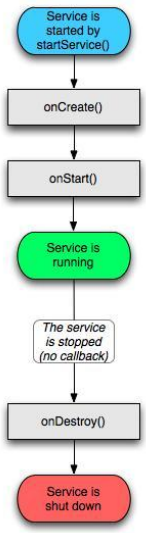

Figure 2.4: The default service life cycle, from [2]

A service can be run by the system for two reasons. If someone call *Context.startService()*, in which the system calls *onCreate()* followed by *onStart()*. The service will be running until someone call *Context.stopService()* or *stopSelf()*. If the service is running, *Context.startService()* calls result in multiple calls to the services *onStart()* method. Calls to *stopSelf() or Context.stopService()* will result in the service being stopped, no matter how many times the service has been started.

Another possibility for the system to be running a service is when clients obtain a connection to the service using *Context.bindService()*. This binding will return an *iBinder* object allowing the client to make call backs to the service. On *Context.bindService(), onCreate()* is called but no call is made to *onStart()*, as the *iBinder* object is used instead. The service calls *onDestroy ()* when there are no active connections left. This way of connecting to a service will not be discussed in further detail as it is not relevant in this work.

2.7 Intent and Intent Filters

When focus is to be shifted between activities, or services react as a result of inputs or events, the Android system needs to know what to do next. The Android *Intent* resolves this, as it names the action being requested. An intent contain information describing what an application wants to have done. This is how Android moves from activity to activity, register a broadcast receiver or starts a service. The most common intent constructor used in this work is *Intent(Context context, Class class)*. The *context* describes which Android context is used upon creation of the intent. The *class* parameter names which class is to be started upon fulfillment of the intent.

In the general example of starting a new activity, let us say we are currently running the activity class named, *FirstActivity.java*. Focus should be shifted to the class *SecondActivity.java*. To achieve this, an intent is created with the context from the *FirstActivity.java* and the class *SecondActivity.java*. When starting a new activity with this newly created intent, the system will try to run, and shift focus to the *SecondActivity.java* class. The example code is given below.

```
Intent intent = new Intent(this, SecondActivity.class);
startActivity(intent);
```

There are several other attributes used with intents, where the most important ones are *category* and *extras*. *Category* is used for setting which application component will be launched on application startup. *Extras* is a *Bundle* of additional information, used to provide extended information to the component. For example, if we are going to perform an action of sending an email, the title, body and address could be added as extras to the intent.

2.8 Permissions

The security architecture of an Android application uses permissions which are described in the manifest file, see [6]. Every permission declared in the manifest will have to be approved by the user of the Android device upon installation. This is done among other reasons as a precaution for installing malicious software. The user will at all times have control over what applications are able to access on your phone.

E.g if a map -and localization application wants permission to make calls, use the camera or send SMS, one might suspect the application to access areas it should not and one can then reject the installation.

2.9 Manifest

An Android application consists of several components, intent filters and other specifications. For the Android system to run properly, it needs to obtain a detailed description of the application and how it is composed before being able to run it. The *AndroidManifest.xml*, a necessary file in all Android applications, contain this description and displays important information to the Android system. The manifest has many responsibilities, with important ones such as:

- Describing all components included in the application. All Activities, Broadcast Receivers, Services and Content Providers used in the application.
- Declaring the permissions the application must have in order to interact with other applications and access protected parts of the API. One can also declare permissions necessary to interact with the components in the described application.
- Naming the Java package for the application and addition version code and version name for the application.
- Declaring necessary libraries and a minimum level of the Android API that the application requires.

The following subsections will have a closer look at different parts of the manifest file, and the manifest file of the *HelloLocationWorld* application will be used as demonstration. The manifest file has to be located in the root directory of an Android project.

2.9.1 Manifest File Structure

The manifest starts with declaring the package name, plus version and name information.

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
      package="ntnu.android"
      android:versionCode="1"
      android:versionName="1.0.0">
```

Then, a permission for the application is added, permitting it to access the API's location updates. Several other permission could be added here and one could also create new ones. If other SDK's are needed for the application, these are added in this part of the manifest with the tag, <uses-sdk />.

```
<uses-permission android:name="android.permission.ACCESS_FINE_LOCATION" />
```

The next part of the manifest is used to describe the application with all its components. Usually, an application is labeled with a name and an icon. This is

to provide users with an clickable icon in the Android devices' menu. After the <application> -tag all components have to be listed. Both activities, services and broadcast receiver are built in the manifest using the same structure. First comes the <activity>, <service> or <receiver> tag, followed by a description of the component's intent-filter and additional meta data. The intent-filter and meta data are not required for all components, as can bee seen in the code below for the activity named *Alert*, which have none of these. Definition of content providers follows a different structure, but is not relevant for this work.

```
<application android:icon="@drawable/icon" android:label="@string/app_name">
        <activity android:name=".AlarmService"
                  android:label="@string/app_name">
            <intent-filter>
                <action android:name="android.intent.action.MAIN" />
                <category android:name="android.intent.category.LAUNCHER" />
            </intent-filter>
        </activity>
      <activity android:name=".Alert"
     android:label="@string/app_name">
     </activity>
</application>
```

After all components have been described, any prospective libraries necessary for the application is defined using the <uses-library> -tag. Then, the manifest -tag is finished, ending the *AndroidManifest.xml* file.

2.9.2 File Conventions

This section will present some general rules and conventions which apply to all elements and attributes in the manifest.

- **Elements**: There are a few necessary elements in the manifest, namely <manifest> and <application>. These elements can only appear once, in contradiction to most other elements which might appear many times or none at all. An application does need other elements than the two required, in order to do something meaningful. The sequence in which the other elements are described does not matter, meaning one can have <activity>, <service> and <receiver> elements unordered. Element values are always set through attributes.

- **Attributes**: The bottom line is all attributes are optional, although some are needed for different components to accomplish its purpose. This means applications does not fail when lacking certain attributes, although they might not function as intended. Some attributes are truly optional as the Android system will use a default value for them in case of absence. All attribute names, except someone used in the <manifest> -tag uses *android:* as prefix.

- **Declaring class names**: Many elements, such as activities, services, broadcast receivers and content providers have corresponding Java classes. These classes need to be referenced in the manifest file and this is declared through the *name* attribute. The name attribute must include the full package designation. If the Java class is part of the same package as defined in the <manifest> -tag, one does not need the full path. Instead, a *."'class name"'* notation is used. The manifest for the *HelloLocationWorld* application uses this notation.

- **Resource and string values**: Attributes might have values, such as labels and icons, that can be displayed to the user. The values of these attributes have to be localized. The above given manifest code have two examples of this. The *icon* attribute uses an icon localized *@drawable* and has the name *icon*. Same goes for the *label* attribute which value is localized *@string* with the name *app_name*.

  The general way of referencing a resource value is by definition *@[package:]type:name*. The package name can be omitted if the resource is within the same package as the application. The type is the type of resource, such as seen above, *@drawable* or *@string*. The *name* is the name that identifies the resource.

2.10 Notifications

The user of an Android device should be able to receive information on events that has occurred, at all times. These messages have to be able to arrive even though other applications are running and have focus. Android solves this by the use of notifications. Notifications are used for displaying messages to the user. This is the only type of messages which can displayed by the Android system at any time, since notifications arrive regardless of what is currently being performed by the Android device.

Figures 2.5, 2.6 and 2.7 shows how the Android device behaves as a notification arrives. When no notification has arrived, the status bar is blank. Then a notification arrives and the status bar is updated with an icon for the notification along with a text. When the user expands the status bar of the Android device, more details on the current notification (along with possible other notifications) is shown. Selecting a notification from the expanded status bar will result in an action such as opening an Android activity.

Figure 2.5: No notification

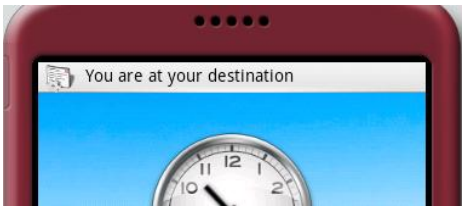

Figure 2.6: Notification has arrived

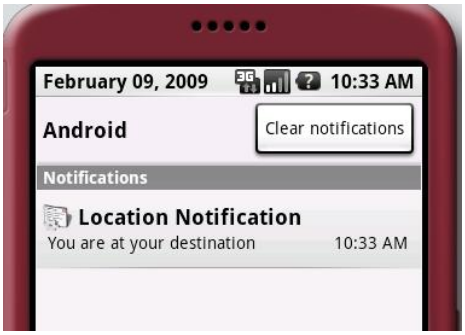

Figure 2.7: Notification bar expanded

Notifications can be created at any time during the life of an application. An Android component, being a broadcast receiver, service or activity, creates the notification object and feeds it to the default Android notification manager. Using this manager, the Android system will start the notification service which in turn will display the notification on the Android device. Below follows a walk through of the code used for displaying the notification in our example application, *HelloLocationWorld*.

First off, the manager is configured using a default Android service.

```
NotificationManager mNM = (NotificationManager)getSystemService(
NOTIFICATION_SERVICE);
```

Then the notification is created using values from the resources. An icon from the *drawable* folder and a text from *strings.xml*. The current system time is used for setting the time in which the notification was created. This is the information shown in figure 2.6.

```
Notification notification = new Notification(R.drawable.icon, getText(
R.string.dest_found), System.currentTimeMillis());
```

The next step is to create a pending intent. This is an intent which is triggered as someone selects the notification. It tells the Android system what activity or service to run on selection. In this case, the activity having the class name *AlarmService.class* will be run.

```
PendingIntent contentIntent = PendingIntent.getActivity(
```

```
this, 0, new Intent(this, AlarmService.class), 0);
```

By now, we can configure the notification to include the pendingintent and set a body text. *R.string.location_found_notif* contains the message which will be shown upon expansion of the status bar, see figure 2.7.

```
notification.setLatestEventInfo(this, getText(R.string.
location_found_notif), getText(R.string.dest_found), contentIntent);
```

All necessary parameters are now set and we can use the notification manager to display the notification on the Android device. The Android device can handle several notifications at the same time. To be able to tell them apart, a layout id is used since it is a unique number. This number is later used to cancel the notification.

