



PETER FOLDES
ANAR HUSEYNOV
JUSTIN KILLIAN
ISHWINDER SINGH

CARNEGIE MELLON UNIVERSITY
INSTITUTE OF SOFTWARE RESEARCH

Table of Contents

Introduction	2
Steps for Porting Actors	2
Identify Incompatible Code	2
Implement the PortablePlaceable interface	3
Creating an Interface	3
Creating the Platform-Specific Implementation	3
Mapping the Interface to the Implementation	3
Initializing Implementation Classes at Run-Time	4
Example: Porting AudioCapture actor	4
Sharing Common code	5

Introduction

To create a version of an actor that can run on Android, it involves both creating an Android-specific implementation for the actor and also modifying the code of the existing Ptolemy actor. The idea is to keep the parts of the actor that can run on both platforms common to both and wherever there is a platform specific functionality in an actor we abstract away the platform-specific logic by implementing a call to an interface. One should try to reuse as much code as possible without breaking other actors that are not being ported.

The architecture allows only the sources and sinks to run on the client and the middle actors run on the server. So the only sources and sinks can be ported to Android. For porting the sources which take some form of input from the handheld like the microphone in case of AudioCapture, it is required to create a platform specific implementation for capturing the data from the hardware.

For sink actors are used for visualization of the results on the simulation. In Verilog, Swing libraries are used to create the visualization like the in case of array plotter or the display actor. To port a sink actor to Android it is required that an implementation of the actor is created using Android specific UI libraries.

For actors that have implemented the `PLACEABLE` interface, they must be changed to implement the `PORTABLEPLACEABLE` interface. The `PORTABLEPLACEABLE` interface is analogous to the `PLACEABLE` interface. However, this interface is used by Homer and the Android application to inject the interface-based implementation without disrupting the rest of the Ptolemy code base. This is done in order to eliminate the strict dependency on packages like `java.awt` and `javax.swing`.

Steps for Porting Actors

The process for porting actors to the Android platform depends highly on the actor code itself and how intertwined platform-specific calls are. It involves separating the platform-dependent code from the platform-independent code and creating an interface with the matching signatures for methods that differ between implementations. The following are some of the various steps that can be taken to port an actor so that it is capable of running and/or creating a visual display on an Android device.

Identify Incompatible Code

In order to port an actor to Android, we have to identify the classes that the actor uses which are not supported on the Android platform. These include but are not limited to Swing, AWT, `javax.sound` or others depending on the functionality of the actor. The most effective starting point is to identify the imports to the class that do not compile on Android by looking in several places:

1. Direct imports to the class in the header section
2. In dependent Ptolemy classes that does not compile on Android - explore the class hierarchy of the actor and for each parent class, look for dependencies on classes that do not compile on Android.
3. Subclassed actors - make sure that all the child actors are also identified so that we do not break them while porting their parent actors.

Implement the PortablePlaceable interface

To port actors that implements the **PLACEABLE** interface they will have to implement the **PORTABLEPLACEABLE** interface instead. This interface contains a **PLACE()** method which adds itself to the provided platform independent container (**PORTABLECONTAINER**). At run-time, depending upon the target platform, this container is mapped to `java.awt.Container` or `android.widget.FrameLayout`. Actors that have UI components can be placed in these containers. Since the place method's implementation will be different for android so this method will be part of the interface that we will create in the next step.

Creating an Interface

After identifying the parts of code that will differ from platform to platform, create an interface that contains those method signatures. These will need to be implemented for each target platform. Next, move the code that is not supported on Android to the Java SE implementation of the interface. Replace the platform-dependent code in the actor with calls to an instance of the interface. For example, if the interface is named **ACTORINTERFACE** then create the implementation object using the `PtolemyInjector` as shown below.

```
ActorInterface _implementation =  
    PtolemyInjector.getInjector().getInstance(ActorInterface.class);
```

Creating the Platform-Specific Implementation

For creating the concrete implementation for an actor, implement all the methods of the interface created in the previous step. The Java SE implementation will contain code that was in the original actor but had dependencies on libraries that are incompatible with or not supported on Android. Because Android does not support all standard java features, the Android implementation of the interface of the actor will need to implement the interface using Android-specific classes.

