Creating SAP xMII 11.5 User-Defined Action Blocks

Applies to:

xMII 11.5 SP3

Summary

This article will describe step-by-step the process of writing user-defined Action Blocks in Java which can be deployed in xMII 11.5 SP3 and manipulated using xMII's Business Logic Editor.

Author: Tim Drury

Company: Visiprise

Created on: 10 October 2007

Author Bio

Tim Drury is the lead Integration Architect at Visiprise, a leader in discrete manufacturing execution software. Tim has a bachelor's degree in electrical engineering from the Georgia Institute of Technology where he studied real-time embedded systems and data acquisition systems. He spent 14 years building embedded systems before realizing his software skills were not on par with his hardware skills. Focusing on software, Tim came to the realization that software development has less expensive, faster development cycles than hardware development – perfect for someone with a short attention span and little patience. Hardware development is now relegated to hobby status along with all things in the realm of motorsports. Tim has spent the last 8 years writing software to integrate applications, machines, and technology now with a focus on manufacturing. On the side he tinkers with race car data acquisition and control systems.

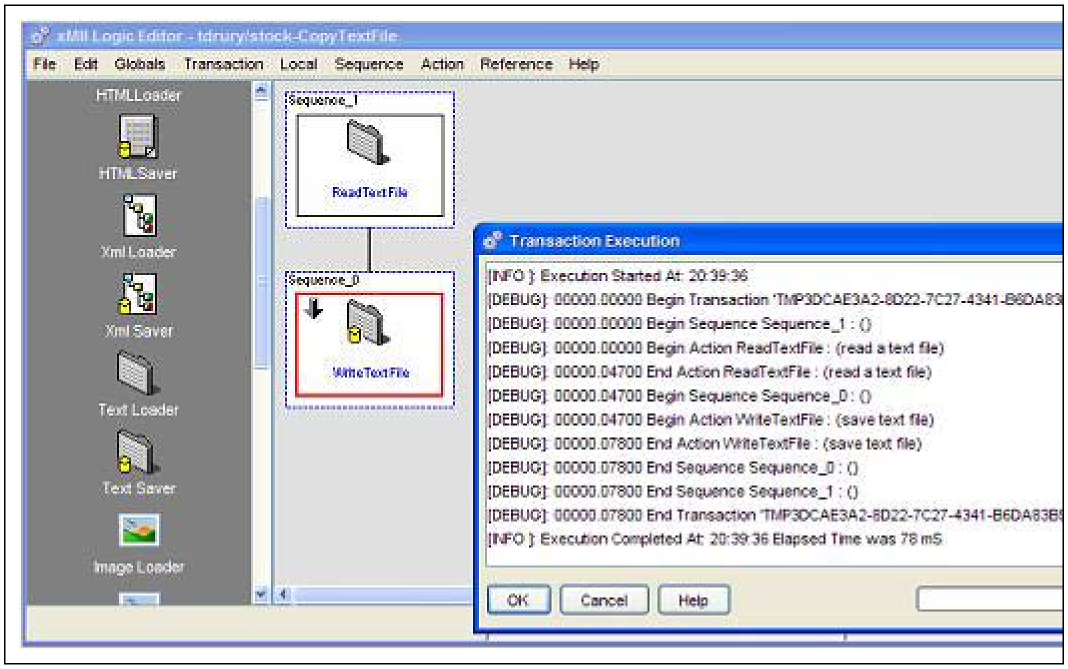
Table of Contents

Introduction

SAP xMII is a toolkit for building manufacturing business processes and visualization dashboards. It maintains a smart balance between functionality and succinctness – it provides just enough Action Blocks and software operations to build almost any integration workflow. However, over time an xMII developer will uncover situations where the stock xMII Action Block palette doesn't provide the functionality or performance required. Enter the user-defined Action Block, or action. In xMII 11.5 (and 12.0) it is possible to create your own Action Blocks in Java and make them available in the xMII Business Logic Editor action palette and the xMII engine. In this first article, I'm going to demonstrate how to create and deploy a file copying action imaginatively named `FileCopy`. `FileCopy` will use Java's Native I/O (NIO) package to greatly improve the performance of copying files.

xMII Text File Issues

Copying a text file in xMII is trivial. You drop in a `Text Loader` action followed by a `Text Saver` action and link the `Loader`'s `Contents` output to the `Saver`'s `Contents` input. The results of copying a 1 MB text file are reasonable as Figure 1 can attest. It took 78 ms in this particular screenshot from my Dell Latitude D620 laptop to copy the 1 MB text file. I ran the test several times (both before and after the subsequent tests) and saw some variation in times but 70 ms was about average. Don't put too much weight on the absolute numbers from these tests – instead we'll focus on the relative differences of the times of the tests.

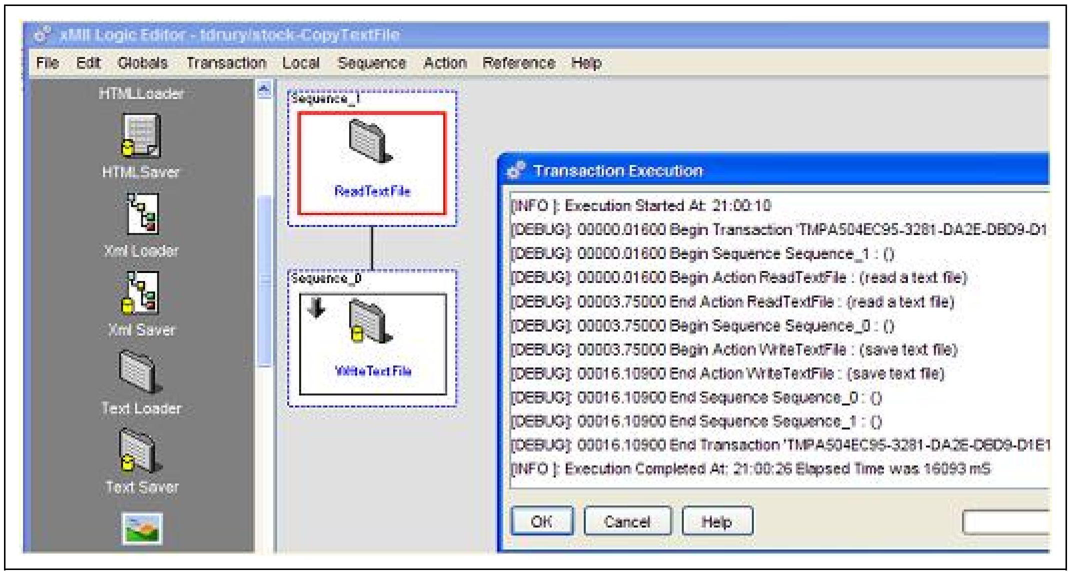


```
[INFO ]: Execution Started At: 20:39:36
[DEBUG]: 00000.00000 Begin Transaction 'TMP3DCAE3A2-8D22-7C27-4341-B6DA83
[DEBUG]: 00000.00000 Begin Sequence Sequence_1 : ()
[DEBUG]: 00000.00000 Begin Action ReadTextFile : (read a text file)
[DEBUG]: 00000.04700 End Action ReadTextFile : (read a text file)
[DEBUG]: 00000.04700 Begin Sequence Sequence_0 : ()
[DEBUG]: 00000.04700 Begin Action WriteTextFile : (save text file)
[DEBUG]: 00000.07800 End Action WriteTextFile : (save text file)
[DEBUG]: 00000.07800 End Sequence Sequence_0 : ()
[DEBUG]: 00000.07800 End Sequence Sequence_1 : ()
[DEBUG]: 00000.07800 End Transaction 'TMP3DCAE3A2-8D22-7C27-4341-B6DA83B
[INFO ]: Execution Completed At: 20:39:36 Elapsed Time was 78 mS
```

Figure 1.