```
mNM.notify(R.string.alarm_service_started, notification);
```

All notifications have a set of flags that can be set. The most common, and the one used in this work, is the *FLAG_AUTO_CANCEL* flag which tells the Android system to cancel the notification when it is clicked by the user. Canceling the notification removes it from the status bar.

2.11 Android Project in Eclipse

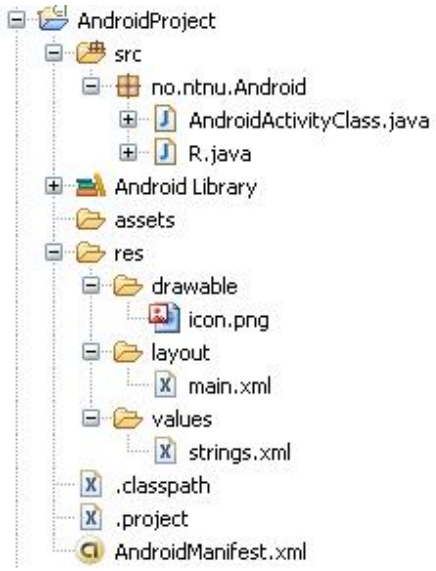

Figure 2.8: An Android project in Eclipse

Figure 2.8 depicts the project hierarchy of an Android project created in Eclipse. The project has one package containing an Android Activity class and

an *R.java* class. These classes are necessary for running the application. The activity class will be the launch class and the *R.java* class is an auto generated class containing references to the application's resources. The project may, as a standard Java project, have several other classes in the same package or in other packages.

Android Library contains the default Java Archive (jar) for Android, but may also contain application specific external jars.

The *res* folder is an important part of any Android project. *res* is short for resources and the sub folders all describe a type of resources which can be used when designing the application. Every picture, screen-layout, and text string used in the application have to be defined in these sub folders. The folder *drawable* will for a large application typically contain several pictures, as applications may use pictures for notification (see 2.10) icons and other pictures as part of different screens. The *layout* folder contain all layouts for every screen used in the application. Each layout demands its own XML file. Lastly, there is the *values* folder, which as standard contain one XML file, namely *strings.xml*. In this file, every text-string used in the application is defined. When resources are added to the folders mentioned above, the *R.java* file is automatically updated with references to the new resources. Writing Android code with use of resources is done by calling the *R.java* file and locating the correct resource.

The *.classpath* and *.project* files respectively describe class path entires and build specifications plus nature of the project. Finally, there is the *AndroidManifest.xml* file which describe the project's components. The manifest is described in detail in 2.9.

Chapter 3

Methods, Tools and Languages

3.1 Arctis and Ramses

Figure 3.1 depicts development using the SPACE approach [11, 19, 20, 21, 10] and the Arctis and Ramses tools. SPACE focus on using collaborative building blocks to compose services, which are transformed into executable state machines and components. The collaborative building blocks is built using UML 2.0 activities, collaborations and external state machines. The Arctis tool [18], developed at NTNU, provides functionality for editing building blocks, collaborations and composing services. Arctis also have a model transformer which transforms the service specifications into executable state machines. Ramses, a tool suite also based on the UML 2.0 repository and developed at NTNU, provides plug-ins contributing to cover modeling tasks. Ramses has a code generator which is used to generate code from the executable state machines. Upon generation, a Java 2.0 Software Development Kit (J2SDK) generator class from Ericsson is called.

3.2 Java JET

Java Emitter Templates (JET) is a tool which generates source code by the use of templates. It is part of the Eclipse Modeling Framework (EMF) project and uses a subset of the JavaServer Pages (JSP) syntax in writing templates expressing the code that is to be generated. The JET template engine may be used to generate, among others, Structured Query Language (SQL), XML and Java source code and is located in the EMF version 2.0 code gen plug-in.

Before the templates can be written, EMF will need to be installed and the java project, in which generation is wanted, has to be converted to a JET project. After creating a folder for the templates and selecting which folder

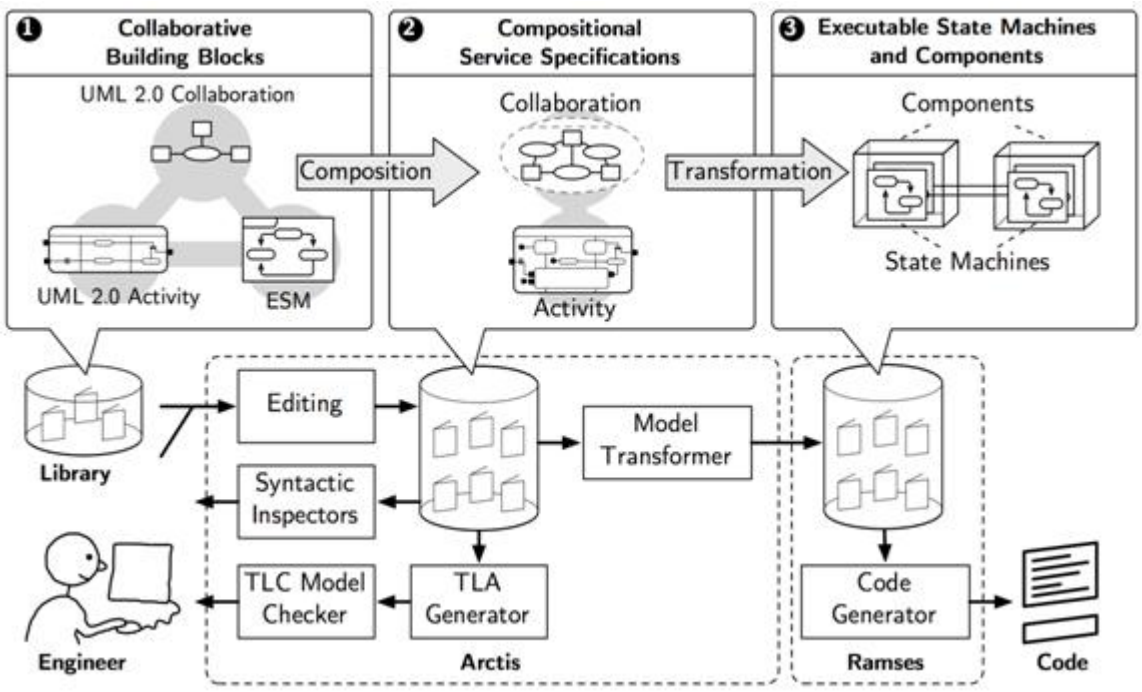

Figure 3.1: Application developement using Arctis and Ramses, from [23]

(usually the *src* folder) will contain the translated templates, JET template files can be created. Each JET template file will have to be declared with a dedicated package name a file name for the translated template.

The template files uses JSP syntax for passing arguments, importing packages and changing tags to the translated template. The translated template will have a public method named *generate(java.lang.Object argument)* which is called upon generation. To build a generated file, a *java.lang.Stringbuffer* is used. The generated file is built line by line using the *Stringbuffers append* method.

The information above is based on the tutorial on JET found at [28].

Chapter 4

Android Application Design in Arctis

4.1 Introduction

This chapter discuss different solutions on how Android applications can be designed in Arctis. Solutions will be proposed as to cover the initial goals of the implementation given below. Design challenges and problems will be considered and discussed before a final solution is proposed. The final solution will be used for the Android code generator implementation.

The main goal is to design applications in Arctis for the Android platform. To achieve this, a set of building blocks that access Android services will be developed. These building blocks provide access to Android services. A common structure for the blocks is wanted, as to enhance design of new building blocks. Another important goal is to design the architecture with regards to code generation, making sure implementation of the design is feasible. The classes generated by Arctis and Ramses should not be altered to a great extend.

4.2 Architecture

An Android application has a designated Android activity which is run on startup. To have all processes run in the background, it is convenient to start the *arctis.runtime.scheduler* in the startup Android activity. Before the scheduler is started, all Ramses state machine classes should be initialized.

The initialization process leads to all objects and threads being spawned from the startup Android activity. This is favorable as the Android system will not kill the startup activity unless there is an extreme demand for memory, see section 2.4. Also, the startup activity can first be killed after all activities stacked on top of the startup activity is killed.

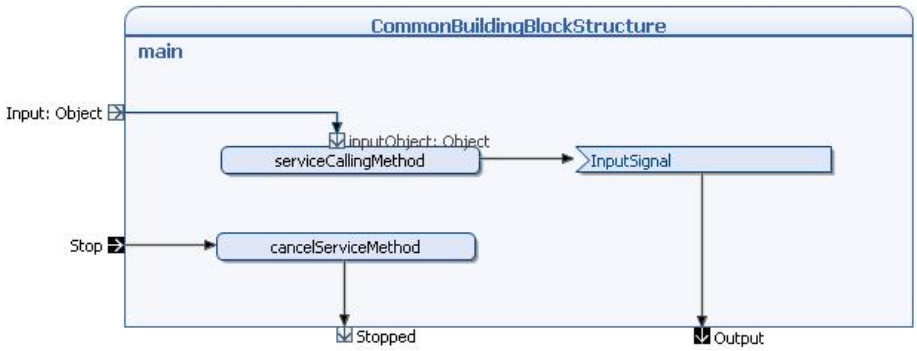

Figure 4.1: Structure of the building blocks

The Arctis building blocks provides access to Android system services. Android services, see section 2.6, consists of a number of managers, which all provide one or more services. This leads to a design decision for the building blocks. Blocks can either represent all services or a single service for a given manager. Some managers, such as the location manager, provides several services. A building block including all possible services for these managers will lead to a complex block with a wast amount of input and output -pins. Also, the building blocks will differ, as a result of the different manager's amount of services provided. How we partition Android services into building blocks may be a matter of taste, but for us a goal was to have a design with a general structure, hence the second solution will be chosen. This results in each building block providing one service. In the following, we will focus on applications without Graphical User Interface (GUI), since GUI development comes with its own challenges not part of this thesis.

The building blocks will be based on the common structure depicted in figure 4.1. A service building block will start with an input object. The object type will vary depending on the service. Then, the *serviceCallingMethod* will be called, setting up the correct service and feeding it with the necessary parameters. Most blocks now goes into an accept input signal action. The signal is usually a confirmation of the service being finished. After receiving the input signal, the building block outputs via a streaming output pin. Every block could have a range of variables associated with them, although none are depicted in the common structure. Nor is any logic for resetting the service depicted, as figure 4.1 provides a simplistic structure of the building blocks.

Some services, such as the *proximity alert* service, will provide a response to the system when an event has occurred. This is reflected in Arctis using an accept signal action (labeled with an generic input signal in figure 4.1). Further, this demands that the Android service is capable of sending signals to the *arctis.runtime.sheduler*. The default Android services are not capable of alternation, meaning there is no way they can send signals to the scheduler. The architectural solution to this is to introduce custom built classes, which will handle all service calls to default Android services. System responses from event happenings will hence be handled by the custom classes, which are able to send signals to the scheduler.

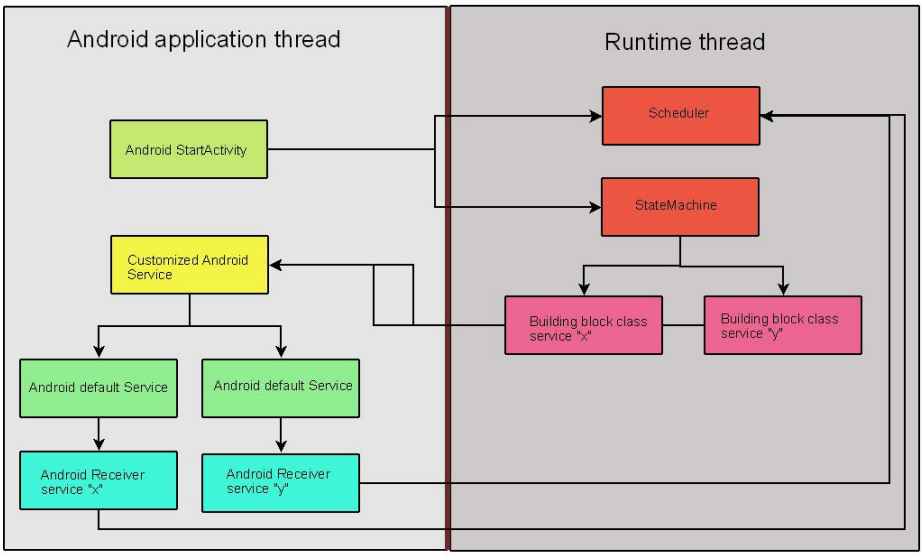

Figure 4.2: Initial Architecture for Android on Arctis

4.2.1 Initial Architecture Design for Android on Arctis

The initial system architecture is depicted in figure 4.2. Java classes are depicted, belonging to one of two threads, *Android application thread* or *Runtime thread*. The first thread contain all classes which are Android components. The second contain runtime classes generated by Ramses and Arctis. Below follows an explanation of the model followed by a discussion on the architecture design.

As described above, the first class that is run is the *Android StartActivity*. This class will be in the Android thread. Both the *Scheduler* and the *Statemachine* will be started from the start activity, but these classes will be run in the separate *Runtime thread*.

The state machine is an auto generated class which instantiates all building blocks used in the Arctis designed system. Figure 4.2 depicts a system with two building block classes for demonstration purposes. If these building blocks are designed for use with Android services, they will make calls to a customized Android service with information on what service they will have initiated. The custom service class will be created upon the first call. From the second call and on, only a certain part of the class's code will be run, see section 2.6.1.

Customized Android Service, see section 4.2.5, will start default Android services on demand from the building blocks. Setup of the default service, along with information on what Android receiver class the service will respond to is the responsibility of the customized Android service. This setup is possible as the class is an Android service and hence is running in the Android thread.

The *Android receiver* classes, see section 4.2.4, are designed to be corre-

sponding with a building block. Typically, a building block sets up a service before it is prepared to receive an incoming signal. It is the receiver class' responsibility to send the correct signal to the correct state machine. The signal will be sent when the default Android service triggers the Android receiver as a result of certain device events.

Review of Initial Architecture

An apparent question regarding the initial architecture is the choice of using a customized Android service class. Why are not the building block classes accessing the default Android services directly? The answer is a result of several factors.

- First of all, if the building block classes were to setup default services directly, they would need to be able to access default Android services using the *getSystemService* method. Usually, this implies that building block classes extend either *Activity* or *Service*.
- For the Android system to be able to include the building blocks as Android activities or services, they would need to be started using *startActivity* or *startService*. This requires that either the *Android StartActivity* having knowledge of all building blocks or the *StateMachine* being able to call *startActivity* or *startService*. As each Arctis designed application could have a varying range and amount of building blocks it seems unfeasible for the Android start activity to have knowledge of all. Having the state machine calling *startActivity* or *startService* contradicts the goal of not alternating the Ramses and Arctis code generator.

There is however a significant drawback to the initial architecture. For every new building block added to the library, the service class should be updated as to handle calls from the new building blocks. The challenge is not updating the code itself, but rather providing access to the class containing the code. The customized service class will be part of the *Java SE for Android generator* and hence generated upon deployment. A developer wanting to update this class needs access to the code generator for Android. To give a wast amount of developers access to the source code of the Android generator is unwanted.

An architecture without the customized service class would allow developers to design new building blocks without altering the code generator.

4.2.2 Revised Architecture

Figure 4.3 depicts the architecture without the customized service class. Removing the customized service class implies that the challenges referenced above in 4.2.1 has to be resolved.

getSystemService can be called without the building block being either an Android service or activity if the Android *StartActivity*'s context is used. This is the same context used when starting the customized service class in the initial

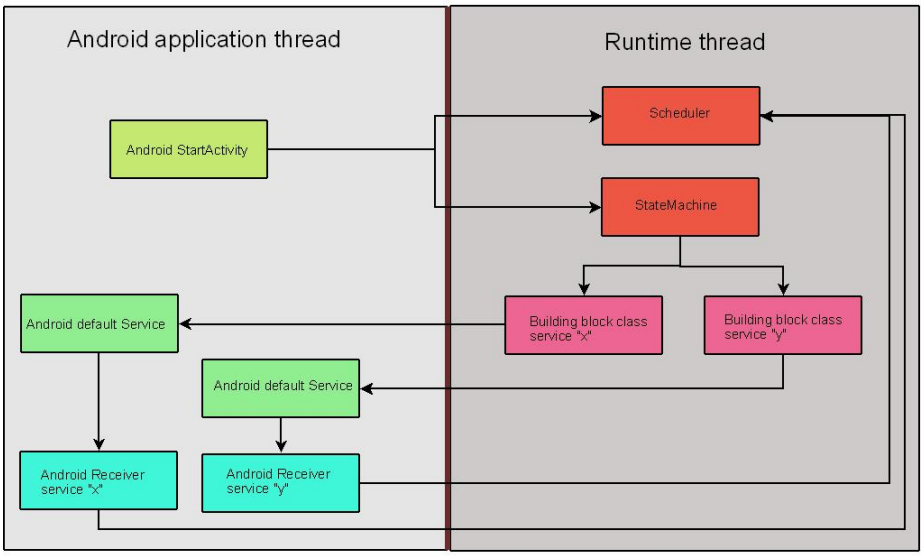

Figure 4.3: Revised Architecture for Android on Arctis

design. The building block will now have the possibility to create and use Android service managers. How the context is referenced is discussed in section 4.3.

This solution has one essential difference from the initial, namely where the code responsible for setting up the default services are run. In the initial solution this was done in the customized service class, an Android service component and part of the Android application thread. In the revised solution this is done within the building block, which is neither an Android component or run in the Android thread. This has implications on what default services can be called from the building blocks.

Some services, such as listening to the telephony state or displaying notifications, demand that this is done from an Android application thread, meaning this cannot be done directly from our building blocks as the runtime thread is not an Android thread. These services are **thread dependent**. Other services, provided by e.g the location manager or alarm manager can be called directly from our building blocks.

The above implications require another revision of our architecture.

4.2.3 Final Architecture

Figure 4.4 depicts our final version of the architecture. This is a hybrid from the two previous version. Some building blocks will be able to reach the default Android services directly while the other, thread dependent, have to call the service via our customized service class.

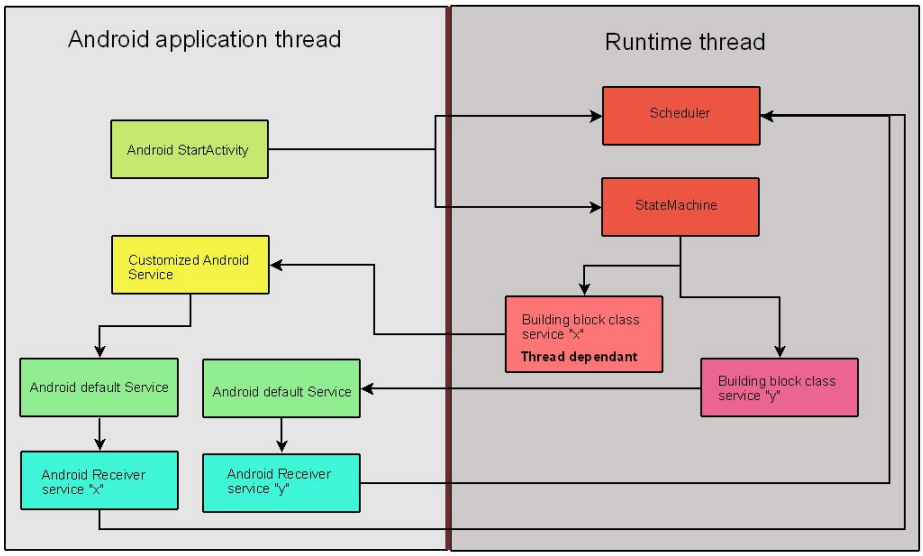

Figure 4.4: Final Architecture for Android on Arctis

This architecture will allow developers to create new building blocks without having access to the code generator as long as the services used are not thread dependent.

A simplified sequence diagram for setting up the proximity alert (non-thread dependant service) and a call listener (thread dependent service) are given below, figures 4.5 and 4.6.

Our proposed architecture has some drawbacks. The ideal solution would be to have specific code for a certain service contained within the building block for all services. The receiver class should also ideally be contained within the building block class as this would remove the need for specific receiver classes being declared in the Android application. Unfortunately, this could not be achieved in this work due to the problem regarding separate threads.

An apparent solution to both of the above problems would be to adjust the runtime scheduler into an Android service, and the state machine and all building blocks into Android components. This would result in all code being run in an Android thread, hence allowing service calls and receiver classes to be contained within the building blocks. However, a setup like this would demand complex alternation of the current scheduler and state machine. All building blocks designed for use with Android would have to be started as Android components, whereas other building block classes should be initiated the original way. This alternative solution is not feasible for us in regards to the goal of only small alternations to the classes generated by Ramses and Arctis.

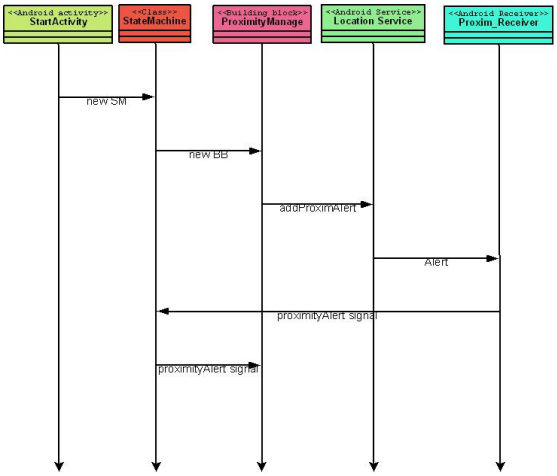

Figure 4.5: Non-thread dependent service initiation

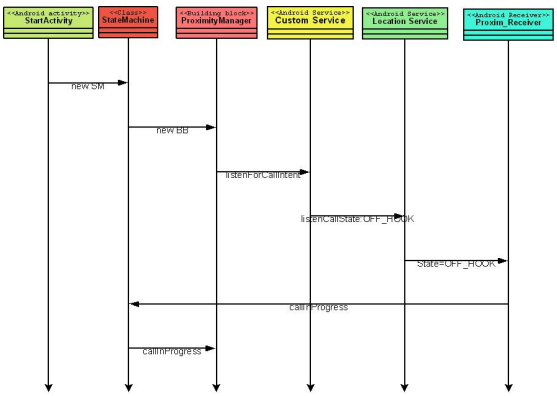

Figure 4.6: Thread dependent service initiation

4.2.4 The Receiver Classes

The task for the receiver classes is to send signals to the scheduler. The signals should reflect the ones the corresponding building blocks is prepared to receive. The role of the receiver class can be seen in figures 4.4, 4.5 and 4.6. The scheduler and the state machine address necessary for sending signals is retrieved from the Android start activity. The default class code for a receiver class is given below.