Mapping the Interface to the Implementation

When an application is compiled for a target platform, whether it is Android or JavaSE, the bindings from interfaces to concrete implementations must be specified. This allows the injection framework to instantiate the appropriate implementation, using reflection, for the current platform at run-time. This is done by obtaining the mapping from the interface to

implementation from the **.PROPERTIES** files. The mapping from the interface in step 3 to the actual implementation must be done in the following files depending on the target environment:

- JavaSE - `ptolemy.actor.JavaSEActorModule.properties`
- Android - `ptdroid.actor.gui.AndroidActorModule.properties`

Initializing Implementation Classes at Run-Time

All classes that will be platform-independent such as **CONTAINER** will use the injector framework to get the concrete implementation. To load and initialize the appropriate classes at run-time, the **INITIALIZEINJECTOR()** method of **ACTORMODULEINITIALIZER** class must be called. This must be done prior to the instantiation of any platform-specific class. Examples of this can be seen in Vergil and the ptdroid project where the **ACTORMODULEINITIALIZER.INITIALIZEINJECTOR()** is called from the **MAIN()** function of **VERGILAPPLICATION** and in the **ONCREATE()** method of **SIMULATIONACTIVITY** respectively.

Example: Porting AudioCapture actor

AudioCapture class uses LiveSound which had dependency on platform specific libraries.

The following classes used by LiveSound are not compatible with Android and there are dependencies on these classes in various methods within the class:

```
javax.sound.sampled.AudioFormat;  
javax.sound.sampled.AudioSystem;  
javax.sound.sampled.DataLine;  
javax.sound.sampled.LineUnavailableException;  
javax.sound.sampled.SourceDataLine;  
javax.sound.sampled.TargetDataLine;
```

To port this actor we will first create an interface that will contain all the methods from LiveSound. Name this interface LiveSoundInterface. Create a reference variable of the type LiveSoundInterface in LiveSoundClass and assign to it the instance returned from PtolemyInjector as show below:

```
private static LiveSoundInterface _implementation = PtolemyInjector
    .getInjector().getInstance(LiveSoundInterface.class);
```

A static variable is created because it is used inside static methods in the class. Now for each method inside the LiveSound class uses this object to call corresponding methods of LiveSoundInterface. As shown below:

```
public static void removeLiveSoundListener(LiveSoundListener listener) {
    _implementation.removeLiveSoundListener(listener);
}
```

Then create platform specific implementation for LiveSoundInterface on Java SE and on Android. Name these classes LiveSoundJavaSE and AndroidLiveSound. Note that AndroidLiveSound is created in ptdroid project

To implement all the functionality of LiveSound actor on Android we will need following platform specific classes:

```
android.media.AudioFormat;
android.media.AudioManager;
android.media.AudioRecord;
android.media.AudioTrack;
android.media.MediaRecorder.AudioSource;
```

Sharing Common code

There are some methods in LiveSound that can be used on both platforms since they do not use any platform specific dependency. So select these methods and instance variables that can be used in both the implementations and then created a common class that will be inherited by both. In this case following are the methods that do not have and platform specific dependency:

```
void addLiveSoundListener(LiveSoundListener listener)
int getBitsPerSample()
int getBufferSize()
int getChannels()
int getSampleRate()
int getTransferSize()
boolean isCaptureActive()
boolean isPlaybackActive()
void removeLiveSoundListener(LiveSoundListener listener)
void setTransferSize(int transferSize)
void _byteArrayToDoubleArray(double[][] doubleArray, byte[] byteArray)
byte[] _doubleArrayToByteArray(double[][] doubleArray)
void _notifyLiveSoundListeners(int parameter)
```

Inheriting a common class will avoid duplication of code and make it more maintainable. Name this class LiveSoundCommon. All the other methods of the LiveSoundMethods have platform

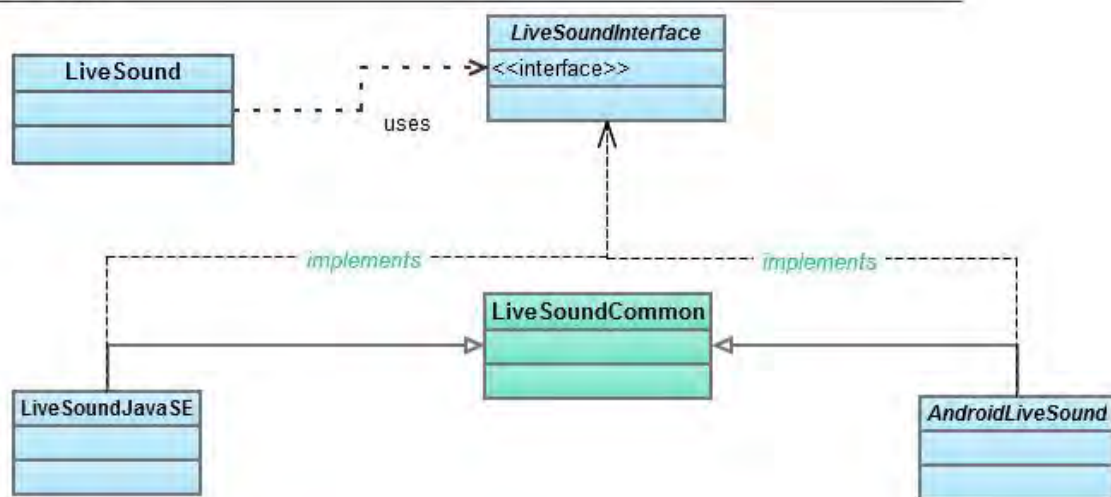
dependency so they will be implemented differently in AndroidLiveSound and LiveSoundJavaSE.