The `Text Loader` action can copy a 1 MB text file in 78ms.

Let's try the same experiment with a 100 MB text file. We can guess it may take 100 times longer to copy this file, but instead we see it takes 16 seconds – some 200 times longer than copying the 1 MB text file (Figure 2).



```
[INFO ]: Execution Started At: 21:00:10
[DEBUG]: 00000.01600 Begin Transaction 'TMPA504EC95-3281-DA2E-DBD9-D1
[DEBUG]: 00000.01600 Begin Sequence Sequence_1 : ()
[DEBUG]: 00000.01600 Begin Action ReadTextFile : (read a text file)
[DEBUG]: 00003.75000 End Action ReadTextFile : (read a text file)
[DEBUG]: 00003.75000 Begin Sequence Sequence_0 : ()
[DEBUG]: 00003.75000 Begin Action WriteTextFile : (save text file)
[DEBUG]: 00016.10900 End Action WriteTextFile : (save text file)
[DEBUG]: 00016.10900 End Sequence Sequence_0 : ()
[DEBUG]: 00016.10900 End Sequence Sequence_1 : ()
[DEBUG]: 00016.10900 End Transaction 'TMPA504EC95-3281-DA2E-DBD9-D1E1
[INFO ]: Execution Completed At: 21:00:26 Elapsed Time was 16093 mS
```

Figure 2.

The `Text Loader` took 16 seconds to copy a 100 MB text file – a factor of 200 times longer than test 1 MB text file.

However, more troubling than the unintuitive time difference is the gluttonous memory consumption. Figure 3 is from xMII's Java Runtime Status applet and shows the heap taking a precipitous, lemming-like dive as soon as the test started. Had my JVM not had a generous 512 MB heap, I would likely be showing a screenshot of an `OutOfMemoryError` stack trace. The reason for this? `Text Loader` reads the entire 100 MB text file into a `StringBuffer` before writing the output file. That's not good.

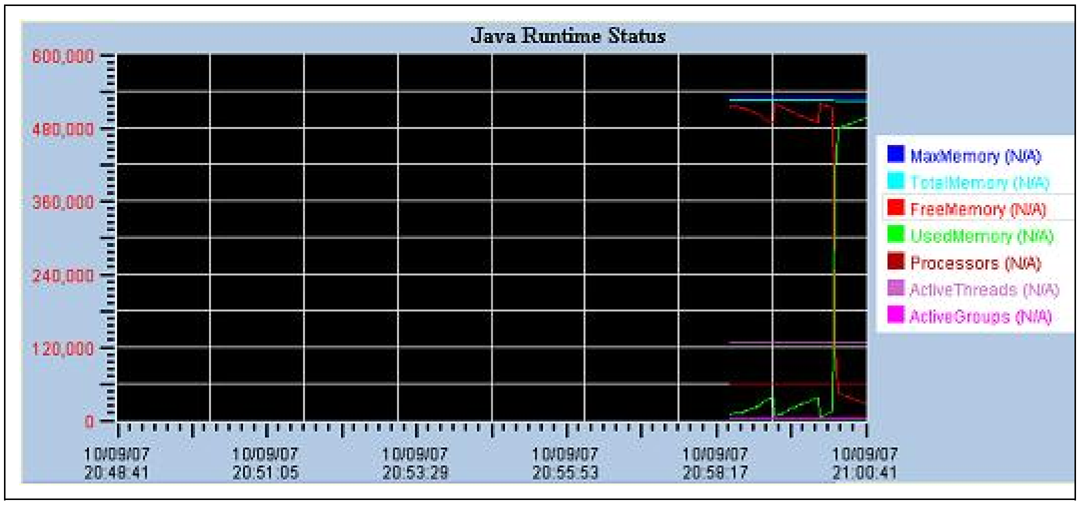
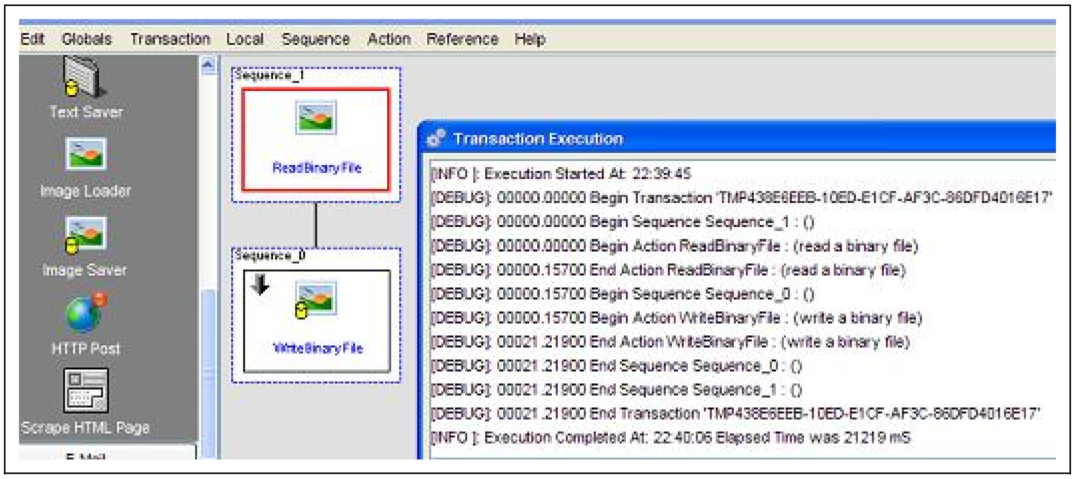


Figure 3.

