Getting Started With STAX 3

Getting Started With STAX 3

Version 3.0.2

Document Owner: David Bender

1. Introduction
    - 1.1. Overview
        - 1.1.1. Programming Language
        - 1.1.2. Execution Engine
        - 1.1.3. STAX Monitor
    - 1.2. Technologies
        - 1.2.1. STAF
        - 1.2.2. XML
        - 1.2.3. Python
        - 1.2.4. Java
    - 1.3. Concepts
        - 1.3.1. STAX Elements
2. STAX Machine Environment
    - 2.1. Typical Setup
3. Configuring STAX
    - 3.1. Requirements
    - 3.2. Download
    - 3.3. Unpackage
    - 3.4. Configure
    - 3.5. Running the STAX Service
    - 3.6. STAX JVM Log
4. Running the STAX Monitor
    - 4.1. Starting the STAX Monitor
    - 4.2. Running a sample STAX job
5. Understanding XML
    - 5.1. XML Elements
    - 5.2. XML Attributes
6. Understanding Python
    - 6.1. Python variable names
    - 6.2. Using Python variables
    - 6.3. Python Lists
    - 6.4. Python Dictionaries
    - 6.5. References
7. Writing and Executing STAX Jobs
    - 7.1. Basic STAX job template
    - 7.2. Starting a process
    - 7.3. Starting a stafcmd
    - 7.4. Checking the stafcmd return code and the result
    - 7.5. Checking the process return code and the result
    - 7.6. Using script elements to define Python variables
    - 7.7. Specifying scripts in the STAX Monitor
    - 7.8. Adding parameters to a function
    - 7.9. Adding logging to the STAX job
    - 7.10. Importing and calling a STAX function

7.11. Adding execution control into a STAX job
7.12. Running tasks in parallel
7.13. Looping in a STAX job
7.14. Adding testcases into a STAX job
7.15. Adding time constraints into a STAX job
7.16. Sending email in a STAX job
7.17. Starting STAX jobs via EventManager
7.18. Starting STAX jobs via Cron
8. Appendix: Sample STAX jobs
8.1. DoesNothing.xml
8.2. RunNotepadProcess.xml
8.3. RunDelayRequest.xml
8.4. CheckSTAFCmdRC.xml
8.5. RunTestProcess.xml
8.6. UsingScripts.xml
8.7. FunctionParameters.xml
8.8. FunctionParametersLogging.xml
8.9. ImportFunction.xml
8.10. Block.xml
8.11. Parallel.xml
8.12. Loop.xml
8.13. Testcase.xml
8.14. Timer.xml
8.15. Email.xml
9. End of Document

1. Introduction

1.1. Overview
1.1.1. Programming Language
1.1.2. Execution Engine
1.1.3. STAX Monitor
1.2. Technologies
1.2.1. STAF
1.2.2. XML
1.2.3. Python
1.2.4. Java
1.3. Concepts
1.3.1. STAX Elements

1.1. Overview

1.1.1. Programming Language
1.1.2. Execution Engine
1.1.3. STAX Monitor

STAX is an XML-based execution engine which is implemented as an external Java STAF service. STAX was designed to make it significantly easier to automate the workflow of your tests and test environments.

STAX can be used to automate any task. For example, some teams use STAX to automate their product build process. Other teams use STAX to automate a variety of types of tests, from unit test, Function Verification Test, System Verfication Test, etc.

This document will guide you through many common tasks that are performed when using STAX. For more detailed information on STAX, see the STAX User's Guide at http://staf.sourceforge.net/getstax.php.

Note that this document is based on STAX V3.0.0. Older releases of STAX may not have the same functionality that is described in this document.

Before reading this document you should already be familiar with the Software Testing Autoamtion Framework (STAF), or at least have read the "Getting Started with STAF" document.

STAX is comprised of three components: A Programming Language, an Execution Engine, and the STAX Monitor.

1.1.1. Programming Language

STAX provides an XML-based programming language, which in many ways provides capabilities similar to other programming languages, but which is specifically designed for automation. Programs that are written in this language are called STAX jobs.