```
public class ReceiverName extends BroadcastReceiver {
private Scheduler scheduler;

public void onReceive(Context context, Intent intent) {
scheduler = Start.scheduler;
scheduler.sendToMe(Start.SMAdrTable.get("SMName"), "signalName");
}
}
```

The *SMName* reflects the name used when adding the address to the hash table in the start activity class. This work concerns systems with a single state machine and hence will we use a universal name for the state machine. Addressing in systems with multiple state machines is left as future work.

4.2.5 The Customized Service Class

The customized service class binds together building blocks with the default Android services they want to access. Inside this class is all the logic for setting up default Android services based on input from the building blocks.

In order for the customized service class to start a service, all necessary information has to be provided. This information might consists of user specific input. Android allows information to be sent to a service by attaching it to the *Intent* which starts the service. Attaching and retrieving information is respectively done by *putExtras(name, value)* and *getExtras().get(name)*. This means that e.g a notification object could be passed along from the building block to the customized service using the intent starting the service.

As there is one customized class, but possibly several building blocks making calls to it, there is a need to be able to separate the different calls. This is done by having each building block add two *Extras* with information on what Android service manager should be used and what type of service is wanted.

There are one last *Extra* which is used for every thread dependent building block. This is a boolean value which indicates whether or not the service should be stopped. When a user of the Android application wants to shut down a service, this extra's value is set to *true* before the intent is sent from the building block. Our customized service class will read this value and stop the service for the given building block if true.

All of the logic in the customized service class is written inside the class' *onStart* method in accordance to the service lifecycle, see section 2.6.1. The *onCreate* method cannot contain the logic as this is only run the first time the service is started. *onStart* on the other hand is run each time *startService* is called.

Below follows some demonstration code on how the service class works.

```
public void onStart(Intent i, int StartID){
NotificationManager nMN = (NotificationManager)getSystemService(
NOTIFICATION_SERVICE);
TelephonyManager tMN = (TelephonyManager)getSystemService(
TELEPHONY_SERVICE);
```

First *onStart* is called and the Android managers are retrieved by calling *getSystemService*.