The `Text Loader` action reads the entire 100 MB text file into memory before writing it back out again. This could be a problem for JVMs with limited heap.

xMII Binary File Issues

Another functionality gap with xMII 11.5 is the lack of actions for copying and/or manipulating binary files. More fundamentally, it xMII's lack of a binary property type. What xMII does provide is the `Image Loader` and the `Image Saver` actions. However, `Image Loader` must perform Base64 encoding on the binary stream to convert it to xMII's `Text` type because there is no binary type available. Likewise, the `Image Saver` requires its input to be a Base64 encoded `Text` type to write the binary stream to a file. This conversion imposes a harsh performance penalty when copying binary files. Indeed, Figure 4 shows that xMII required 21 seconds to copy a 1 MB binary file – recall it took only 78 ms to copy a 1 MB text file.



```
[INFO ]: Execution Started At: 22:39:45
[DEBUG]: 00000.00000 Begin Transaction 'TMP438E6EEB-10ED-E1CF-AF3C-86DFD4016E17'
[DEBUG]: 00000.00000 Begin Sequence Sequence_1 : ()
[DEBUG]: 00000.00000 Begin Action ReadBinaryFile : (read a binary file)
[DEBUG]: 00000.15700 End Action ReadBinaryFile : (read a binary file)
[DEBUG]: 00000.15700 Begin Sequence Sequence_0 : ()
[DEBUG]: 00000.15700 Begin Action WriteBinaryFile : (write a binary file)
[DEBUG]: 00021.21900 End Action WriteBinaryFile : (write a binary file)
[DEBUG]: 00021.21900 End Sequence Sequence_0 : ()
[DEBUG]: 00021.21900 End Sequence Sequence_1 : ()
[DEBUG]: 00021.21900 End Transaction 'TMP438E6EEB-10ED-E1CF-AF3C-86DFD4016E17'
[INFO ]: Execution Completed At: 22:40:06 Elapsed Time was 21219 mS
```

Figure 4.

The `Image Loader` action reads a 1 MB binary file and the Image Saver writes this file back to the disk – total time is 21 seconds.

Compounding the performance penalty, take a look at the CPU usage during the binary file copying process (Figure 5). Base64 encoding the binary stream and decoding the text stream back to binary takes a heavy toll on the CPU. This would usually be grudgingly tolerated if we truly needed to move binary into the text domain and vice versa, but that's not the case here – we don't need these binary-to-text and text-to-binary conversions so why do we have to pay a penalty?

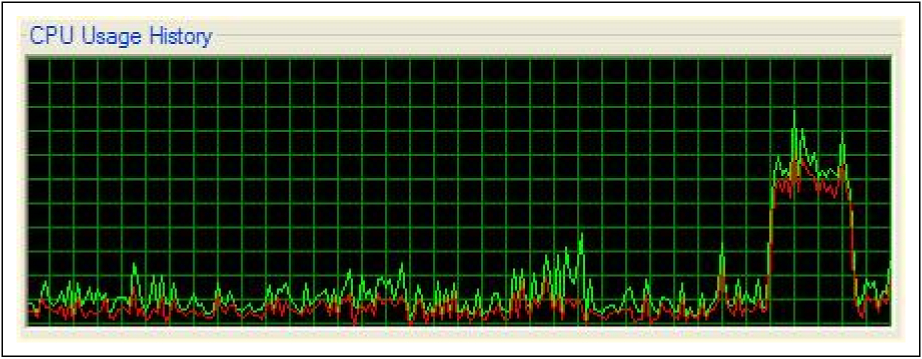


Figure 5.

Base64 encoding of a binary stream takes its toll on my CPU. This is especially disheartening considering I don't need the binary to be encoded – I would have liked it to remain in binary.

Enter the User-Defined Action

There doesn't seem to be any published APIs for building your own action blocks in xMII, but there is information to be found if you know where to look – or who to ask. To remedy this, I will show the two required classes for our `CopyFile` action with some descriptive commentary. Then I will show how to bundle and deploy the user-defined action assembly into xMII 11.5. Lastly, we'll compare and contrast the results of our `CopyFile` action with the text and binary file copy transactions created with the stock xMII actions.

xMII action require two classes: an action class and a configuration class. The action class provides the run-time behavior of the action and is called by the xMII engine when the `CopyFile` action is placed in an xMII transaction. The second is a Swing class used by the Business Logic Editor (BLE) – the Java Web Start application we use to build xMII transactions.

The `CopyFile` Action Class

The xMII developers were kind enough to provide a base class for xMII actions called `ActionReflectionBase`. By extending this class most of the development effort will already be done. In a later installment I'll dive deeper in `ActionReflectionBase`, what interfaces it implements, and when you wouldn't want to extend it.

First, we declare our `CopyFile` action's package namespace:

```
    package com.visiprise.app.integration.sap.xmii.actions.util;
```

I'm not going to include the `import` statements since most IDEs can determine which `imports` are missing and create them. Next, we have our class declaration extending `ActionReflectionBase`:

```
    public class CopyFile extends ActionReflectionBase {
```

Our `CopyFile` action requires five attributes. We need a `String` to hold the path of the file to be copied (the `source` file) and a `String` for the path of the file to be created (the `destination` file). We'll give the user a `boolean` flag to indicate whether we should overwrite any pre-existing `destination` file. We'll also want a `String message` to hold the action status which will provide some feedback to the user informing them of the outcome of the `CopyFile` action. The last attribute, a `boolean` status flag named `_status`, is declared in `ActionReflectionBase` so we don't need to declare it here.

```
      private String source;
      private String destination;
      private boolean overwrite = false;
      private String message = "";
```

Here are some standard Java Bean-style `get`ters and `set`ters and now we'll see how extending `ActionReflectionBase` helps us. As its name implies, `ActionReflectionBase` will use Java reflection to determine the inputs and outputs of the `CopyFile` action. The inputs and outputs are used in the BLE link editor to map actions' arguments to one another, and by the xMII engine to perform the run-time binding of real values to these links. All public methods that begin with "`get`" or "`is`" will be parsed to create outputs for the `CopyFile` action…

```
      public String getSource() {
            return source;
      }
      public String getDestination() {
            return destination;
      }
      public boolean getOverwrite() {
            return overwrite;
      }
      public String getMessage() {
            return message;
      }
```

…and all public methods that begin with "`set`" will be parsed to create inputs for the `CopyFile` action. The actual names of the inputs and outputs are the method name without the "`get`", "`set`", or "`is`" prefix. You may notice that I did not create a `setMessage()` method. This message is an internally generated status and it wouldn't make sense to give the user the ability to set it. By not providing a `setMessage()` method, `ActionReflectionBase` will not create an input for our `message` attribute keeping it from appearing in the BLE link editor.

```
	public void setSource(String source) {
		this.source = source;
	}
	public void setDestination(String destination) {
		this.destination = destination;
	}
	public void setOverwrite(boolean overwrite) {
		this.overwrite = overwrite;
	}
```

`ActionReflectionBase` also provides a helper method that will load an icon given its path relative to the xMII `CLASSPATH` – we only need to provide the path. This icon is used in the BLE action palette. The `CopyFile` action's icon is in a folder called `/icons` which resides in the root of a jar file. I'll describe this jar file a little later.

```
	public String GetIconPath() {
		return "/icons/CopyFile.png";
	}
```

Now we've finally arrived at the interesting part of the `CopyFile` action – the `Invoke()` method. This method is called by the xMII engine when its turn to execute has arrived. Class attributes represented as inputs will have values bound to the attribute references by the xMII engine prior to `Invoke()` being executed. `Invoke()` is also provided a reference to the xMII transaction in which the action resides and a logger for writing status messages. You may notice that `CopyFile` has no need to use the `transaction` reference – in fact, I have yet to encounter a circumstance needing it, but I've only recently begun writing xMII actions. All the code within `CopyFile`'s `Invoke()` method is the typical Java NIO file-copy algorithm.

