The GooglePos basic service (version 060911)

Introduction

The GooglePos basic service is a model that can be compiled and executed. The service performs a simple positioning service producing a file that can be read by GoogleEarth.

Contents

The following items are needed and should be available to you:

This description

1. GooglePos.pdf
   This very description of the GooglePos basic service

The modeling files

2. Google positioner.emx
   This is the UML 2 model with appropriate annotations
3. GooglePos-1.sd2
   This is a SeDi sequence diagram of a partial execution of GooglePos
4. SMSPorts.emx
   The UML package defining the SMSPorts. The implementation of this is given in SMSMediators.jar (see below)

The RSM plugins

5. JavaFrameProfile (inside *JavaFrameTransformation_nnn.zip*)
   The UML-profile for JavaFrame. This is needed to be able to interpret the stereotypes introduced in the UML model.
6. JavaFrameTransformation (inside *JavaFrameTransformation_nnn.zip*)
   The UML to JavaFrame compiler
7. SeDi plugin (no.uio.ifi.uml.sedi_1.2.3.jar)
   The sequence diagram editor that can read the old *.sd2 files. This plugin is only for being able to see sequence diagrams from last semester.

The execution files

8. JavaFrame.jar
   The JavaFrame platform for execution of UML 2 – almost a runtime system
9. SMSMediators.jar
   The implementation of the SMSPorts.emx. The implementation is handmade and not a transformation from the UML model.
10. no.uio.ifi.pats.client.jar
    The low-level implementation of PATS – used by SMSMediators.jar. Source is not provided

JFTrace

11. JFTrace.jar
    The stand-alone tracing program accompanying JavaFrame executions
12. JFTraceUG.htm
    The JFTrace User Guide
13. fileopen.jpg
    Picture used by JFTraceUG.htm – must be in the same directory as JFTraceUG
14. filterform.jpg
    Picture used by JFTraceUG.htm – must be in the same directory as JFTraceUG

Setting up eclipse for painless upgrades

This subsection has been copied from http://simplygenius.com/geekblog/2005/04/08/1112982083849.html by Matthew Conway. Another site explaining the same procedures can be found at http://www.javalobby.org/java/forums/t18678.html by R. J. Lorimer.

Creating an extension location works under both the 3.0.x and 3.1.x releases of eclipse, under both linux and windows (and I assume all others). The steps to accomplish this under linux are as follows:

1. As root, I install eclipse to /opt/eclipse
2. As myuser, I create the directories
   - ~/eclipse-config
   - ~/eclipse-config/eclipse
   - ~/eclipse-config/eclipse/features
   - ~/eclipse-config/eclipse/plugins
   - ~/eclipse-config/workspace
3. I edit ~/eclipse-config/eclipse/.eclipseextension to contain:

```
4.          name=My Eclipse Configuration
5.          id=my.eclipse.configuration
6.          version=1.0.0
```

7. As myuser I install plugins/feature to ~/eclipse-config/eclipse (manually or through eclipse update mechanism after step 6 completed)
8. As myuser, I start eclipse like "/opt/eclipse/eclipse -data /home/conway/eclipse-config/workspace -vmargs -Xmx512M"
9. In Help->Software Updates->Manage Configuration... I "Add an Extension Location" to ~/eclipse-config/eclipse. The setting for this ends up being stored in ~/.eclipse

The thing I really like about this method is that aside from installing the eclipse runtime as root, I never have to do anything else as root again - I can install all my plugins as myself in my home directory, and when it comes time to upgrade eclipse, I don't have to deal with moving my plugins to a new install - I simply install the new eclipse into it own directory, run it, perform step 6, and all my plugins are present (assuming they work across eclipse versions =)

Also, for large multi-feature/multi-plugin extensions like the eclipse WebTools Project, I'll create a new extension location just for that extension so that I can treat it as a single unit for enable/disable/upgrade.

The process under windows is identical except you don't have to worry about user permissions, and you need to choose windows paths that make sense to you. I use c:\devtools\eclipse for my eclipse install and c:\devtools\eclipse-config for all my configuration.

Getting started

The following is a step-by-step instruction for how to install the different software in the appropriate order.

All of the material is collected in one zip-file. You may import this zip-file into a temporary project of RSM. From there you should distribute the files according to the sequence of steps shown below.

1. Install the JavaFrame Profile

Make sure that *no.uio.ifi.javaframeprofile_xxx* folder is extracted from *JavaFrameTransformation_nnn.zip* into your Eclipse plugin folder.

2. Install the UML to JavaFrame compiler

Make sure that *no.uio.ifi.javaframetransformation_yyy* folder is extracted from *JavaFrameTransformation_nnn.zip* into your Eclipse plugin folder.

2b. Install the (old) SeDi plugin

Place no.uio.ifi.uml.sedi_1.2.3.jar in the Eclipse plugin-directory.

After having installed these plugins, you must restart the RSM – possibly with the "-clean" argument.