```
if(i.getExtras().get("ManagerName").equals("TelephonyManager")){
try{
tMN.listen(new PhoneStateListener(){
public void onCallStateChanged(int state, String incomingNumber){
super.onCallStateChanged(state, incomingNumber);
switch(state){
case TelephonyManager.CALL_STATE_OFFHOOK:
Intent TelephonyIntent = new Intent(
AndroidServiceForArctisBuildingBlocks.this, TelephonyReceiver.class);
sendBroadcast(TelephonyIntent);
break;
}}, PhoneStateListener.LISTEN_CALL_STATE);
catch(Exception e){System.err.println(e);}}
```

Above is the code snippet from the case of the *ListenForCalls* being set. First, a check is done for the manager name. Then, a phone state listener is setup. In case of the state being *OFFHOOK*, an intent is created determining what class should be run next. This intent is broadcasted so the receiver class can receive it and send the correct corresponding signal to the scheduler.

Every thread dependent service is started in this fashion, by using else-if sentences checking the manager name.

4.2.6 The Common Screen Layout

Every application created using the *Java SE for Android* generator will be equipped with a common screen layout displaying two simple buttons. One for starting the service(s) the application provides and one for stopping it. The XML file describing this layout is added to the generator and will be used in the *onCreate()* method of the Android start activity. The common application screen is shown in figure 4.7.

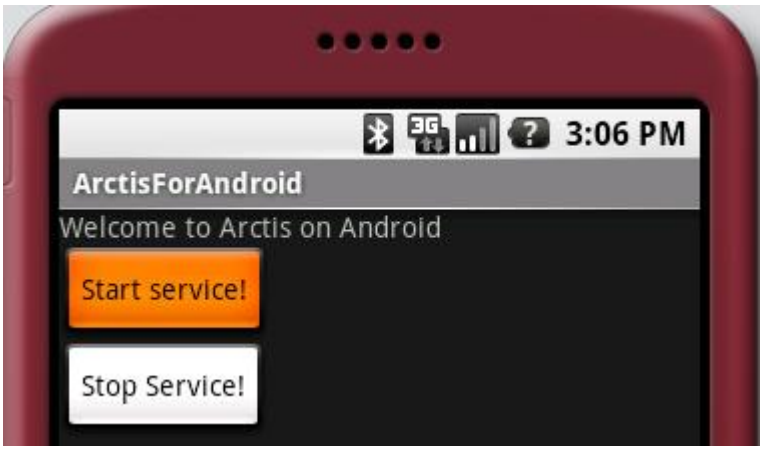

Figure 4.7: The common screen layout for applications designed in Arctis

To enable the buttons to work, listeners have to be set. The method called when *Start service!* is pressed perform setup of the scheduler, state machines and other variables. The scheduler is started. The method called for *Stop service!* will send a *stop* signal to the scheduler, initiating shutdown of all started services.

4.2.7 Permissions

The permissions, see section 2.8 and 2.9.1, used in our architecture will reflect the ones necessary for having all our designed building blocks run after deployment. This is not the best solution possible as there will be applications developed which does not use all of our building blocks. E.g will an application which does not use the location service still ask for permission to access the phones location. This might cause dis concern to the user upon installation.

As our solution does not involve dynamic creation of the XML files, such as the manifest, the recommended solution to this will be to develop applications as normal and remove the excess permissions before the application is published.

4.3 The Thread Problem

Android applications designed in Arctis and running the *arctis.runtime.scheduler* will be run in two major threads, because the generated scheduler is not defined as an Android component. One thread will run the Android activities, services and other Android components, while the other will run the scheduler and the state machines, as seen in figure 4.4. The challenge is to enable classes spawned from the two separate threads to make calls on each other. The solution will have to take the code generation process into consideration.

The building blocks have to use an *android.content.context* object when call-

ing upon services. This is usually done by calling *startService(Intent intent)* for a typical Android application. The *startService* method call is only available for classes which have an *Activity*, *Service* or *Receiver* extension, hence being an Android component. An apparent solution is to extend the Arctis building block classes with the *Activiy* extension.

But, even after adding an extension, one cannot call *startService* from the building block class. The main problem is that the context is only set when the class is created from another Android component, using the correct method call. In our case, the building block classes are instantiated upon creation of the state machines. The auto generated state machine class will initialize the classes using the *new class* call and not one of the required Android calls such as *startActivity*. Also, since the state machine is not an Android component, calls to *this.context* in the building block classes will return **null**. The building blocks does not acquire the necessary Android context required for starting services.

The second problem is that the above described custom service and receiver classes can not reference and send messages to the scheduler, as the service and the scheduler are running in separate threads. This will be solved using a *java.util.hashmap* where all state machine addresses are saved upon creation. The hash map will be public and localized in the startup activity, enabling all Android activities to retrieve state machine addresses.

The below subsections will describe two different solutions regarding the context problem before concluding on which is chosen.

4.3.1 Solution Alternative 1

The first solution is to provide the state machine classes with an Android context on initialization. This allows the state machine classes to provide the building block classes with a valid context which can be used upon service calls. To achieve this, both building block - and state machine classes need to add a new constructor, taking the context object as a parameter.

The context used will be the one of the class that initialized the state machine, in our case, the Android startup activity. Calls made to start an Android service will then be perceived by the Android system as coming from the startup activity. As the startup activity is the last activity killed before application shutdown, the context reference will be usable during the lifetime of the application.

Implications from this solution is to add to the current Arctis generator, enabling it to handle creation of state machine classes with a context as parameter in addition to the standard scheduler. Also, the generator needs to be able to separate Android building block classes from other building blocks, as only the Android classes makes use of the new constructor.

4.3.2 Solution Alternative 2

The second solution is based on making the building block classes able to reference the Android context without the use of a constructor parameter. To achieve this, the scheduler will be used. A hash map is added to the scheduler, enabling it to hold Android context objects.

During startup, the Android context is added to the scheduler's hash map. As the building block classes are part of the same thread as the scheduler, they can access the hash map by making calls to the *AbstractRuntime*.

This solution demands a quick and simple alternation of the scheduler class and the abstract runtime.

4.3.3 Conclusion on Thread Problem

The conclusion is based on the overall implementation goals listed in the introduction of this chapter. Both solutions were tried implemented and they both worked on examples.

The first solution demanded addition of a constructor class extending the state machine class. The Arctis generator was adapted and the state machine were started in the Android startup activity using *new StateMachine(scheduler, context)*. The solution ran without any problems. However, the example application used for testing consisted only of building blocks developed for use with Android. If other building blocks were to be used, the state machine class could not start them as they have no constructor which includes an Android context. To solve this, the state machine class would have been able to differentiate between Android building blocks and regular ones. This would demand severe alternations to the current generators, contradicting an implementation goal.

The second solution demanded adding of a hash map to the scheduler. The hash map could be reached through the interface of the scheduler. Unlike the first solution, this one has the strength of demanding no alternation to the state machine class. The solution will work just as well for regular building blocks and building blocks built for Android use.

The second solution will be chosen as it requires the least alternations and has no apparent drawbacks.

4.4 Considerations

This chapter has discussed thread dependent and non-thread dependent Android services. Our solution will contain building blocks of both types. The Android API contain a wast amount of other services. A complete list or overview over which of these are thread dependent or not are not included in this work, as the documentation on the API does not state in a clear fashion what services needs to be called from an Android thread.

4.4.1 Strengths

The main strength is that, given the library consisting of Android building blocks, applications can be rapidly designed and deployed on an Android device. New building blocks can be designed for non-thread dependent Android services without necessary alternation of generated classes. The default design for building blocks enhances design of new ones.

Our architecture relieves the building blocks created for use with Android from the constraint of having the *Activity* or *Service* extension. This simplifies the Android manifest. Another benefit is regarding multiple instances of building blocks. Our solution uses one receiver class for every building block no matter how many instances might be used in the system design. With the *Activity* or *Service* extension, multiple instances in the system design would result in multiple descriptions in the manifest.

4.4.2 Challenges

The biggest challenge is to update the customized service class. For every thread dependent building block added to the library, the service class have to be updated as to handle calls from the new building blocks. The challenge is not updating the code itself, but rather providing access to the class containing the code. The customized service class is part of the *Java SE for Android generator* and is generated upon deployment. A developer wanting to update this class, pre-generation, needs access to the code generator for Android. To give a wast amount of developers access to the source code of the Android generator is unwanted.

There are two possible ways to handle the above challenge. Either by having developers provide code to someone who has access to the code generator and is able to update it, or by having the developer insert code after deployment. During development and testing of applications, deployment might be done several times. There is a drawback if code has to be inserted after each deployment, as this is time consuming.

The customized receiver classes have the same challenge, and possible solutions, as described above. Most of the building blocks accepts and responds to incoming signals and these are provided from the receiver classes depicted in the architecture, figure 4.4. The Android receiver classes used for handling the events have to be created as part of the the Android project and described in the Android manifest.

Chapter 5

Manual Adaption of Generated Code

Arctis provides an implementation option were an application can be deployed to Java SE. This makes the application run on a Java platform, but the generated application does not satisfy the requirements of an Android application. In the process of developing a code generator for Android, the steps needed to make the application runnable on Android is significant for what the *Java SE for Android* generator should cover.

This chapter explains the adjustments and additions needed to make a generated Arctis application run on Android and is outlined as follows. First we take a look at the example application, *HelloLocationWorld*, and how it is designed in Arctis. The Arctis model will be implemented as a Java SE project. Then a step by step explanation follows, describing in detail the changes needed for making the application run on Android.

Discussions and figures in this chapter will be based on development using the Eclipse tool, with an installed Android SDK [4].

5.1 The Example Application - *HelloLocationWorld*

The example application, described in section 2.2, generates a location and adds a *proximityAlert* using the Android location manager. As the device reaches a predefined radius of the desired location, a notification will be displayed to the user of the Android device using the notification manager. When designing the system in Arctis the solution was to make one building block for handling the proximity alert and one for handling the notification. The system behavior is depicted in figure 5.1.

As UI is not relevant for this work, the common layout (see section 4.2.6)

will be used for the generated version of *HelloLocationWorld* instead of the one depicted in figure 2.2.

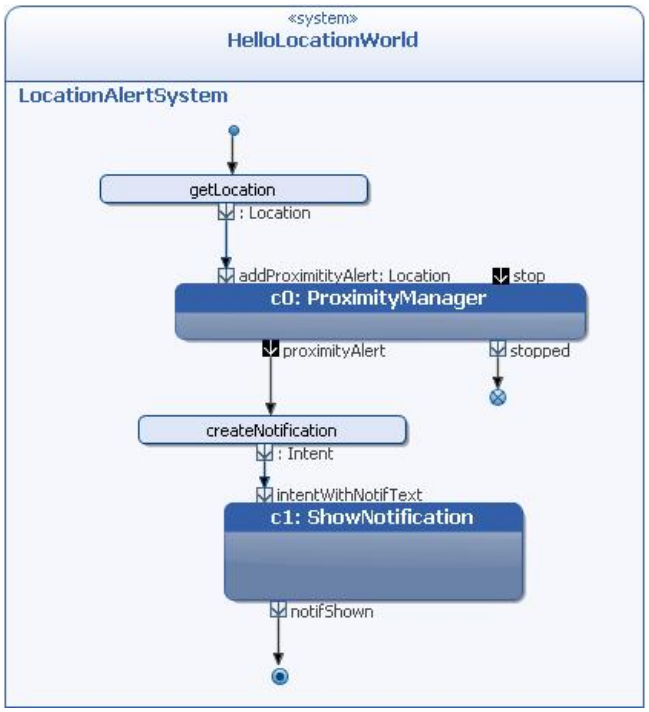

Figure 5.1: *LocationAlertSystem behavoir*

HelloLocationWorld, figure 5.1 starts with the method *getLocation* being called. This method generates an Android location object which is fed to the *ProximityManager* building block, see section 7.4 for a detailed look at this building block. *ProximityManager* makes use of the Android location manager and sets up a proximity alert, hence the name of the building block. *ProximityManager* outputs via the *proximityAlert*-pin, and triggers the *createNotification* method. This method generates an Android notification which is fed into the second building block, *ShowNotification*, described in detail in section 7.2. *ShowNotification* terminates via the *notifShown*-pin, ending the UML-activity.

5.2 Adapting to an Android Application

There are several necessary steps in adjusting the generated code for running on an Android device. The following subsections each address a part of the adaptation process and covers what needs to be done to have our Arctis designed system, *HelloLocationWorld*, implemented and treated as an Android project. The subsections are divided as to cover general areas of importance when adapting Java SE applications to working Android applications. These areas will be addressed under development of the code generator, described in chapter 6.

5.2.1 Implementing And Moving the Project

Code generation is needed as to deploy the system as a runnable Java project. Hence, the first step is to implement the example system and deploy it as a Java SE project. This is illustrated in figure 5.2.

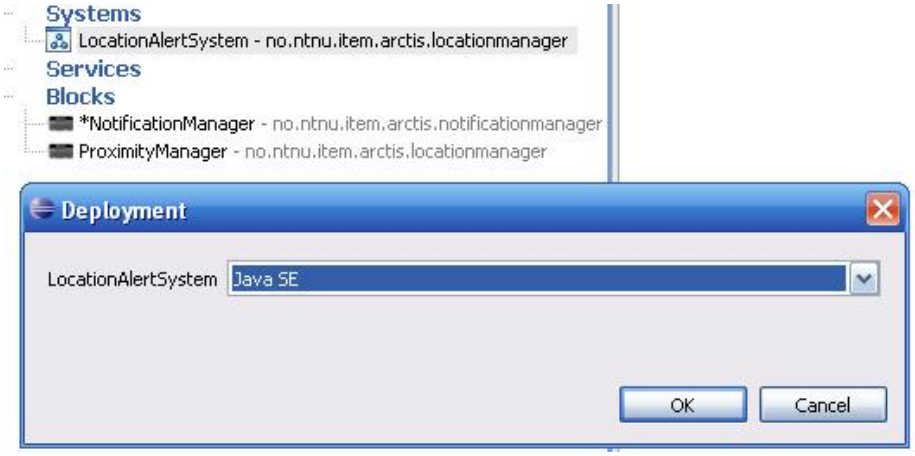

Figure 5.2: Implement and deploy as Java SE

To make the example application run on an Android device, the project has to include all necessary components of an Android project, described in 2.11. After implementing the system as Java SE, the resulting Java project obviously has no Android nature. The project needs to have the Android nature. Solving this is done by creating a new Android project. The newly created project will automatically have the correct Android nature and contain all the mentioned necessary components.

Figure 5.3 depicts the creation of a new Android project. Project name and all property names may be chosen freely, but since the launch class in our generated project is named *Start.java*, it is convenient to have the Android project's *Activity name* set to *Start*. This results in the *Start* activity being set as the launch class of the Android project.

By now, code is generated and the Android project is created. The generated packages should to be moved into the newly created Android project. This is done by marking all packages in the source folder (*src*) of the generated *packagename.Systemname_exe* and dragging them into the source folder (*src*) of the Android project. This completes the first phase of the adjustment process, as we now have our generated code in an Android project, effectively solving the nature and component problems addressed above.

5.2.2 The *Start.java* Class

The deployment of the system generates a launch class named *Start.java*. This class sets up the state machines and a scheduler for the implemented system.

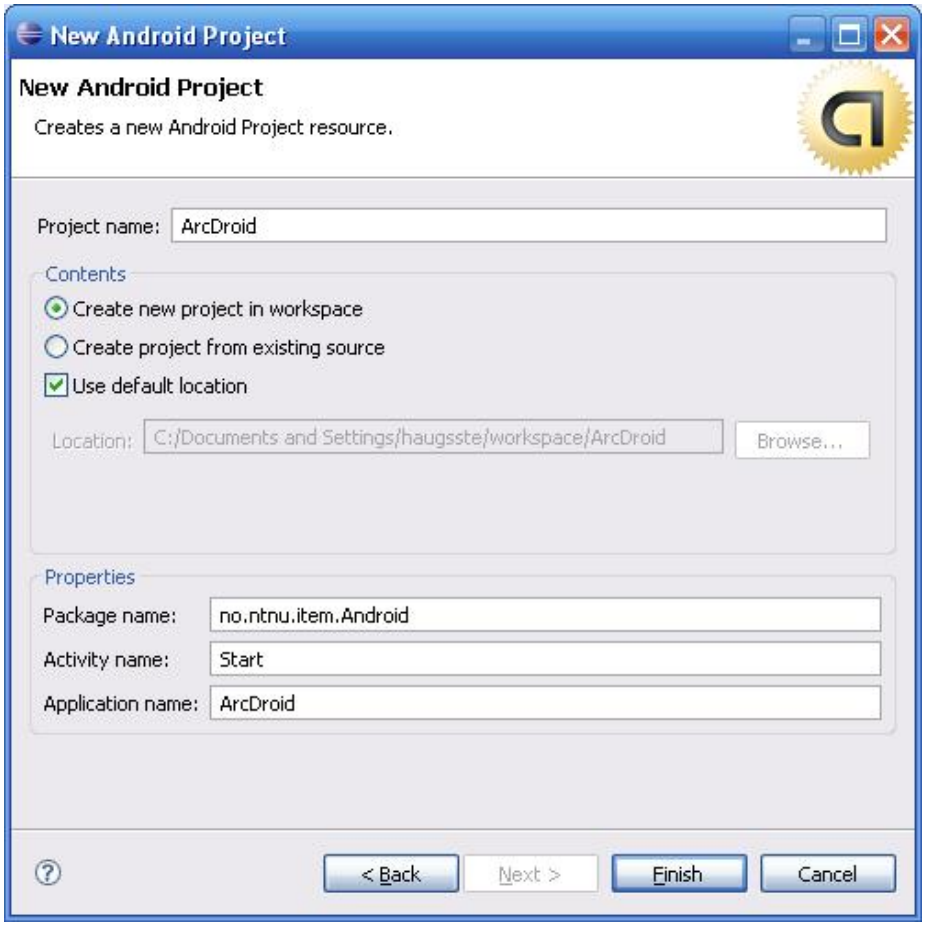

Figure 5.3: Create a new Android project

The content of this class have to be moved into the launch class for the Android application, mentioned in 5.2.1, in order for the system to run on an Android device.

In addition to moving the generated code into the Android project's *Start.java* class, there are several other changes needed in order to have the system behave as required, running on an Android device. Below follows an explanation of the *Start.java* class, describing changes, adaptations and the reason they have been done.

```
public class Start extends Activity{
```

First we declare the class name which extends *Activity*. The extension is necessary for the Android application to run on startup and also enables the use of the *Start.java* context to be used as a launcher of Android services and other Android activities.

```
public static Scheduler scheduler;
public static HashMap<String, String> SMAdrTable =
new HashMap<String, String>();
```

Then, we create a public *Scheduler* and a *HashMap* able to hold String values containing addresses to the system state machines. The reference to the scheduler and the hash map is used as to address the *The Thread Problem* described in 4.3. *SMAdrTable* and *scheduler* will be used by Android classes in instances where a signal is going to be sent to the scheduler. Due to the thread problem, the Android activity class *Start.java* is the only class where a reference to the scheduler is set.

```
    public void onCreate(Bundle savedInstanceState) {
super.onCreate(savedInstanceState);
```

Is the default Android code needed for every activity. The *onCreate(Bundle savedInstanceState)* method is called as to tell the Android application what code to run as the start activity is created. All code necessary for setting up button listeners in accordance to our common screen layout, see section 4.2.6, is done here.

```
int socketServerPort = GlobalRoutingConstants.JAVA_STANDARD_TCP_SERVER_PORT;
Serializer serializer = new SoapHandler();
SocketTransporter transporter = new SocketTransporter(socketServerPort);
Router router = new JavaStandardRouter(serializer, transporter);
scheduler = new Scheduler(router);
```

The code above are the default code generated by the Java SE generator for setting up the scheduler, with one exception. In the last line, the public static scheduler object is initialized, but not started. The generated code starts this object as default, but in order to reference the scheduler object, startup of the scheduler is postponed.

```
scheduler.addContext("Android", Start.this);
```

Invokes the method *addContext(String, context)* on the scheduler. This method adds a reference to the *Start.java* class, referenced by *this*. Adding a context to the scheduler object is necessary for an Android application as to allow UML activities to start Android activities and services. The creation of this method is a result of the discussion on threads described in 4.3.3.