Since I'm not entirely sure how xMII deals with exceptions thrown from within the `Invoke()` method, I catch `Exception` itself. I don't want to catch `Throwable` because there isn't much my `CopyFile` action can do with a `ThreadDeath` or `OutOfMemoryError`. Also, for the sake of brevity I create an `Exception` for all error conditions within my `try` block – this is not a Java best-practice as exception handling is expensive in terms of performance. This action would execute more quickly if, for each error condition, I simply set the `message` and `_status` and then `return`ed.

```
	public void Invoke(Transaction transaction, ILog log) {
		FileInputStream sourceStream = null;
		FileOutputStream destinationStream = null;
		FileChannel sourceChannel = null;
		FileChannel destinationChannel = null;
		try {
			if (source == null)
				throw new Exception("Source file is not defined.");
			if (destination == null)
				throw new Exception("Destination file is not defined.");
			// validate source file
			File in = new File(source);
			if (!in.exists())
				throw new Exception("Source file \""+source+"\" not 
   found.");
			if (in.isDirectory())
				throw new Exception("Source file cannot be a 
   directory.");
			// validate destination file
```

```
              File out = new File(destination);
              if ( (!overwrite) && (out.exists()) ) throw new
Exception("Destination file \""+destination+"\" exists and overwrite mode is
off.");
              // create NIO file channels and copy the file
              sourceStream = new FileInputStream(in);
              sourceChannel = sourceStream.getChannel();
              destinationStream = new FileOutputStream(out);
              destinationChannel = destinationStream.getChannel();
              long num = sourceChannel.transferTo(0, sourceChannel.size(),
destinationChannel);
              message = "copied ["+num+"] bytes from ["+source+"] to
["+destination+"]";
              _success = true;
        } catch (Exception ex) {
              message = ex.getMessage();
              log.error("Copy File Error: " + message);
              _success = false;
        } finally {
```

We would never think of performing I/O without a `finally` block to clean up our resources would we?

```
              if (sourceChannel != null)
                    try { sourceChannel.close(); } catch (IOException ex) {
                          log.debug("CopyFile: source channel failed to
close cleanly; error="+ex.getMessage());    }
              if (destinationChannel != null)
                    try { destinationChannel.close(); } catch (IOException
ex) {
                          log.debug("CopyFile: destination channel failed to
close cleanly; error="+ex.getMessage()); }
              if (sourceStream != null)
                    try { sourceStream.close(); } catch (IOException ex) {
                          log.debug("CopyFile: source stream failed to close
cleanly; error="+ex.getMessage()); }
              if (destinationStream != null)
                    try { destinationStream.close(); } catch (IOException
ex) {
                          log.debug("CopyFile: destination stream failed to
close cleanly; error="+ex.getMessage()); }
        }
  }
}
```

The Copy File Dialog

Each action in the xMII BLE can be manipulated by two dialogs: the configuration dialog and the link editor dialog. The link editor dialog is managed entirely by xMII; its only requirement of our user-defined action is to make sure our action implements the `IMappable` interface which `ActionReflectionBase` does for us. This is the interface used to acquire the action's input and output names, types, and values. The configuration dialog, on the other hand, is entirely our responsibility to construct and manage. Figure 6 shows the `CopyFile` configuration dialog.

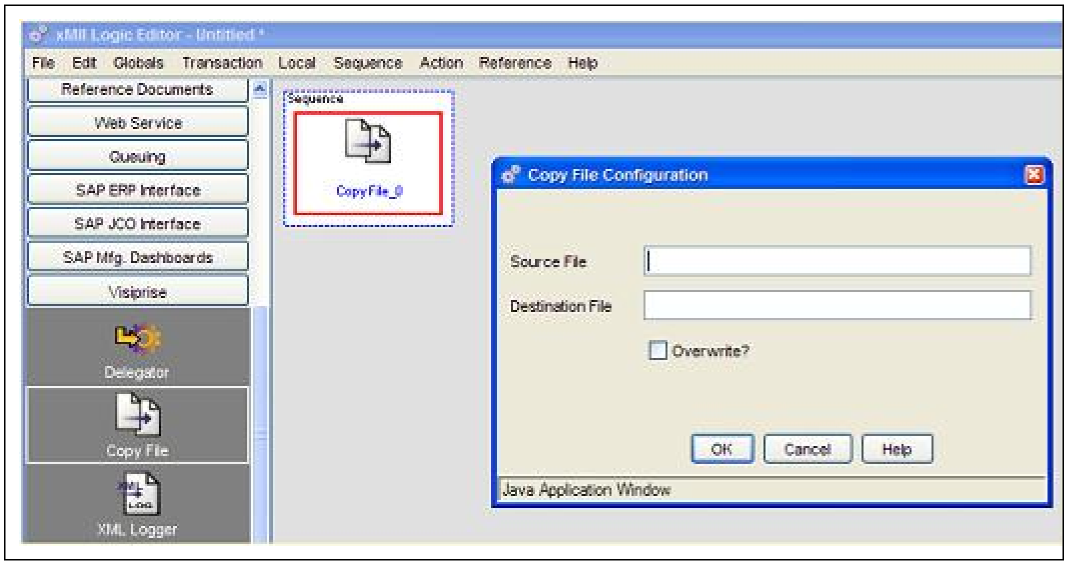


Figure 6.

The Copy File Configuration dialog box is a simple Swing JDialog.

Now let's go through the `CopyFile` configuration dialog class. The package namespace is the same as the CopyFile action class:

```
package com.visiprise.app.integration.sap.xmii.actions.util;
```

I'll skip the `import` statements again. `AbstractActionDialog` is another helpful base class that can be extended to minimize development effort:

```
public class CopyFileDialog extends AbstractActionDialog {
```

We only need three Swing widgets to configure the `CopyFile` action. The `source` text field will hold a path to the file being copied, the `destination` text field will hold a path to the destination file to create or overwrite, and the `overwrite` check box indicates if overwriting an existing destination file is allowed.

```
    private JTextField source;
    private JTextField destination;
    private JCheckBox  overwrite;
```

We don't have any initialization to perform, so our constructors can simply defer to the `AbstractActionDialog`'s constructors:

```
    public CopyFileDialog(JFrame parentFrame, Object actionObject, Transaction
transaction, Step selectedAction) {
            super(parentFrame, actionObject, transaction, selectedAction);
    }
    public CopyFileDialog(JDialog parentFrame, Object actionObject,
Transaction transaction, Step selectedAction) {
            super(parentFrame, actionObject, transaction, selectedAction);
    }
```

`prepareDialog()` is used to give the configuration dialog box a size and title and `createLayoutObjects()` initializes the configuration dialog Swing widgets:

```
      public void prepareDialog() {
            setSize(400, 250);
            setTitle("Copy File Configuration");
      }
      public void createLayoutObjects() {
            source = new JTextField(20);
            destination = new JTextField(20);
            overwrite = new JCheckBox("Overwrite?");
      }
```

`layoutMainPanel()` is the method in which the configuration dialog is constructed. I've chosen to use the `GridBagLayout` layout manager which provides the most flexibility when constructing Swing dialogs, but you're free to use any of the Swing layout managers. If you're not familiar with Swing, then you're going to have to trust me here since a detailed discussion of Swing is out of scope for this article. I'm not going to describe my rationale for these `GridBagConstraints` (nor will I defend them), but you're welcome to learn about them yourself in the Java Swing tutorial.