1.1.2. Execution Engine

The STAX service is the STAX execution engine (similar to interpreters for other programming or scripting languages) which takes a STAX XML job as input, and manages the execution and runtime behavior of the job.

1.1.3. STAX Monitor

The STAX Monitor is a GUI application that provides a dynamically updated view of your jobs as they are executing.

1.2. Technologies

1.2.1. STAF
1.2.2. XML
1.2.3. Python
1.2.4. Java

The are four core technologies used in STAX: STAF, XML, Python, and Java.

1.2.1. STAF

STAF provides the infrastructure upon which STAX builds. This means that all of the STAF services can be accessed and used within your STAX jobs.

1.2.2. XML

The STAX programming language is based on XML. This provides built-in structure to your jobs, as well as providing a set of existing tools for use in constructing your jobs, such as XML (aware) editors and XSLT.

1.2.3. Python

The STAX programming language builds on the Python scripting language for variable and expression evaluation. This means that in addition to the STAX programming language, you can also call any standard Python libraries or custom-written Python libraries.

1.2.4. Java

STAX uses Jython to execute the Python code in STAX jobs. Jython is an implementation of the Python scripting language that is written in 100% pure Java. This means that you can also execute Java classes (standard or custom-written) within your STAX jobs.

1.3. Concepts

1.3.1. STAX Elements

1.3.1. STAX Elements

A STAX Element is a node in a STAX XML job. A STAX Job is comprised of a structured hierarchy of STAX Elements. Some of the items that STAX Elements can represent are: data to be used during the job, commands/processes to be executed, definitions of the logic and control flow within the job, and wrappers such as functions and blocks that encompass other STAX Elements.

Processes

A STAX Process element really defines the execution information for a STAF PROCESS START command. A process element specifies a command to be executed and the machine where it should run.

Commands

A STAX STAF Command (stafcmd) element defines execution information for all other STAF service commands. A stafcmd element specifies the STAF service and request to be executed and the machine where it should run.

Scripts

STAX Script elements are used to define Python code, usually for variable and expression evaluation.

Groups

STAX can execute groups of STAX Elements sequentially or in parallel. When Elements are executed in parallel, STAX will run each of the Elements on a separate thread.

Loops

Loop Elements are available which allow a STAX Element to be executed repeatedly. In addition, there are Iterate Elements (both sequential and parallel) which allow a STAX Element to be executed repeately while stepping through a list of data for each iteration.

Wrappers

STAX has several Wrapper Elements which simply provide additional functionality to another STAX Element. These Wrapper Elements can denote Testcases (with testcase status), Blocks (for which execution control can be manipulated), Timers (for time-based execution control), and Functions (for splitting jobs into pieces that have a well-defined role).

Functions

Functions are a nearly universal program structuring-device. Functions serve two primary development roles: code reuse and procedural decomposition. Functions are the simplest way to package logic you may wish to use in more than one place and more than one time. Functions allow you to group and parameterize sections of XML to be used arbitrarily many times later.

Sub-Jobs

STAX provides a Job Element so that sub-jobs can be executed within a parent job with synchronized completion as well as providing access to the sub-job result.