```
LocationSystemSM ls = new LocationSystemSM(scheduler);
SMAdrTable.put("LocationSystemSM", ls.getId());
```

This will create the state machine for the system and add the state machine address (retrieved by *getId()* to *Start.java's* hash map, *SMAdrTable*. The state machine address is now available for being referenced by Android classes.

```
new Thread(scheduler).start();
```

Finally we create a new thread, starting the scheduler.

5.2.3 Add Android Specific Classes

The solution for creating Android applications in Arctis involve use of an Android service class for handling service calls, as well as receiver classes. The receiver classes are Android broadcast receivers which handle responses from Android services as described in chapter 4.

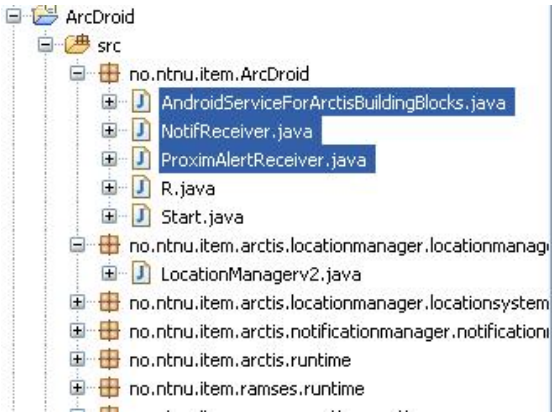

Figure 5.4: Copy Predefined Android Classes Into the Android Project

Figure 5.4 depicts the pasting of the three classes needed for the example application solution. These classes enables the execution of our example application, described in 5.1. A detailed description of the pasted classes are part of chapter 6.

5.2.4 Update the Android Manifest

The Android Manifest, see 2.9, needs to be updated with entries for the activities, receivers and services added in 5.2.3. Also, permissions are required depending on which services are used in the application. Below follows an explaination of the updates done in the *AndroidManifest.xml* file.

```
<uses-permission android:name="android.permission.ACCESS\_FINE\_LOCATION" />
```

First we add the permissions of *ACCESS_FINE_LOCATION* inside the *manifest* tag. This permissions allows the application to access the mobile device's Global Positioning System (GPS) unit and hence derive location updates as they arrive. This is a necessity for the example application, as a proximity alert, which require access to location updates, is added.

```
<service class="AndroidServiceForArctisBuildingBlocks" android:name=".
AndroidServiceForArctisBuildingBlocks">
<intent-filter>
  <action android:value="no.ntnu.item.ArcDroid.TESTSERVICEv2.0"
  android:name="android.intent.action.TESTSERVICEv2.0" />
</intent-filter>
</service>
```

The above lines add the developed serivce class, described in 4.2.5, to the manifest file. First, the class name is declared, followed by the android name for the same class. In the next lines an intent-filter, see 2.7 is added. The *android:name* of the intent-filter provides a means for classes in the Android application to use the service. Classes create an intent which name equals *android::name* and hence, the service class will start whenever such an intent is used as a parameter in the *StartService(Intent)* method.

```
<receiver android:name=".ProximAlertReceiver" android:label=
"@string/proxim\_name" />
<activity android:name=".NotifReceiver" android:label=
"@string/notif\_name" />
```

Finally, we update the manifest with entries for the alert receiver and the notification activity. These entries consists of only an *android:name* along with an *android:label* which is a text string containing the name of the activity. The label text will be used as a headline for the activity if put to the foreground.

5.2.5 Resources

The example application makes use of additional string values, as explained in 5.2.3. The code below is from the XML file describing the different string values available for the Android application. *proxim_name* and *notif_name* are declared as required by the manifest file.

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
<string name="hello">Hello World, Start</string>
<string name="app_name">ArcDroid</string>
<string name="proxim_name">Proxim</string>
<string name="notif_name">Notif</string>
</resources>
```

Also, the application makes use of an logo for its notifications. This logo needs to be added to the Android application's *drawable* folder. This is obtained simply by copying the image file named *ntnu.png* to the correct folder, as depicted in figure 5.5.

Figure 5.5: Add logo image

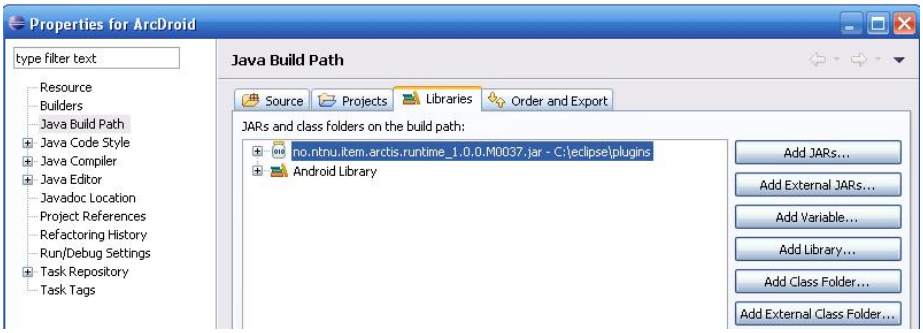

Figure 5.6: Step 10: Add build path for the updated Arctis runtime

This completes the adaptation of an Java SE generated project into an Android project. The Android project is now runnable on an Android SDK's device.

Chapter 6

The Android Code Generator

Section 5.2 described changes needed in order to enable Arctis designed applications, see 3.1, to run on an Android device. Changes included alternation of the generated *Start.java* class and addition of an Android service class, an Android activity and receiver classes. These classes should be generated by our *Java SE for Android* generator and added to the Android project at the correct package and folder location.

This chapter will describe how the necessary changes are handled by our *Java SE for Android* generator. Our generator will be developed based on use of the existing code generators in Arctis [22, 25, 29], Java builder, described in Eclipse Plug-in Development Environment (PDE)[14], the Java Development Tool (JDT) [13] and the EMF [12].

6.1 Generation Overview

Figure 6.1 depicts an overview over what packages, classes and files are contributed by the Ramses, Arctis and Android parts of our *Java SE for Android generator*.

Ramses contributes the Ramses runtime environment including the scheduler. Transport and routing used by the scheduler is also included.

The Arctis part generates packages for all building blocks used in the system design. It also generates the system package. The system package include the system class describing system specific methods and variables. The state machine class are also a part the system package. The Arctis runtime provides classes enabling interaction with the scheduler generated by Ramses.

The Android part is the focus of this work and will generate an Android system package along with a file system. The package consist of the *Start* class

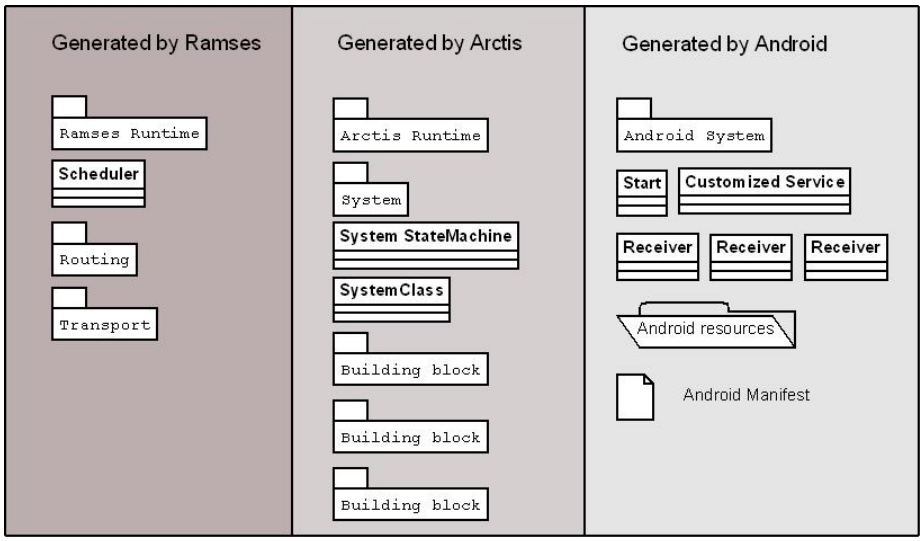

Figure 6.1: Resources contributed by which part of the generator

(an Android activity), a customized Android service class and several Android broadcast receiver classes. All elements of our Android in Arctis architecture, section 4.2. The file system consists of a folder containing Android resources and the Android manifest, both required parts of an Android project.

6.2 Generation Components

6.2.1 XML Generators

The XML generators are included as to generate all necessary XML files for the Android project. There are three types of resources in which two of them, *layout* and *values*, contain a document describing the resource. The Android manifest file, section 2.9, is also an XML file).

All XML generator classes are constructed in the same manner. The string generator class code is used to illustrate.

```
public class AndroidStringsGenerator {
private static final String N = "\n";
public static String generateLayout() {
StringBuilder b = new StringBuilder();
```

First off, a final string object is created which will be used whenever a line break is wanted. Then the method of the class, *generateLayout* is defined with a string return object. A string builder, used for building the resulting XML file is instantiated. Next is the line by line addition of the content of the XML file.

```
b.append("<?xml version=\"1.0\" encoding=\"utf-8\"?>" +N);
b.append("<resources>" +N);
b.append("<string name=\"hello\">Welcome to the application</string>" +N);
b.append("<string name=\"app_name\">ArctisForAndroid</string>" +N);

return b.toString();
```

Above is the first lines of the *strings.xml* file. Each line is added to the string builder using the *append* method. All lines of the original XML document is added this way until the method return the string builder as a string.

The below subsections describes the generation of the different XML files. The subsections also discuss the files in regards of dynamic creation, expansion and adaptations to new systems.

Strings Generator

An Android project needs a *strings.xml* file declaring all text strings that will be used in the application. There are no restraints on the use of textstrings decleared in *strings.xml*, meaning it is possible to add text strings even though they are not used in the application. Although, it is important to ensure that a minimum of text string are represented in the *AndroidStringGenerator* class. This minimum consists of an application name and names for every additional Android activity, service and receiver generated by our *Java SE for Android* generator.

Layout Generator

The layout generators goal is to create a basic layout for the Android application. An Android application does not need to have described a layout, if the application does not contain any screen pictures. Although layout and screen pictures are of small focus in this thesis, our Android code generator will generate a simple layout XML file. This layout will simply put a text string, described in the *strings.xml* file, on the main screen of the Android device. The layout will also provide a couple of buttons, one for starting the generated system, and one for stopping it.

Layouts are developer dependent, some might want to use rich screen pictures, whereas others want simplistic layouts to go with their applications. Generating screen pictures suitable for a wide range of applications and developer desires is hence out of the scope of this thesis. When a developer wants to add new layouts, this can be done by using the Android layout tool which is a part of the Android SDK.

Manifest Generator

The Android manifest, see 2.9, is one of the most important parts of an Android project. The application will not run as intended if not all activities, intent-

filters, services and other Android components are declared in the manifest. On the other hand, one cannot describe components in the manifest file, which does not exist in the Android project. The above statements need attention during generation of the Android manifest file.

The architecture used when building Android applications using the Arctis tool, see section 4.2, is developed with consideration to the manifest. The solution is built around one central service class, section 4.2.5, which UML activities may use at any time. The service class is part of any project being implemented for Android and because of this, the service can be described in the generated manifest file. The Android activities and receivers used in the solution for service results are also generated for every Android implementation. Hence, the receivers and activities may also be described in the manifest at the generation stage.

The description of the service and the receiver activities will be as described in 5.2.4.

As described above, the Android manifest file can be generated with fixed descriptions of both Android activities and services. No further adaptations are needed when using the building blocks provided in the Arctis editor. When the generator is expanded with new services and receiver classes, the manifest file generator needs to be updated with descriptions of the new Android activities.

6.3 Templates

This section describes how the launch class (*Start.java*), the Android Service class (*AndroidServiceForArctisBuildingBlocks.java*) and the receiver classes are created by the *Java SE for Android* generator. JET, see 3.2, will be used for generating source code and hence, all templates have the **.javajet* extension.

Every **.javajet* file used in the Android generator starts off with the same structure. Below is the first lines of the *javaandroid.proximreceiver.javajet* file as to give an illustration of how the templates are written.

```
<%@ jet package="no.ntnu.item.ramses.generator.android.javase.jet"
imports="java.util.List java.util.Iterator"
class="JavaForAndroidProximReceiverGenerator" %>
```

Declares the packagename, the imports and the class name of the JET generated **.java* file. When saving the **.javajet* file, a **.java* file named *JavaForAndroidProximReceiverGenerator* will be created in the given package. The created file will have the imports declared in the second line above.

```
<%  List args = (List)argument;
    Iterator i = args.iterator();
    String pckg = (String)i.next();
%>
package <%= pckg %>;
```

Every **.java* file created from a **.javajet* template will contain a method name *public String generate((Object argument)*, see 3.2. The code snippet above makes use of this argument in order to set the correct package name for the generated class. The argument passed along will have to contain a *java.util.List* object. An iterator is assigned the list, and the first list object contains a *String* with the package name used under generation. This package name is used when the package name for the generated file is set.

After the above initialization, the remaining class code is pasted in the **.javajet* file with no further alternations necessary, unless the code makes use of some input from the list or iterator.

6.3.1 Start Class Generator

The start class generator's initialization process demands more than described above. Every generated state machine class, see 3.1, from the Arctis design will have to be imported into the start class. These state machines should also be created during startup. The list of arguments passed to the template also include lists of imports and state machines. As JET allows use of JSP written code, the lists will be traversed, importing the correct references to the state machine classes and creating a new instance of each state machine in *Start.java's onCreate()* method. The code is given below.

```
List imports = (List)i.next();
List stateMachines = (List)i.next();
```

First, the iterator (i) is traversed as to create a list of imports and state machines.

```
<% for (Object o : imports) { %>
import <%= (String)o%>;
<% }; %>
<% for (Object o : stateMachines) { %>
new <%= (String)o %>(scheduler);
<% } %>
```

The remaining code in the start class template is written as to mirror the *Start.java* class described in section 5.2.2.

6.4 The Java for Android Generator

This section will give a description of the core methods of the main generator class, *JavaForAndroidGenerator.java*. This class is developed as to cover all required adaptations found in chapter 5. The XML files and templates described above will come into use in the generator.