`layoutMainPanel()` begins by creating a panel to hold the three configuration dialog widgets – the two text fields and the check box – as well as some labels. The panel is assigned the `GridBagLayout` manager and `GridBagConstraints` will be created:

```
        public void layoutMainPanel() {
            JPanel panel = new JPanel(new GridBagLayout());
            GridBagConstraints gbc = new GridBagConstraints();
```

The widgets are placed in the panel from top-to-bottom and left-to-right. The first widget placed is a label for the `source` text field. Then we place the `source` text field widget itself:

```
            gbc.gridy = 0;
            gbc.insets = ActionUtilities.standardSquareInsets;
            gbc.weightx = 0.1;
            gbc.fill = GridBagConstraints.NONE;
            gbc.anchor = GridBagConstraints.WEST;
            panel.add(new JLabel("Source File"), gbc);
            gbc.weightx = 0.9;
            gbc.fill = GridBagConstraints.HORIZONTAL;
            panel.add(source, gbc);
```

Then a label for the `destination` text field followed by the text field itself:

```
            gbc.gridy = gbc.gridy+1;
            gbc.insets = ActionUtilities.standardSquareInsets;
            gbc.weightx = 0.1;
            gbc.fill = GridBagConstraints.NONE;
            gbc.anchor = GridBagConstraints.WEST;
            panel.add(new JLabel("Destination File"), gbc);
            gbc.weightx = 0.9;
            gbc.fill = GridBagConstraints.HORIZONTAL;
            panel.add(destination, gbc);
```

Then the `overwrite` check box is added. Since a Swing check box already has a label, I don't need to create a label for it, but to keep the widgets in line vertically, I've added an empty label widget:

```
            gbc.gridy = gbc.gridy+1;
            gbc.insets = ActionUtilities.standardSquareInsets;
            gbc.weightx = 0.1;
            gbc.fill = GridBagConstraints.NONE;
            gbc.anchor = GridBagConstraints.WEST;
            panel.add(new JLabel(""), gbc);
            gbc.weightx = 0.9;
            gbc.fill = GridBagConstraints.HORIZONTAL;
            panel.add(overwrite, gbc);
```

The last step is to add this panel to the configuration dialog widget. I use `GridBagLayout` for the configuration dialog also and I create a fresh set of `GridBagConstraints` – this isolates changes that may be made to the above `GridBagConstraints` from inadvertently impacting this new dialog layout. We add the panel we constructed above to this new dialog, and we add a button panel provided by our `AbstractActionDialog` parent class. This button panel contains the "OK", "Cancel", and "Help" buttons and `AbstractActionDialog` handles the events generated by those buttons for us:

```
            getContentPane().setLayout(new GridBagLayout());
            GridBagConstraints gbc1 = new GridBagConstraints();
            gbc1.insets = ActionUtilities.standardSquareInsets;
            gbc1.gridy = 0;
            gbc1.weightx = 1.0;
            gbc1.weighty = 1.0;
            gbc1.fill = GridBagConstraints.BOTH;
            getContentPane().add(panel, gbc1);
            gbc1.gridy = gbc1.gridy + 1;
            gbc1.weightx = 1.0;
            gbc1.weighty = 0.0;
            gbc1.fill = GridBagConstraints.NONE;
            gbc1.anchor = GridBagConstraints.EAST;
            getContentPane().add(okPanel, gbc1);
        }
```

The last two methods are the pivotal ones. `setAction()` is responsible for setting the values in the configuration dialog widgets from the actual values that have been assigned to the `CopyFile` action object. The `actionObject` attribute used below is provided by `AbstractActionDialog` and always provides a valid reference to our `CopyFile` action object:

```
        public void setAction(Object obj) {
            source.setText(((CopyFile)actionObject).getSource());
            destination.setText(((CopyFile)actionObject).getDestination());
            overwrite.setSelected(((CopyFile)actionObject).getOverwrite());
        }
```

`performOK()` does the reverse of `setAction()` above – it takes the values entered into the `CopyFile` configuration dialog widgets and assigns these values to the corresponding reference in the `CopyFile` action object:

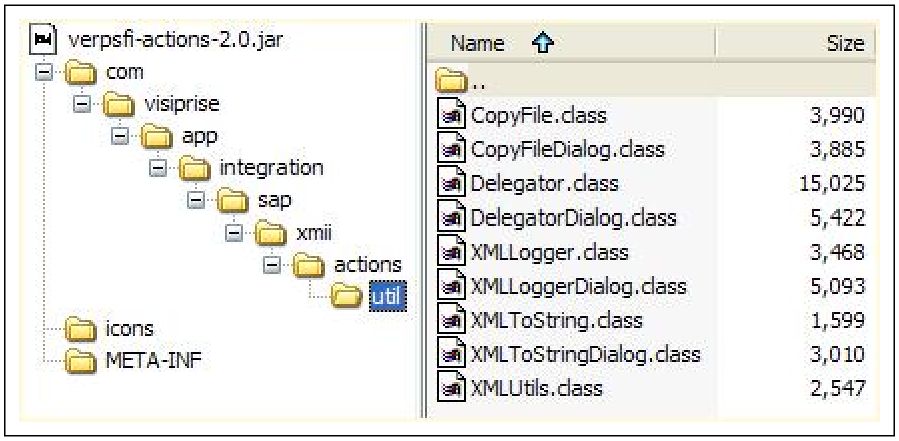
```
        public boolean performOK() {
            ((CopyFile)actionObject).setSource(source.getText());
            ((CopyFile)actionObject).setDestination(destination.getText());
            ((CopyFile)actionObject).setOverwrite(overwrite.isSelected());
            return true;
        }
    }
```

Compiling The Action and Dialog Classes

To compile the `CopyFile` and `CopyFileDialog` classes, some xMII classes need to be included on the compilation `CLASSPATH`. These classes can be found in `<ServletExecHome>\se-xMII\webapps\default\Lighthammer\WEB-INF\classes`. The jar, `<ServletExecHome>\se-xMII\webapps\default\Lighthammer\WEB-INF\lib\LHCommon.jar`, may also be required.