3. Import the SMSPorts model

Make a project called *SMSPorts* and import the *SMSPorts.emx* file into that.
Make sure that the JavaFrame profile is still available by doubleclicking the *SMSPorts.emx* file and see that JavaFrame profile is under the applied profiles.

4. Import the GooglePos model

Make a project called (e.g.) GooglePos and import the *Google positioner.emx* file into that.
Make sure that the *SMSPorts* package is properly imported by the GooglePos package. You should see the *SMSPorts* package mentioned in the pane called "Referenced models" in the main window for the *Google positioner.emx*.
Also Import the SeDi-file *GooglePos-1.sd2* into the GooglePos project.

5. Place the java jar files

We suggest that you store the java jar-files *JavaFrameIFI.jar, JFTrace.jar, SMSMediators.jar* and *no.uio.ifi.pats.client.jar* in a project in the workspace.
You can do that by just storing them in the appropriate folder in the file system and then perform a Refresh on the project, or you can import the files by selecting the project and performing *File/Import/File System* etc. If you have already imported these files from the overall zip-file onto a temporary project, you may just create the *JavaJars* project and drag the files from the folder JavaJars to the created project. Include also the documentation files for JFTrace.

6. Set up compilation target

We are now going to set up the target for the UML to JavaFrame compilation.

1. Create a Java project
2. Add the appropriate jars to the build path by
    a. Select java project in package navigator and choose *Project/Properites*
    b. Click *JavaBuildPath* in the menu on the left
    c. Select tab *Libraries*
    d. Click *Add Jars ...*
    e. Find the project with the java jar and select the three jars needed (JavaFrame, SMSMediators, no.uio.ifi.pats.client.jar)

7. Set up compilation configuration

We need to define a compilation configuration that will describe the source and target of a transformation.
Make sure you are in the Modeling perspective.

1. Go to: Modeling > Transform > Configure Transformations
2. Create a new UML2 to JavaFrame transformation and select the Java project as target.
3. Click Apply and Close.

8. Compile GooglePos

Right-click on the package with the name GooglePos. To reach this you will need to open the GooglePos.emx file to see the insides. We only want to transform the GooglePos package. The two other packages have implicit implementations.
Select *Transform* and choose your transformation configuration

Now you should have produced java files in the java project. We are ready to execute.

9. Setting up the execution of GooglePos

We are going to execute GooglePos with trace of the transitions. It is also possible to execute without trace and we shall come back to that below.
When we are going to use trace, this means running JFTrace, and it must be started first. Then the GooglePos java program will be executed and communicate with JFTrace.

To run JFTrace as well as GooglePos, we shall need ”run configurations”.

1. Run configuration for JFTrace
    a. Run / External Tools/External Tools ...
    b. Click New
    c. A dialog appears
        i. Select a proper name like *JFTrace*
        ii. Main tab: Location: ${system:ECLIPSE_HOME}/jre/bin/java.exe or some other path to the main java interpreter
        iii. Main tab: Working Directory: ${workspace_loc:/JavaJars} you get this string by Browsing the workspace
        iv. Main tab: Arguments: -jar JFTrace.jar make sure you have the minus before jar!
        v. Common tab: check Display in favorites menu External Tools and check Launch in background
    d. Click Apply and close
2. Run configuration for GooglePos
    a. Right-click on GposMain.java
    b. Run / Run ...
    c. The same dialog box appears. On the left side, select *Java Application* and click *New*
    d. Give the run configuration a good name, e.g. GooglePos
    e. Click tab Arguments and in the pane named Program Arguments, put your user name and then the following indicating communication with JFTrace *-remote localhost:54321*
    There should be a blank between the user name and the ”-remote localhost:54321”
    f. Click Apply

We now have the run configurations and we are ready to execute!

10. Executing GooglePos

As we mentioned above, JFTrace needs to start prior to the application.

1. Start JFTrace by selecting JFTrace run configuration from External Tools (a button or from Run/External Tools)
    a. A dialog appears. Choose File / Open input socket
    b. Type 54321 and click OK
    c. another dialog appears, but this shall have to wait a few seconds ...
2. Start GooglePos by Run / Run ... and select GooglePos
    a. Eventually a GUI dialog will appear.
3. Return to the JFTrace dialog and press Apply
    a. A trace window appears showing the transitions
    b. Bring the GooglePos dialog to the forefront – now we are ready to really run the system

11. Running GooglePos

Now GooglePos is executing. There is a GUI on the screen making it possible to input SMS signals without really sending an SMS, but in order to perform real positioning we shall need to provide a real static ID and to acquire such a real static ID, we need to send real SMS-es and look at the trace to find the static IDs.