6.4.1 The *generateComponents* method

generateComponent is the main method of *JavaForAndroidGenerator.java*. This method builds the whole resulting Java project by making calls to other parts of the generator and using *IGenerator2* [17] variables and methods. The source code of the method is explained below.

```
IJavaProject newProject = CommonGenerator.generateCommonParts
(components, targetProject, monitor,
PlatformIdentifiers.JAVA_STANDARD);
IFolder newProjectSrcFolder = newProject.getProject().getFolder("src");
IClasspathEntry[] entries = newProject.getRawClasspath();
```

First off, *generateComponent* creates a new *IJavaProject* and adds objects for the source folder of the new project and its classpath. All objects and the *generateCommonParts* method, which sets up a Java project for the given *JAVA_STANDARD* platform, are from *IGenerator2*.

```
ResourceHelper.writeFileToProjectRoot(newProject.getProject(),
ANDROID_MANIFEST_FILENAME, AndroidManifestGenerator.
generateManifest(components.iterator().next().
getNearestPackage().getName()), new NullProgressMonitor());
generateAndroidLaunchClasses(components, newProjectSrcFolder, monitor, newProject);
```

Next, the Android Manifest file is generated from the above described class and written to the root of the project. All template defined classes are added to the project using the *generateAndroidLaunchClasses* method, see 6.4.2.

```
IResource runtimeFolder = newProject.getProject().findMember(
"/src/no/ntnu/item/arctis/runtime/AbstractRuntime.java");
runtimeFolder.delete(false, null);
entries = JavaResourceHelper.removeEntry(entries, JavaRuntime.
getDefaultJREContainerEntry());
IClasspathEntry androidEntry = JavaCore.newContainerEntry(
new Path(ANDROID_CONTAINER_PATH));
ResourceHelper.copyFile(Activator.getDefault(), new Path(
ArctisCodeGenerationConstants.PATH_TO_ABSTRACT_RUNTIME), newProject.
getProject(), new Path(ArctisCodeGenerationConstants.
PATH_TO_ABSTRACT_RUNTIME));
entries = JavaResourceHelper.addToEntry(entries, arctisRuntimePath);
newProject.setRawClasspath(entries, new NullProgressMonitor());
```

Generation for the *JAVA_STANDARD* platform created a runtime class and a Java Runtime Environment (JRE) container which is not suitable for Android. These are removed and replaced with the Android container and an updated runtime class.

```
ResourceHelper.addProjectNature(newProject.getProject(), ANDROID_NATURE);
```

To have the project behave as an Android project, adding features such as designing layouts and adding to the manifest from the Eclipse editor, the Android *projectNature* is added to the *IJavaProject*.

```
IPath layoutPath = new Path("res").append("layout");
IPath valuesPath = new Path("res").append("values");

ResourceHelper.writeFile(newProject.getProject(), valuesPath,
ANDROID_STRING_VALUES_FILNENAME,
AndroidStringsGenerator.generateLayout(), new NullProgressMonitor());
ResourceHelper.writeFile(newProject.getProject(),
layoutPath, ANDROID_MAIN_LAYOUT_FILENAME,
AndroidLayoutGenerator.generateLayout(), new NullProgressMonitor());
ResourceHelper.copyFile((Plugin)Activator.getDefault(),
new Path("icons/ntnu.png"),
newProject.getProject(), new Path("res/drawable/ntnu.png"));
ResourceHelper.copyFile((Plugin)Activator.getDefault(),
new Path("icons/icon.png"),
newProject.getProject(), new Path("res/drawable/icon.png"));

JavaResourceHelper.closeAndReOpenProject(newProject, monitor);
```

Finally, the file system has to match the one of an Android project. The file system is constructed using *IGenerator2* [17] and files are created using the above described XML generators and copies of image files. This creates an Android file system containing all necessary files.

6.4.2 The *generateAndroidLaunchClasses* Method

The second method of importance in *JavaForAndroidGenerator.java* is *generateAndroidLaunchClasses*. This method will generate classes for the templates described in 6.3 and put these into the *src* folder of the newly created *IJavaProject*. The templates makes use of an argument object which have to contain a list of imports, state machines and the package name. Hence, in the *generateAndroidLaunchClasses* method, an object named *templateArgs* is built containing the necessary component data. The argument object will be passed along with calls for generation of templates.

In this section there will be one example on how classes are generated from templates and added to the file system of the generated project. The creation and addition of the *Start.java* class will be used in the example.

```
JavaForAndroidStartGenerator startTemplate = new
JavaForAndroidStartGenerator();
```

First, an instance of the template created generator class is instantiated.

```
String startCode = startTemplate.generate(templateArgs);
```

Now, generation of *Start.java* code is done by calling the *generate(Object argument* method on the template class. The argument passed along is the *templateArgs* object referenced above. The return object will be a string containing the *Start.java* class.

```
ResourceHelper.writeFile(targetFolder, new Path(path),
GenerationConstantsAndroid.START_CLASS_FOR_ANDROID_NAME,
startCode, monitor);
```

The generated class will now be written into the project, using the *IGenerator2* method *writeFile*, which writes a file with a given name to a given path and a given source code. In this case, the path will be the default package of the project and the code is the string object *startCode*. The target name is resolved from a class containing generation constants for the project, having the class name for this instance being *Start.java*.

By now, the *Start.java* class will be part of the generated project and will behave dynamically in accordance to state machines.

Chapter 7

Building Block Library

7.1 Commonalities

All building blocks developed in this work shares a common part. That is the creation of an Android intent used for starting the Android services and a context for making the service call. The intent is created with a reference to the custom or default Android service and parameters describing the purpose of the building block. The common setup code are given below.

```
public android.content.Context context;
public android.content.Intent IntentName;
context = (Context)AbstractRuntime.getRuntime(this).getContext("Android");
IntentName = new Intent();
```

First off, all classes declear an Android context and intent. The Android context is retrieved from the scheduler. This is the context added at startup of the Android application, a result of the discussion in 4.3.

Setup of the services are done in a different manner for thread dependent and non-thread dependent building blocks, see section 4.2.

7.1.1 Logic for Restarting a Service

By restart of the services we mean how the system re initiates a service after it has been triggered. E.g when using a proximity alert, the system has to be able to restart tracking of a location after the alert has been triggered. Same goes for resetting of alarms and listening for incoming phone calls.

This work does not focus on user input, resulting in the locations and alarms that are to be monitored being coded in our examples. However, the behavioral design of the building blocks are done with concern on resetting mechanisms.

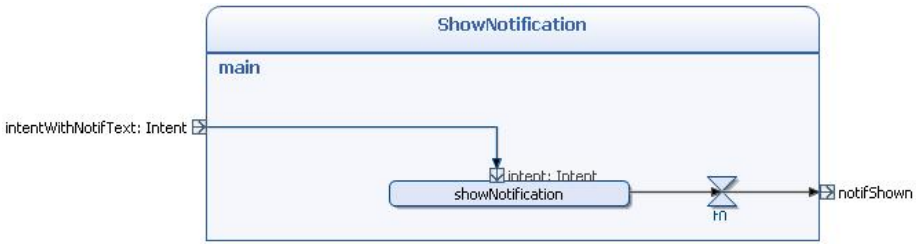

Figure 7.1: ShowNotification behavoir

In cases where resetting of the service is necessary, a timer is usually used for simulation. The timer will hold for a predefined time before re initiating the service. This works well for demonstration purposes.

7.2 ShowNotification

Figure 7.1 depicts the behavior of the *ShowNotification* building block developed in this work. This building block has a simple function, namely to display a notification on the Android device. The input of the block is named *intentWithNotifText* and as the name suggests, the type of input is an Android intent. This intent should always contain headline text, notification title, notification text and the a name of the notification icon which is to be used. Creation of an example intent is given below. The intent is fed to the building blocks only method, *showNotification*. After the method call is done, the block terminates via the *notifShown* pin.

```
public android.content.Intent createIntentForNoticationManager() {
Intent notificationIntent = new Intent();
notificationIntent.putExtra("HeadlineText", "This is the headline text!");
notificationIntent.putExtra("NotificationTitle", "This is the
notification title!");
notificationIntent.putExtra("NotificationText", "This is the text
of the notification!");
notificationIntent.putExtra("NotificationIcon", "picturename");
return notificationIntent;
}
```

The intent created above is fed to the notification manager's *showNotification* method where, in addition to the default setup, other parameters are set up. The whole method is given below.

```
public void showNotification(android.content.Intent intent) {
Notification notification = new Notification(0, intent.getExtras().getString
("HeadlineText"), System.currentTimeMillis());
notification.flags = Notification.FLAG_AUTO_CANCEL;
context = (Context)AbstractRuntime.getRuntime(this).getContext("Android");
```

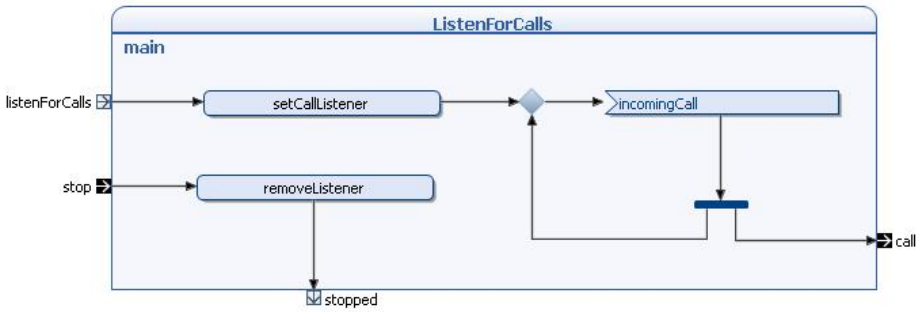

Figure 7.2: ListenForCalls behavior

```
intent.putExtra("Notification", notification);
intent.setAction("android.intent.action.TESTSERVICEv2.0");
intent.putExtra("ManagerName", "NotificationManager");
context.startService(intent);
}
```

ShowNotification is a thread dependent building block, resulting in use of our custom service class during service setup. A notification object is created using the headline text of the input intent. Then a flag is set as to make sure the notification icon disappears from the status bar of the Android device when clicked. The notification is then added to the input intent. The intent is then set up as to start our custom service and the notification manager. Finally, *startService* is called using the modified input intent as parameter.

The timer, *t0*, has no function, but is required in the current version of Arctis for building blocks which only contain a single method as in this case.

7.3 ListenForCalls

The *ListenForCalls* building block, depicted in figure 7.2, has no object type for its starting event, *listenForCalls*. The phone states such as *ringing, off-hook and idle* does not need any user input when being set up. However, use of the telephony manager is thread dependent, see section 4.2, hence will setup of the service go through our custom service class.

After being started, this building block enter the *setCallListener* method. The flow then continues to a merge node before it enters an accept signal action, namely *incomingCall*. When the signal arrives the flow forks to an output streaming pin, *call*, and the merge node. The output informs the environment of the recent happening and the merge node resets the listening of calls.

This building block does not have a simulated resetting of the service as no new input is needed for this process.

The *setCallListener* method is given below. Because the method has no input arguments, all that is done is the setup towards our custom service class.

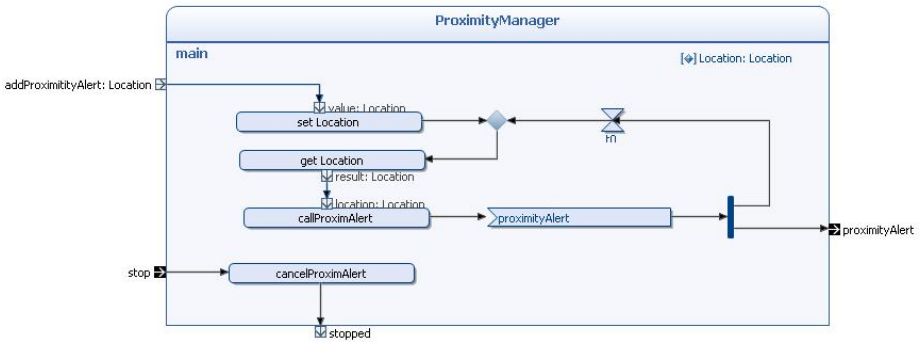

Figure 7.3: ProximityManager behavior

The intent is equipped with information on what manager is wants to use and which telephone state it wants to listen to.

```
public void setCallListener() {
context = (Context)AbstractRuntime.getRuntime(this).getContext("Android");
TelephonyIntent = new Intent();
TelephonyIntent.setAction("android.intent.action.TESTSERVICEv2.0");
TelephonyIntent.putExtra("ManagerName", "TelephonyManager");
TelephonyIntent.putExtra("TypeOfService", "OnThePhone");
context.startService(TelephonyIntent);
}
```

ListenForCalls has a input streaming pin named *stop*, which is used to stop the service. After calling *removeListener*, the block terminates via *stopped*.

7.4 ProximityManager

Figure 7.3 depicts the proximity manager building block. As the name illustrates, this building block will setup a proximity alert for a given location. It has one variable which is an Android location object. This building block can setup the desired service directly (non-thread dependent).

The building block starts via the *addProximityAlert* input pin, with the location object as parameter. The location is stored for the building block using *set Location*. Then the flow enters a merge node, after which the location variable is fetched in *get Location* and fed into the building blocks most important method, namely *callProximAlert*. This method calls upon our custom service which sets up the proximity alert for the given location.

After setting up the proximity alert the building block now accepts input from the system. The accepted signal is named *proximityAlert*. When the signal arrives, the control flow enters a fork node, which initiates a streaming output *proximityAlert* indicating that the alert has been triggered. The other control

flow from the fork node enters a timer, *t0*, used for representing the resetting of the proximity alert as explained above in 7.1.1.

The building block also have a streaming input pin called *stop* which will call the *cancelProximAlert* method, canceling the service.