The Action Assembly Jar

To deploy user-defined actions within xMII, create a jar for the class files and icons. Figure 7 below shows the jar structure containing the `CopyFile` action and configuration dialog classes (as well as some other actions I created). The `/icons` folder in the root of the jar contains the icon PNG files which will be shown in the BLE action palette. The `/META-INF` folder contains a standard `MANIFEST.MF` file – there are no special entries in this file.



```
verpsfi-actions-2.0.jar
  com
    visiprise
      app
        integration
          sap
            xmii
              actions
                util
  icons
  META-INF
```

Name	Size
..	
CopyFile.class	3,990
CopyFileDialog.class	3,885
Delegator.class	15,025
DelegatorDialog.class	5,422
XMLLogger.class	3,468
XMLLoggerDialog.class	5,093
XMLToString.class	1,599
XMLToStringDialog.class	3,010
XMLUtils.class	2,547

Figure 7.

The structure of the jar containing our xMII 11.5 user-defined actions is very simple. Our classes and icons are in the root of the jar along with the META-INF folder for the manifest file. There are no special contents in the manifest.

The Action Assembly Catalog

The last file needed is an XML deployment descriptor, or action catalog. This file will describe to xMII the actions contained within the jar, above, and provide metadata for the BLE. For brevity's sake, I removed all the `<Component>` entries except for the `CopyFile` action. Refer to Figure 8 below for the remainder of the description of the action catalog.

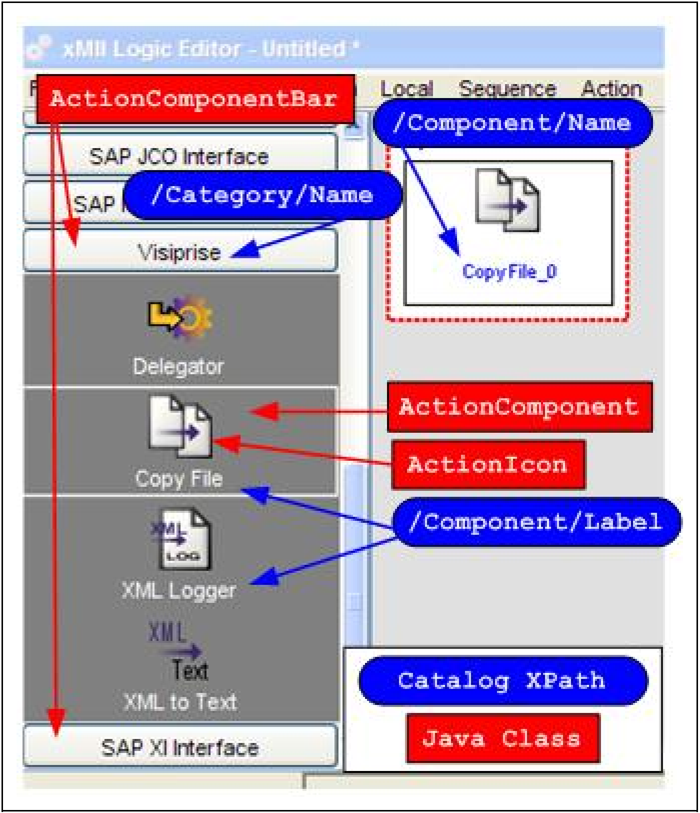


Figure 8.

This diagram shows how the action assembly catalog elements map to xMII classes and how the catalog element attributes map to visual elements of the xMII Business Logic Editor.

The `<Category>` element creates a new `ActionComponentBar` in the BLE action palette and the `Name` attribute of `<Category>` is the label displayed at the top of the `ActionComponentBar`. An `ActionComponentBar` is the accordion-like widget in the action palette (see) which holds the actions we define using the `<Component>` element below. Within the `ActionComponentBar` are xMII Swing widgets known as `ActionIcons` and each keeps a reference to an `ActionComponent` which is a simple Java Bean that holds all the metadata in the action's `<Component>` entry of the action catalog. I do not know of any use of the `<Category>`'s `Description` attribute.

```
<?xml version="1.0" encoding="UTF-8"?>
<ComponentCatalog>
  <Category Name="Visiprise"
                Description="SAP Manufacturing Execution System by
Visiprise">
```

For this article, I've removed all the `<Component>` entries except for the `CopyFile` action. For each action to be included within the `ActionComponentBar`, create a `<Component>` element with the attribute `Type` of "Action":

```
        <Component Type="Action"
```

The `<Component>` `Name` attribute serves two purposes. Its primary purpose is as a widget identifier – it is used internally by xMII as the `ActionIcon`'s AWT `source` name. If you're not familiar with AWT, `source` is an identifier for the widget which AWT places within any AWT `EventObject` that originated from the widget. `Name` should be unique within the `<Category>`. `Name`'s other purpose is to be the base of the instantiated action's name within a sequence; xMII will append a unique Integer suffix of the form "_nn" to `Name`:

```
                 Name="CopyFile"
```

I have found no use for the `Description` attribute:

```
                 Description="Copy a file"
```

The `Label` attribute's value, "Copy File" in this example, is added to the `ActionIcon` widget and displayed under the action's icon within the `ActionComponentBar`:

```
                 Label="Copy File"
```

`ClassName` should be set to the fully qualified name of the action. In this example, it refers to the `CopyFile` action class name:

```
       ClassName="com.visiprise.app.integration.sap.xmii.actions.util.CopyFile"
```

The `AssemblyName` attribute identifies the jar that contains the user-defined action classes and icons. Exactly where this assembly jar must be placed is discussed in the deployment section below.

```
                 AssemblyName="verpsfi-actions-2.0.jar"
```

The `HelpFileName` attribute is part of a URL which xMII adds to the base URL, "/LighthammerCMS/Help/SAP_xMII_Help.htm#Business_Logic_Services/" and adds the suffix ".htm". This URL is loaded in a browser when the Help button is clicked in the configuration dialog. For example, if `HelpFileName` was defined as "CopyFileHelp", then clicking the `CopyFile` configuration dialog's help button would load URL "/LighthammerCMS/Help/SAP_xMII_Help.htm#Business_Logic_Services/CopyFileHelp.htm". Since it would be difficult for us to locate a help file at that URL, the `HelpFileName` provides no help to us.

```
                 HelpFileName=""/>
      </Category>
   </ComponentCatalog>
```

Deploying Our User-Defined Action

To deploy the user-defined actions, shut down xMII by stopping the ServletExec service. Then copy the assembly jar to two directories:

- `<ServletExecHome>\se-xMII\webapps\default\Lighthammer\WEB-INF\lib`

This directory is accessed by the xMII engine which must be able to access the `CopyFile` action class so it can run its `Invoke()` method.

- `<ServletExecHome>\se-xMII\webapps\default\Lighthammer\CMSLogicEditor`

And this directory is used by the BLE's Web Start application. All the jars in this directory will be bundled up in the Web Start archive and distributed to all the xMII clients.

Be sure to clear your client's Java Web Start cache every time the user-defined action assembly jar is updated and deployed.