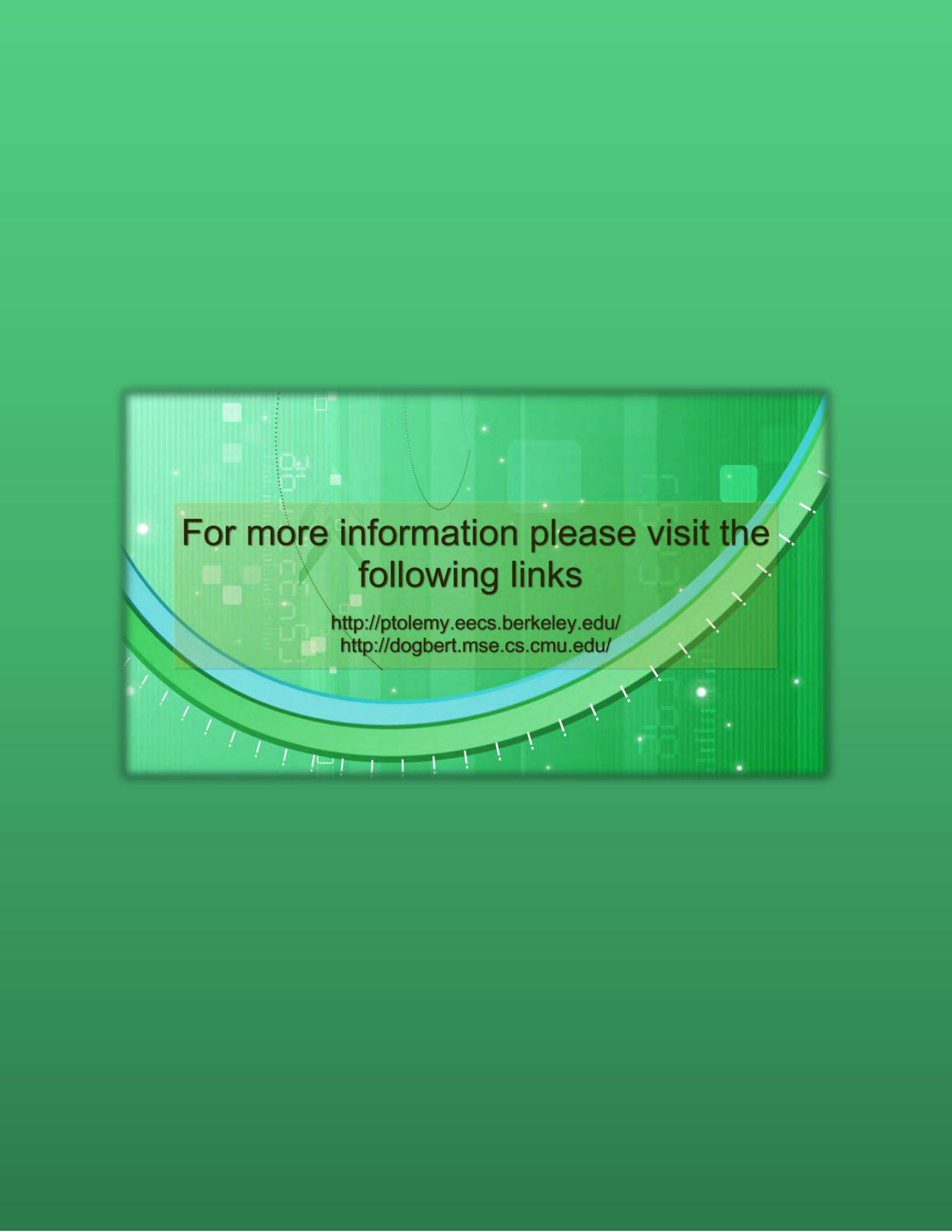
Update the following property files to enable instantiation of correct object during runtime:

- JavaSE - ptolmy.actor.JavaSEActorModule.properties
- Android - ptdroid.actor.gui.AndroidActorModule.properties

Add mapping from LiveSoundInterface to AndroidLiveSound for ptdroid and to LiveSoundJavaSE for Vergil.

Class Diagram





For more information please visit the
following links

<http://ptolemy.eecs.berkeley.edu/>
<http://dogbert.mse.cs.cmu.edu/>