The location object starting the proximity manager is an Android location. The location object is created with a name of choice along with a latitude and a longitude.

The *callProximAlert* method sets up the default Android location service using information from the input location object. The method code is given below.

```
public void callProximAlert(Location location) {
context = (Context)AbstractRuntime.getRuntime(this).getContext("Android");
lMN = (LocationManager)context.getSystemService(context.
LOCATION_SERVICE);
LocationIntent = new Intent();
LocationIntent.setClassName(context, context.getPackageName()
+".ProximAlertReceiver");
contentIntentLocation = PendingIntent.getBroadcast(
context,0,LocationIntent,0);
lMN.addProximityAlert(location.getLatitude(),
location.getLongitude(), 1000, 333333333, contentIntentLocation); }}
```

The location manager is instantiated using the *getSystemService* method called via the context retrieved from the start activity. Then, the location intent is set up with references to the context and the receiver class. The reference to the receiver class can be fixed at context package name plus the *ProximAlertReceiver* extension due to this file always being generated at this location by our code generator. A pending intent is setup and used for adding the proximity alert along with parameters for latitude and longitude from the input location object.

The proximity alert demands that an expiration time and a radius is set. The expiration time determines how long, in seconds, the proximity alert is going to be valid. For a consistent proximity alert, it is recommended to use a large number, as we have done. The radius determines how close, in meters, one have to be to the target location before the alert is triggered. This is set to 1000 meters in our examples, as the emulator currently has a location bug, but could of course be altered. A different idea could be to have this radius as part of an input object of the building block, leaving the choice of radius to the developer or user for each proximity alert.

The *cancelProximAlert* is used for cancelling the service and is done by removing the proximity alert for the given pending intent.

```
lMN.removeProximityAlert(contentIntentLocation);
```

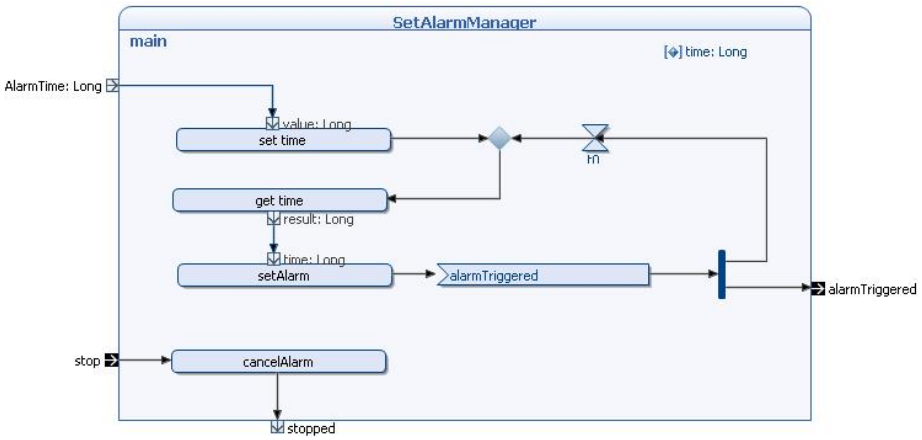

Figure 7.4: SetAlarmManager behavior

7.5 SetAlarmManager

The *SetAlarmManager* building block's, figure 7.4, task is to set an alarm on the phone. The start pin of this building block is hence named *AlarmTime*, with object type *long*. This object variable is first set in *set time* before it enters a merge node. After the merge node, the flow enters *get time* which retrieves the *AlarmTime* object and feeds into the *setAlarm* method. This flow will now accept an input signal, namely *alarmTriggered*. When it arrives, the flow is forked, outputting *alarmTriggered* to the environment and entering a timer.

The timer *t0* is used here, as in the case of the location manager, for simulating resetting of the alarm.

SetAlarmManager has a streaming input pin, *stop*, which will call the *cancelAlarm* method, before the building block terminates via *stopped*.

This building block is non-thread dependent. The *setAlarm* method sets up the alarm in the same manner as the proximity manager added the proximity alert above. The difference being the alarm manager being used instead of the location manager.

```
public void setAlarm(java.lang.Long time) {
context = (Context)AbstractRuntime.getRuntime(this).getContext("Android");
aMN = (AlarmManager)context.getSystemService(context.ALARM_SERVICE);
AlarmIntent = new Intent();
AlarmIntent.setClassName(context, context.getPackageName()+".AlarmReceiver");
contentIntentAlarm = PendingIntent.getBroadcast(context,0, AlarmIntent, 0);
aMN.set(AlarmManager.RTC_WAKEUP, alarmTime, contentIntentAlarm);
}
```

7.6 Cancelling a Service

Cancellation of the services are done in the building blocks according to thread dependence and other circumstances. E.g will the *ShowNotification* building block need no cancellation mechanism as this is done by setting flags upon creation of the notification. For thread dependent services, the logic for canceling a service is in the custom service class. All that needs to be done in the building block's method for canceling the service is to add an *Extra* to the intent, stating that the service is to be stopped. By calling *startService* with the new extra will stop the service. The standard code for stopping the service is given below.

```
IntentName.putExtra("Stop", true);
context.startService(IntentName);
```

For non-thread dependent classes the service is stopped by canceling, stopping or removing the pending intent for a given Android manager, see example code below.

```
managerInstance.cancel(pendingIntent);
```

Chapter 8

Example Application - *TwitterFromAndrid*

Twitter is a micro blog service which allows people to send status messages to a network of followers [30, 26, 32]. Followers are people which have made an active choice of following your status updates. A twitter user may update the status using the twitter website or a mobile device of choice. The updates are mostly done manually. Some people have a high frequency of twittering, wanting to share their every move and day-to-day life. Automating the twitter process could be of interest for these "'eager"' twitters. Also, use of automatic twitter updates could be useful in a school or work setting, as one could use it to register which persons are *in office* or perhaps *out to lunch*.

We will develop an application which automatically creates twitter updates as a reaction of interface events. The events could the phone ringing or use of a location service. The application will be running on Android and developed using the *Java SE for Android* code generator.

8.1 Specification

The application will be completed to a degree in which updates will be done on a twitter account in accordance to events on the Android device. The events the application handles are telephone calls, proximity alerts and alarms. When one of these events are triggered, twitter should be updated with the recent event information and the Android device should receive a notification confirming the twitter update. All building blocks described in chapter 7 will be used. The default Android services used is hence the alarm service, the location service, the telephony service and notification service.

As the work done in this thesis does not focus on user interface the application will have one user interface, the common screen layout, see 4.2.6.

8.2 Arctis Behavior for *TwitterFromAndroid*

Figure 8.1 depicts the behavioral design of the *TwitterFromAndroid* application. On initialization, the control flow enters a fork node, forking the flow in several directions. Two flows initiates method calls, namely *generateAlarmTime* and *generateLocation*. These method calls are responsible for generation of simulated user input. An alarm time as well as a location is generated. The resulting *Long* time object and *Location* object is fed to its respective building blocks, namely *SetAlarmManager* (see section 7.5) and *ProximityManager* (see section 7.4). One flow enters the *ListenForCalls* (see section 7.3) building block. As no object is required upon startup for this building block, only a control flow is needed.

A *StopListener* is also enabled. This listener continually listens for a stop signal, indicating that the user have pressed the *Stop Service* button. When the stop service signal arrives, *StopListener* will terminate via *stoppedReceived*. The control flow is forked to all managers' *stop* input, effectively stopping the services started upon initiation of these managers.

Each system response from either of our services result in a method call. Both *generateAlarmTwitterMsg, generateLocationTwitterMsg* and *generateCallTwitterMsg* returns a *TwitterStatusUpdate* object, describing the recent event. The *TwitterStatusUpdate* object contain username, password and a update message. This is fed into a *SetTwitterStatus* building block, see [24], updating the twitter status for the given user with the given message. The design makes use of three instances of *SetTwitterStatus*. These are necessary as each event is associated with a different twitter message.

The *SetTwitterStatus* building block results in either *ok* or *failed*. If *ok*, the control flow enters a method responsible for outputting an Android intent. This intent should be in the form presented in 7.2 as the object flow enters a merge node leading to a *ShowNotification* building block. A fail means the update did not complete and the control flow enters a flow final.

ShowNotification displays the information, contained in the intent, on the Android device and terminates via *notifShown*. The flow then enters a flow final.

As both *SetAlarmManager, ProximityManager* and *ListenForCalls* have internal behavior which resets the service after an occurring event, there will possibly be several outputs from *alarmTriggered, proximityAlert* and *call* resulting in the *TwitterFromAndroid* system generating several twitter status updates and notifications. These are produced until the system receives a stop signal, having *StopListener* output *stoppedReceived*.

8.3 Source Code for System Entity

Figure 8.1 displays several method calls. The content of some of these are given below, presented top-down in accordance to the system's behavioral model.

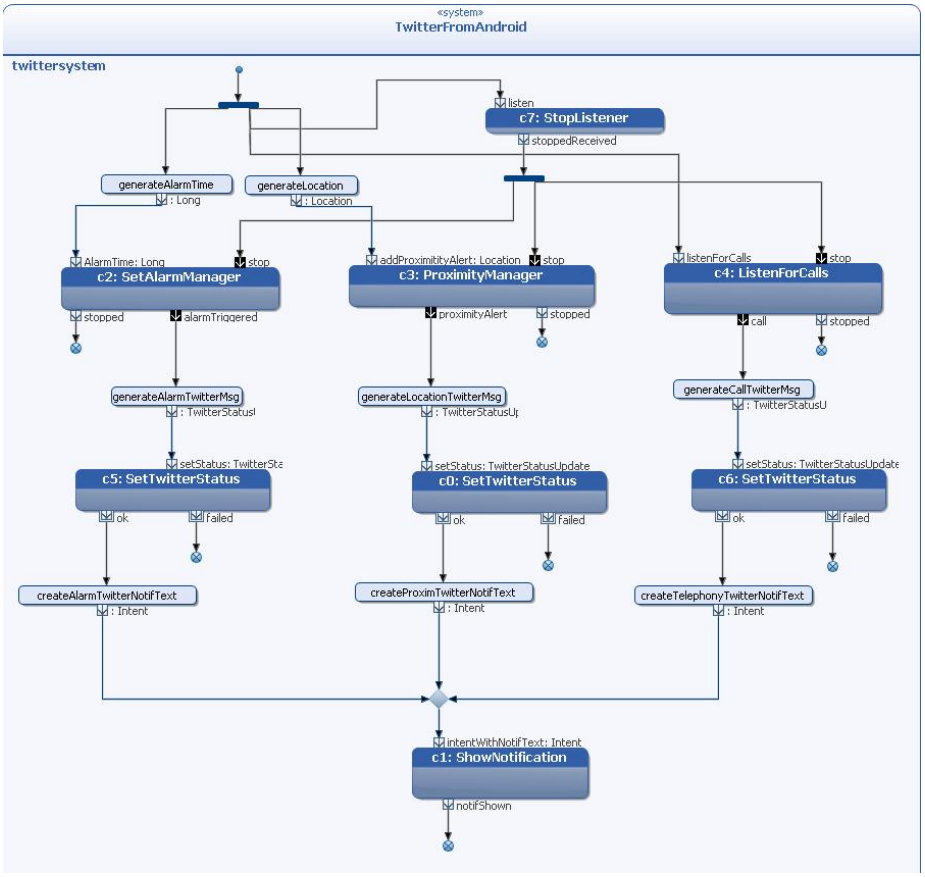

Figure 8.1: *TwitterFromAndroid* behavoir

```
public java.lang.Long generateAlarmTime() {
Long time = System.currentTimeMillis() + 10000;
return time;
}
public android.location.Location generateLocation() {
Location l = new Location("At home");
l.setLatitude(63.416014);
l.setLongitude(10.407400);
return l;
}
```

The two first methods generates input objects for our services. The alarm time is set to ten seconds in the future and is a *java.lang.long* object. The location is an Android location object which has a name, latitude and longitude. The location given here is close to the *NTNU* campus in Trondheim.

```
public TwitterStatusUpdate generateCallTwitterMsg() {
return new TwitterStatusUpdate("testnordmann", "androidntnu",
"On the phone!");
}
public TwitterStatusUpdate generateLocationTwitterMsg() {
return new TwitterStatusUpdate("testnordmann", "androidntnu",
"Arrived at school!");
}
public TwitterStatusUpdate generateAlarmTwitterMsg() {
return new TwitterStatusUpdate("testnordmann", "androidntnu",
"My alarm just went off!");
}
```

Next, generation of twitter messages is done. These include the user name, password (see 8.5 and update message. As described above, the twitter message is alternated for the different kinds of events.

```
public android.content.Intent createAlarmTwitterNotifText() {
Intent notificationIntent = new Intent();
notificationIntent.putExtra("HeadlineText", "New twitter update!");
notificationIntent.putExtra("NotificationTitle", "Alarm twitter!");
notificationIntent.putExtra("NotificationText",
"Your twitter has been updated with a recent alarm event!");
notificationIntent.putExtra("NotificationIcon", "ntnu");
return notificationIntent;
}
```

All creation of intents for notifications are done as above for *createAlarmTwitterNotifText*. The only difference from the other methods, *createProximTwitterNotifText* and *createTelephonyTwitterNotifText* are the notification title and notification text. These methods follows the default setup described in 7.2.