1. Send the following SMS to 2034:
   Stud1 konto <user name> reg <SMSsender ident>
   a. <brukernavn> without the < and> and must be the same user name as given in the GooglePos run configuration.
   b. <SMSsender ident> can be anything that will identify this mobile phone.
2. Find the static ID in the trace and make a note of what it is.
3. In later executions, you can simulate this SMS sending by the GUI
   a. Click SMS
   b. Fill in: ”Stud1 konto <*user name*> reg <*SMSsender ident*>,2034, <*static ID*>” as parameters
   c. Click send.
4. Never send the same registration twice in the same execution. The program is not made to handle that well. (You may like to find out why)
5. Do not run the program more than a few minutes since positioning requests are done every minute.

Now GooglePos is running, but you see nothing of the results.

NB: We have included a hack such that the given model will only run for a small number of iterations – for about 5 minutes if no parameters are changed. This is to make sure that you use too much resources and not knowing. You will easily be able to find the hack when you understand the model.

12. Running GoogleEarth

To see what GooglePos has found out, you need to display the information through GoogleEarth. You may find all about GoogleEarth at http://earth.google.com/
Here is how you may display the information in GoogleEarth:

1. Start GoogleEarth
2. Add/Network Link ...
   Then a dialog appears
3. Give it an appropriate name – such as GooglePos
4. Click Browse and specify the produced .kml file. By default the file is named googlepos.kml and is situated in the project folder of the target java project.
5. Click the box Refresh Parameters
   a. Time-based refresh may be e.g. every minute for the testing period
   b. View-based refresh is not so important.
   c. Create the placemarks in Temporary Folder (you will later be asked if you want to move it to a more permanent place)
   d. Click OK

6. You will now see the registered mobiles under the name chosen
7. If you doubleclick on any of the mobiles, GoogleEarth will fly you to where they are. Enjoy!

About the GooglePos model

The GooglePos model consists of a number of packages representing different aspects of the model. Only one of the packages – namely GooglePos will be translated into JavaFrame code. The other packages have already been transformed or manually programmed.

GooglePos

GooglePos is the model of the system running on one computer. We have chosen not to include a package describing the context of the system. Such a context could have included the mobile user, PATS etc. The context is rather represented as formal ports on the *GPos* class representing the whole system.

SMSPorts

The GooglePos application uses special ports that are handmade to cope with communication with the PATS Telenor lab. The SMSPorts package also defines some signals that should be used to take advantage of the SMS services of PATs. The SMSPorts model library contains only the UML counterparts that mimic the SMSMediators java package. The SMSPorts package then makes sure that the compiler produces correct code for the SMS communication within GooglePos.

Java packages in UML

Within the GooglePos project there are also a couple of other packages – java and javaframe. These packages are also just mock-ups of existing java utilities where the implementation is given by the packages that are imported on the java level. In our case the java packages (io and util) are there because we need a linked list and we want to output on the kml-file.

JavaFrame concept in UML

The JavaFrame package on the other hand is there because there was one concept for which we had not defined a JavaFrame stereotype – namely a MultiCastMediator, a port that replicates a signal to all its outgoing connections. Most other special JavaFrame concepts have been introduced to UML through a stereotype. Please consult the compilers help function for explanation of the stereotypes and how to model towards JavaFrame.

About JavaFrame, JFTrace and the UML Compiler

The intention is that the students should not have to be experts in JavaFrame since JavaFrame is simply the execution target, some intermediate language. Therefore we have not provided a programming manual for JavaFrame.

We have, however, provided full source and javadoc for JavaFrame, but we do not recommend that you spend time resource familiarizing yourself in detail with JavaFrame.

The right approach is to familiarize yourself with the specialties of JavaFrame through reading the UML compilers documentation.

JFTrace is a stand-alone program used to accompany JavaFrame executions tracing the transitions. We have supplied the necessary JFTrace documentation.

About SMSPorts and SMSMediators

SMSPorts is the UML mock-up package of the SMSMediators java library which in turn uses the no.uio.ifi.pats.client.jar library.

SMSMediators is a library defining the special ports/mediators ("mediator" is the JavaFrame word for UML port) used to communicate with PATS.

We have provided javadoc and full source for the SMSMediators, but not for the underlying library. There is no reason for the students to know the details of that level.

The vocabulary of the transitions of the State Machines

Please refer to the compiler help function for how the text of the transitions of the state machines is coded.

The text of the transition is java. It talks about the JavaFrame context and therefore there is some simple vocabulary to be learned. The text of transitions is not checked by the UML compiler and may therefore contain mistakes that only appear as errors in the Java target.

Please always make sure to obey the following simple rules for how transitions are coded:

1. Transitions must always terminate within finite and short time. No intended waiting may occur inside a transition.
2. Transitions must never have side effects. This means that transitions should only update the state of the associated state machine or its local ports.

Here is a summary of the transition vocabulary.

- sending asynchronous signals
  - **output**(*<the signal>*,*<the port>*,*<current state machine>*)
- <the signal>
  - **new** *SignalType(parameters)*
  - **sig**
    - meaning the signal just consumed as trigger
- <current state machine>
  - **csm**
- <the port>
  - **csm**.*portname*

- State machine variables
  - **csm**.*variablename*