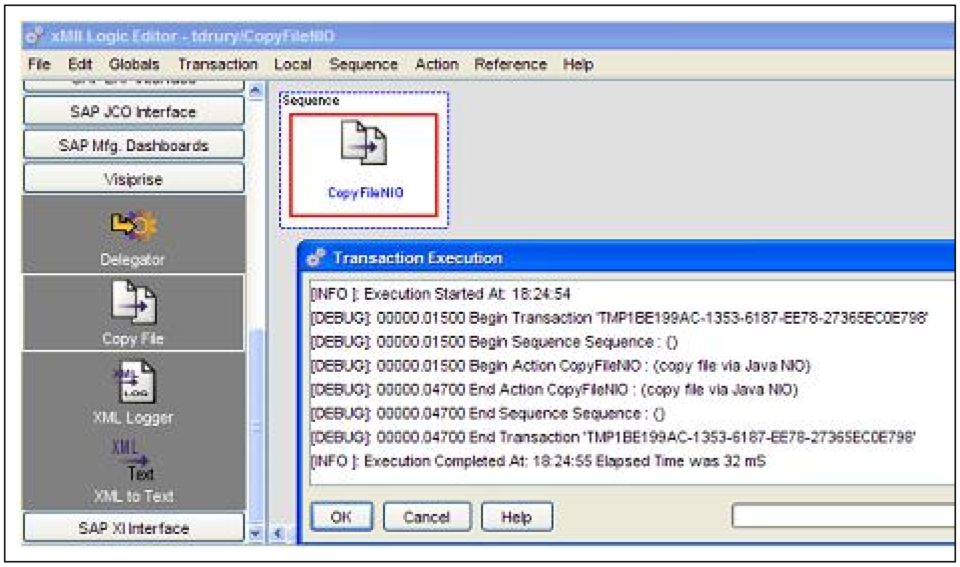
Last, but not least, the action catalog file must be placed in the directory:

- <LighthammerHome>\Xacute\Components

Restart ServletExec and you should be ready to use your new xMII actions!

NIO File Copy Action Results

So how well does the CopyFile action perform? See for yourself. Figure 9 shows the results of copying the same 1 MB text file we copied back in Figure 1. The first time I ran it, it took 78ms – the same amount of time the `Text Loader/Text Saver` transaction took on average. Subsequent tests resulted in execution times between 16ms and 32ms which is likely the resolution limit of my PC's clock.

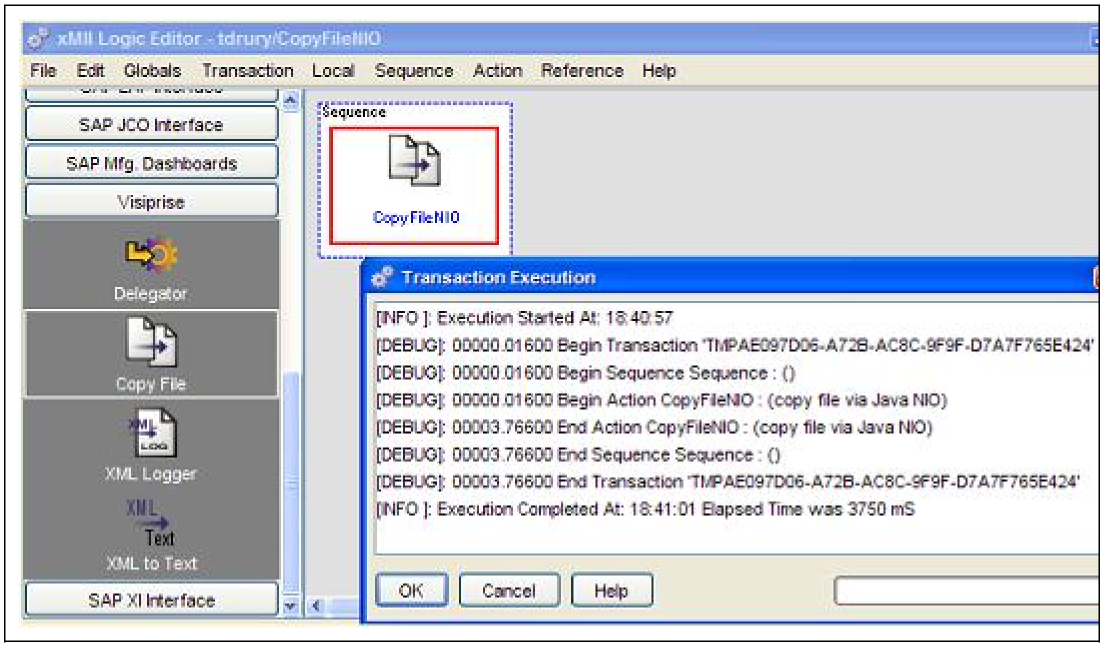


```
[INFO ]: Execution Started At: 18:24:54
[DEBUG]: 00000.01500 Begin Transaction 'TMP1BE199AC-1353-6187-EE78-27365EC0E798'
[DEBUG]: 00000.01500 Begin Sequence Sequence : ()
[DEBUG]: 00000.01500 Begin Action CopyFileNIO : (copy file via Java NIO)
[DEBUG]: 00000.04700 End Action CopyFileNIO : (copy file via Java NIO)
[DEBUG]: 00000.04700 End Sequence Sequence : ()
[DEBUG]: 00000.04700 End Transaction 'TMP1BE199AC-1353-6187-EE78-27365EC0E798'
[INFO ]: Execution Completed At: 18:24:55 Elapsed Time was 32 mS
```

Figure 9.

The NIO-based `CopyFile` action required between 16ms and 32ms to copy a 1 MB text file.

Figure 10 shows the results of copying the 100 MB text file. The first execution of the transaction took around 7 seconds and each test thereafter took less than 4 seconds. The `Text Loader/Text Saver` transaction test required 16 seconds to copy the 100 MB file.



```
[INFO ]: Execution Started At: 18:40:57
[DEBUG]: 00000.01600 Begin Transaction 'TMPAE097D06-A72B-AC8C-9F9F-D7A7F765E424'
[DEBUG]: 00000.01600 Begin Sequence Sequence : ()
[DEBUG]: 00000.01600 Begin Action CopyFileNIO : (copy file via Java NIO)
[DEBUG]: 00003.76600 End Action CopyFileNIO : (copy file via Java NIO)
[DEBUG]: 00003.76600 End Sequence Sequence : ()
[DEBUG]: 00003.76600 End Transaction 'TMPAE097D06-A72B-AC8C-9F9F-D7A7F765E424'
[INFO ]: Execution Completed At: 18:41:01 Elapsed Time was 3750 mS
```

Figure 10.
The NIO-based `FileCopy` action required only 4 seconds to copy a 100 MB text file.

However, the real problem with copying the 100 MB file with the `Text Loader`/`Text Saver` transaction was its insatiable appetite for heap. Figure 11 shows my NIO-based `FileCopy` action's heap usage - or lack thereof. The `FreeMemory` graph didn't move from its initial starting point of 2.777 MB throughout all my FileCopy tests.