8.4 Manifest.xml

TwitterFromAndroid's manifest file includes some specific permissions along with descriptions of all necessary services, receivers and activities. All components are declared inside an application tag as described in 2.9.

```
<uses-permission android:name="android.permission.INTERNET" />
<uses-permission android:name="android.permission.ACCESS_FINE_LOCATION" />
```

Declares the permissions necessary for the example application. To enable the twitter updates to be sent, an Internet permission is required. The access fine location permission is required as to be able to retrieve location updates.

```
<activity android:name=".Start" android:label="@string/app_name">
<intent-filter>
<action android:name="android.intent.action.MAIN" />
<category android:name="android.intent.category.LAUNCHER" />
</intent-filter>
</activity>
<service class="AndroidServiceForArctisBuildingBlocks" android:name="
.AndroidServiceForArctisBuildingBlocks">
<intent-filter>
  <action android:value="no.ntnu.item.ArcDroid.TESTSERVICEv2.0"
  android:name="android.intent.action.TESTSERVICEv2.0" />
</intent-filter>
</service>
```

Declares the default start activity and service, used for every Android application designed in Arctis.

```
<receiver android:name=".ProximAlertReceiver" android:label="@string/proxim_name" />
<receiver android:name=".AlarmReceiver" android:label="@string/alarm_name" />
<receiver android:name=".TelephonyReceiver" android:label="@string/telephony_name" />
<activity android:name=".NotifReceiver" android:label="@string/notif_name" />
```

Finally, the receiver components for *ProximityManager, SetAlarmManager* and *ListenForCalls* are added, as well as the activity that should be started when a notification is selected, *NotifReceiver*.

8.5 The Twitter Account

To enable twitter updates to be done a twitter account is needed. Also, account information is needed as to be able to test the application by following the twitter updates done.

A twitter account was created with the following information.

Figure 8.2: *TwitterFromAndroid* part 1

Name: "'Ola Testnordmann"'

Username: "'testnordmann"'

Password: "'androidntnu"'

The user name and password is needed when wanting to update the twitter status. The name and user name can be used when searching for the twitter account.

8.6 Demonstration of *TwitterFromAndroid*

This section will present a demonstration on how our application works on the Android emulator. A series of events from numerous sources are simulated on the Android emulator [7] as to give a realistic view on how the application operates. Figure 8.2 depicts the first step of the demonstration. Our application is opened and started by pressing the *Start service!* button. An on-screen

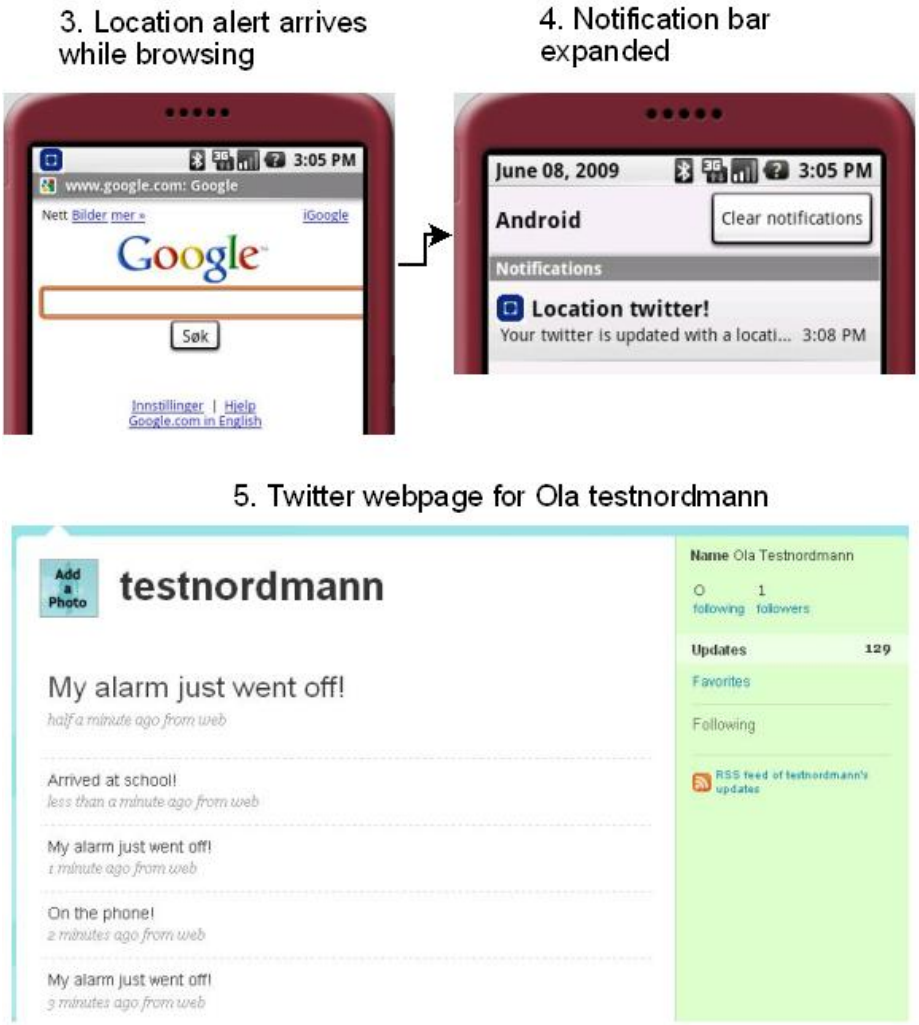

Figure 8.3: *TwitterFromAndroid* part 2

message confirms the startup. Shortly after startup, as described above in 8.3, an alarm goes off (1. in figure 8.2) and a notification is added. As can be seen, the title of this notification is *Alarm Twitter!* indicating the *twitter* account being updated with an alarm message.

While browsing the extended notification bar an incoming call arrives. The call is answered (2. in figure 8.2) and a new twitter update is done. A new twitter update notification arrives in the status bar as the call is in progress. Figure 8.3 depicts the second part of the demonstration. The call has been terminated and the user of the Android device is browsing the Internet while walking to school. When approaching the school, the proximity alert is triggered. A twitter update is done and a notification has arrived (3. in figure 8.3). Expanding the status bar will now show a notification named *Location twitter!* (4. in figure 8.3).

The twitter update messages sent by our application is depicted in 5. of figure 8.3. In accordance to our demonstration, an *My alarm just went off!* message was posted followed by an *On the phone!* message. As we have designed the alarm building block with a repeating alarm for demonstration purposes, two more *My alarm just went off!* messages was posted. One before and one after the location message, *Arrived at school!*. (The arrival of the second and third alarm notification was left out in the figures.)

As demonstrated will the twitter updates, notifications and services started by the application continue to arrive and run regardless of what the user is currently doing on the Android device. This will continue until *Stop service!* is pressed on the main screen of the application.

8.7 Possible Future Work on *TwitterFromAndroid*

For the *TwitterFromAndroid* application to be ready for release on the Android market [1], there are two areas which need attention. One is user interfaces, which has been out of focus in this work. The application will benefit from interfaces allowing the user to choose which type of twitter messages that shall be updated and also what the content of these messages should be.

The changes described above have an impact on the behavioral model of the application, depicted in 8.1. First of all, the fixed alarm time and location used in our demonstration will need to be replaced with logic for receiving these from the user. This also goes for the generation of twitter messages.

Lastly, the logic for resetting services has to be resolved for the *SetAlarmManager* and the *ProximityManager*. The resetting is currently simulated using a timer, as described in section 7.1.1.

Chapter 9

Conclusion and Future Work

9.1 Conclusion

Our motivation was to find a method for developing applications for the Android platform using the Arctis tool. Understanding the life cycle of Android components, adjusting the existing Java Standard Edition (SE) code generator to work with Android and providing building blocks for use with Android services were central parts of this work. Also, developing an Android application using the generator and building blocks were of focus.

The Android platform has been described, with focus on details and components most essential to our work. An example application was developed and used as part of the description. This application was developed in Eclipse using the Android development kit, but without use of the Arctis tool.

Through a discussion, an architecture describing how Android applications can be developed using Arctis was proposed. The major discussion issue was how to resolve problems regarding the runtime scheduler running in a separate thread from the Android components. The proposed architecture is best suited for development of applications which use Arctis building blocks currently existing in our *Android building block library*. There are drawbacks to the proposed architecture regarding applications using newly created building blocks which access thread dependent Android services. Solutions to the problems are discussed and proposed. Building blocks accessing default Android services not demanding an Android thread can be designed without alternation of our code generator.

Our method for adjusting the code generator was to first design some building blocks composing an Android application in Arctis and deploy it using the existing Java SE generator. The building blocks were designed according to our architecture. The adaptations necessary for enabling the generated application to run on an Android device were explored and noted. The adaption process

along with the architectural discussion made the foundation for development of our new, *Java SE for Android* code generator.

This work has resulted in the Arctis editor being equipped with an deployment option for Android applications called *Java SE for Android*. A library consisting of building blocks for use with development of Android applications is added. We have successfully developed and deployed two applications for the Android platform using Arctis and our *Android for Java SE* code generator. *HelloLocationWorld* is a simple location tracking application which was used in describing the Android platform and discovering adjustments. *TwitterFromAndroid* is a more comprehensive application which uses our library's building blocks and the modeling abilities provided by Arctis in creating a *twitter* application for the Android device. *twitter* updates for a given account will automatically be done as a result of events on the Android device.

9.2 Future Work

The final architecture used in this work can be reviewed based on the issues identified in this thesis. Alternative architectures which should be looked into includes having the scheduler run as an Android service, hence possibly avoiding the thread problems. If successful, this will yield an easier solution in regards to designing new building blocks.

The *TwitterFromAndroid* application could be finalized for the Android market. Adding a possibility for designing Android applications with several Android activities and hence user interfaces are of interest as this should be part of any complete Android application including *TwitterFromAndroid*.

Systems with multiple state machines and how this affects our code generator is also of interest.

Bibliography

[1] Android. Android market website. http://www.android.com/market/, 2009. online: June 2009.

[2] Android Developer. Android application fundamentals. http://developer.android.com/guide/topics/fundamentals.html, 2009. online: June 2009.

[3] Android Developer. Android default system services. http://developer.android.com/reference/android/content/Context.html#getSystemService(java.lang.String), 2009. online: June 2009.

[4] Android Developer. Android sdk download website. http://developer.android.com/sdk/1.5_r1/index.html, 2009. online: June 2009.

[5] Android Developer. public class Activity. http://developer.android.com/reference/android/app/Activity.html, 2009. online: June 2009.

[6] Android Developer. Security and permissions. http://developer.android.com/guide/topics/security/security.html, 2009. online: June 2009.

[7] Android Developer. Using the emulator console. http://developer.android.com/guide/developing/tools/emulator.html, 2009. online: June 2009.

[8] Android Developer. What is Android website. http://developer.android.com/guide/basics/what-is-android.html, 2009. online: June 2009.

[9] Google Developers. Google i/o 2008 - dalvik virtual machine internals. http://www.youtube.com/watch?v=ptjedOZEXPM, 2008. online: June 2009.

[10] P. Herrmann F.A. Kraemer and R. Bræk. Aligning uml 2.0 state machines and temporal logic for the efficient execution of services. In *The 8th Int. Symp. on Distributed Objects and Applications (DOA)*, pages 1613–1632. Springer-Verlag, 2006.

[11] R. Bræk F.A Kraemer and P.Herrmann. *Synthesizing Components with Sessions from Collaboration-Oriented Service Specifications. SDL*. Springer-Verlag, 2007.

[12] Eclipse Foundation. The eclipse modeling framework (emf) overview. http://help.eclipse.org/ganymede/index.jsp?topic=/org.eclipse.emf.doc/references/overview/E, 2009. online: June 2009.

[13] Eclipse Foundation. Java development tools. http://www.eclipse.org/jdt/, 2009. online: June 2009.

[14] Eclipse Foundation. Java development user guide. http://help.eclipse.org/ganymede/index.jsp?nav=/1, 2009. online: June 2009.

[15] Stephan Haugsrud. A mobile treasure hunt as an example for collaborative service specifications, 2008.

[16] Nina Heitmann. Towards modeling of data in uml activities with the space method. Master's thesis, NTNU, 2008.

[17] NTNU ITEM. Contributing a code generator. http://arctis.item.ntnu.no/contributing_a_code_generator, 2006. online: June 2009.

[18] NTNU ITEM. Arctis web page. http://www.arctis.item.ntnu.no/, 2009. online: June 2009.

[19] F.A. Kraemer and P. Herrmann. Service specification by composition of collaborations - an example. In *WI-IAT Workshops*, pages 129–133, Hong Kong, 2006. IEEE/WIC/ACM Int. Conf. on Web Intelligence and Intelligent Agent Techonology.

[20] F.A. Kraemer and P. Herrmann. Formalizing collaboration-oriented service specifications using temporal logic. In *Networking and Electronic Commerce Research Conference*. NAEC, 2007.

[21] F.A. Kraemer and P.Herrmann. Transforming collaborative service specifications into efficiently executable state machines. In *The 6th Int. Workshop on Graph Transformation and Visual Modeling Techniques*, 2007.

[22] Frank Alexander Kraemer. Rapid Service Development for Service Frame. Master's thesis, University of Stuttgart, 2003.

[23] Frank Alexander Kraemer. Arctis and ramses: Tool suites for rapid service engineering. *Norges teknisk-naturvitenskapelige universitet*, 2007.

[24] ITEM NTNU Kraemer. Updating twitter status. http://arctis.item.ntnu.no/examples/updating_twitter_status, 2009. online: June 2009.

[25] Bemnet Yesfaye Merha. Code generation for executable state machines on embedded java devices, 2008.

[26] Erik Nude. Introduksjon til twitter. http://www.eriknude.com/introduksjon-til-twitter-del-1-eriknudecom, 2008. online: June 2009.

[27] OpenHandsetAlliance. Open handset alliance. http://www.openhandsetalliance.com, 2009. online:June 2009.

[28] Eclipse Organization. Jet tutorial website. http://www.eclipse.org/articles/Article-JET/jet_tutorial1.html, 2004. online: June 2009.

[29] A.K Støyle. Service engineering environment for amigos. Master's thesis, NTNU, 2004.

[30] Twitter. Twitter homepage. http://twitter.com, 2009. online: June 2009.

[31] Wikipedia. Dalvik virtual machine. http://en.wikipedia.org/wiki/Dalvik_virtual_machine, 2009. online: June 2009.

[32] Wikipedia.org. Twitter. http://en.wikipedia.org/wiki/Twitter, 2009. online: June 2009.