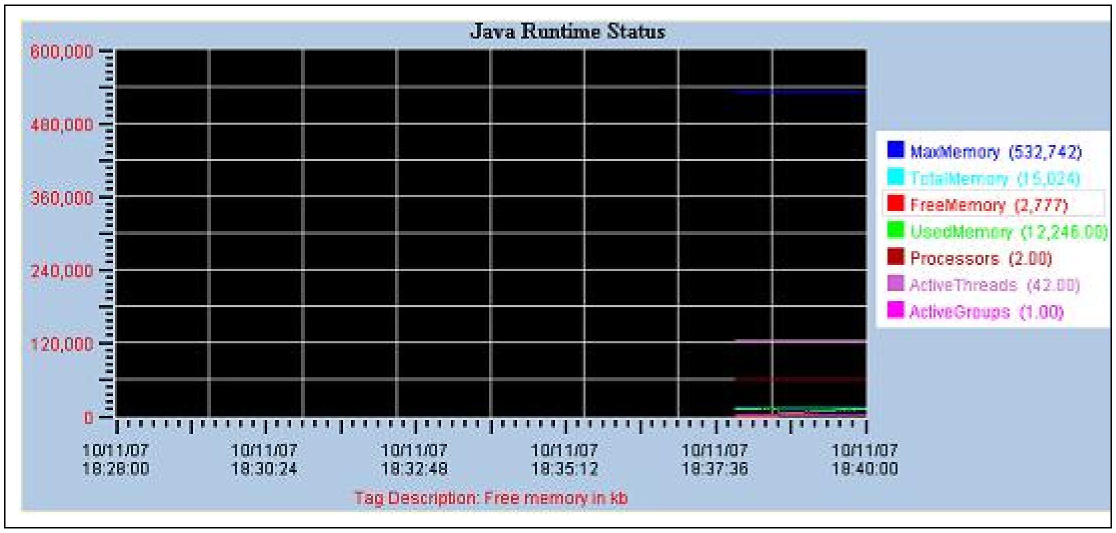
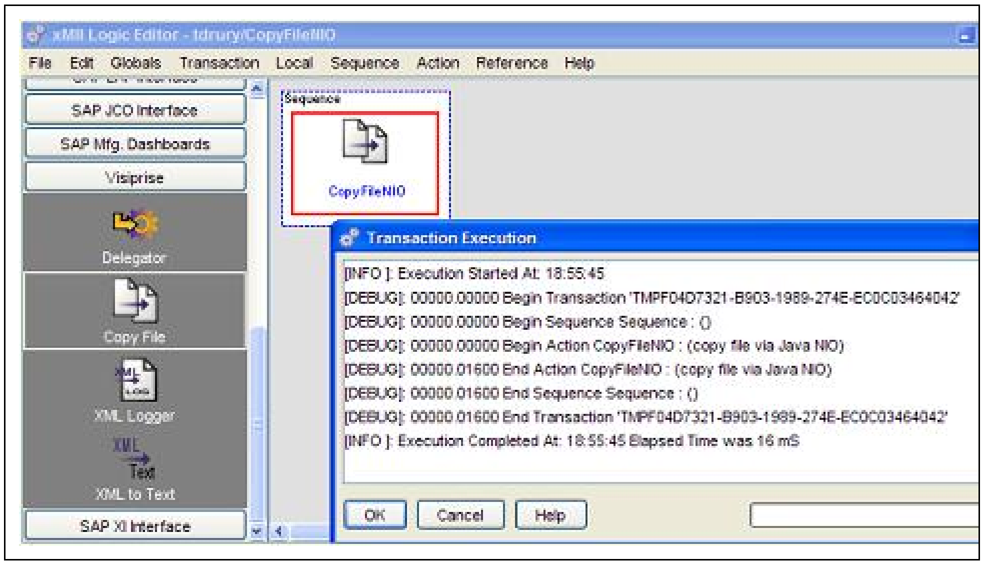


Figure 11.
`FileCopy` used no discernable heap storage while copying a 100 MB file.

The 1 MB binary file copying test results shouldn't surprise anyone (see Figure 12). It took the same amount of time to run – between 16ms and 32ms – as the 1 MB text file copying test. This is because NIO sees no difference between a binary file and a text file and it doesn't perform any sort of binary-to-text conversion.



```
[INFO ]: Execution Started At: 18:55:45
[DEBUG]: 00000.00000 Begin Transaction 'TMPF04D7321-B903-1989-274E-EC0C03464042'
[DEBUG]: 00000.00000 Begin Sequence Sequence : ()
[DEBUG]: 00000.00000 Begin Action CopyFileNIO : (copy file via Java NIO)
[DEBUG]: 00000.01600 End Action CopyFileNIO : (copy file via Java NIO)
[DEBUG]: 00000.01600 End Sequence Sequence : ()
[DEBUG]: 00000.01600 End Transaction 'TMPF04D7321-B903-1989-274E-EC0C03464042'
[INFO ]: Execution Completed At: 18:55:45 Elapsed Time was 16 mS
```

Figure 12.

As with the 1 MB text file test, the NIO-based `FileCopy` action required between 16ms and 32ms to copy a 1 MB binary file. This isn't surprising since a text file is essentially a binary file.

The lack of binary-to-text and text-to-binary conversion is evident in the CPU load during the test (see Figure 13). In fact, I ran the test a dozen or more times during this period in an attempt to get the CPU graph to show any sort of obvious activity.

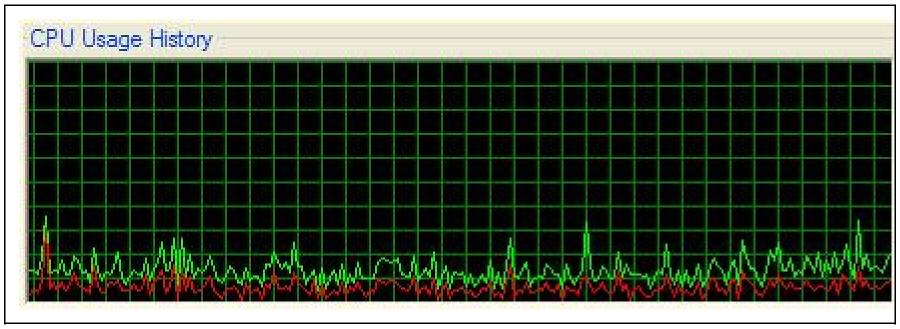


Figure 13.

Believe it or not, I ran the 1 MB binary file copy test with the `FileCopy` action a couple dozen times within the span of this CPU graph snapshot. Can you tell?

More to come…

With the fundamentals of writing xMII actions covered, what's next? In the next installment, I'll uncover the details of an action which extends xMII's existing `Call Transaction` action and eliminate one of its limiting deficiencies. This custom action also has the pleasant side-effect of imbuing xMII with an interface-based development paradigm which greatly improves the efficacy of multi-developer teams.

Related Content

- SDN xMII Forum
- SDN xMII wiki
- xMII 11.5 Help
- Java Swing Tutorial

Disclaimer and Liability Notice