Logic Flow

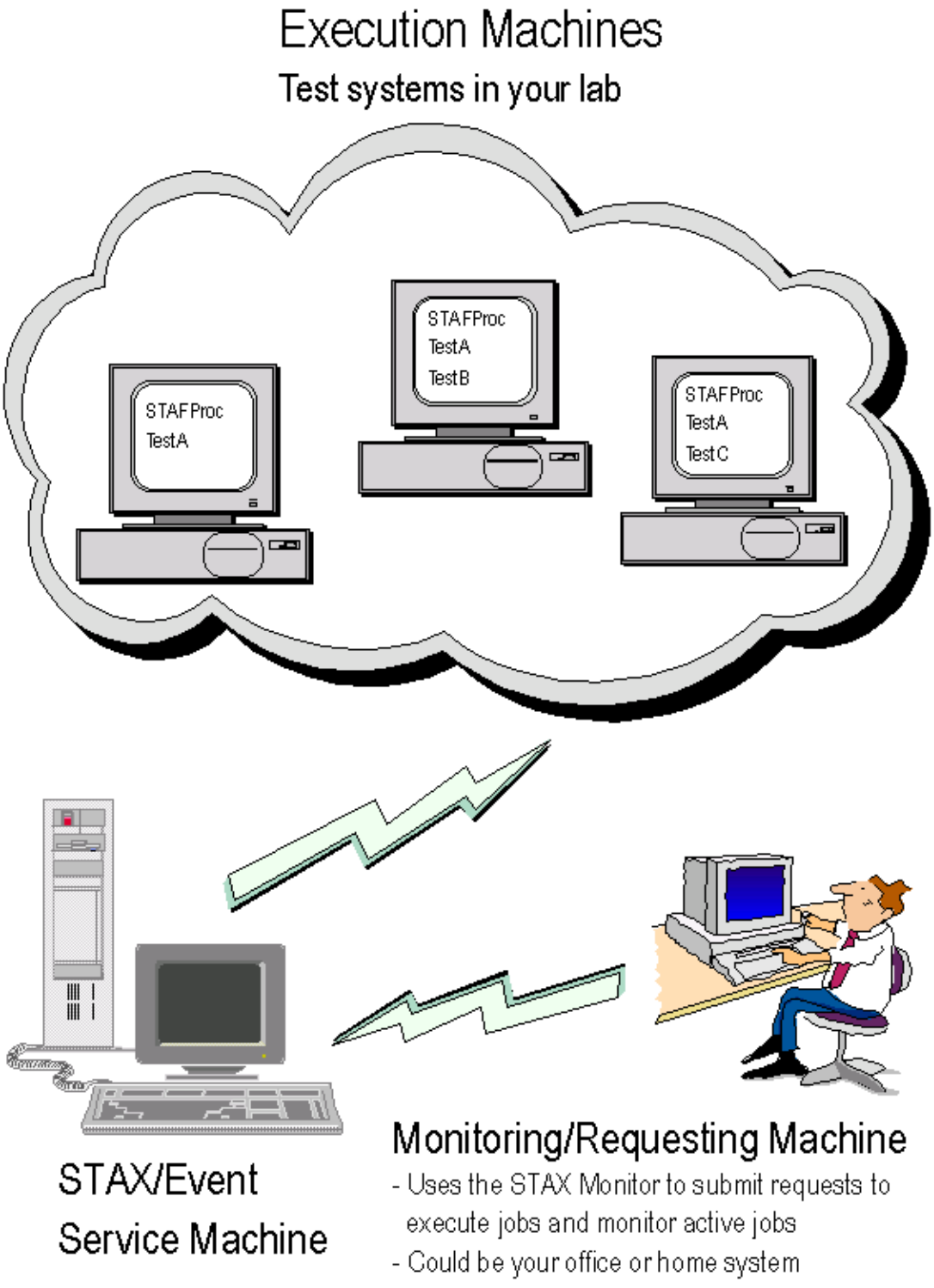
STAX provides an If Element which can evaluate conditions using Python to determine logic flow within STAX jobs, thus allowing job flow to branch dynamically.

2. STAX Machine Environment

2.1. Typical Setup

2.1. Typical Setup

Figure 1.



- Could be your office or home system

Typically you will configure STAX, along with other Java STAF services, such as Event, EventManager, Cron, and Email, on a single server-type machine. We will call this machine the STAX service machine. This machine must be up and running STAF while STAX jobs are executing on it.

The other machines in your STAF environment do not need to be running the STAX service. However, since the STAX service machine will be executing processes on these other machines, each of the other machines must grant the STAX service machine a trust level of 5, so that the PROCESS START requests have the required trust level.

The STAX Monitor is a distributed Java application that displays information about STAX jobs that the STAX service machine is currently running. The STAX Monitor application can be executed on any machine in your STAF environment (including the STAX service machine itself). The machine on which you are running the STAX Monitor must grant the STAX Service machine a trust level of 3.

Note that since both the STAX Service and the STAX Monitor are written in Java, you need to install a JVM (Sun or IBM) on the STAX Service and any machines on which you want to run the STAX Monitor. The STAX Service requires Java 1.4 or later and the STAX Monitor requires Java 1.3 or later. IBM employees must download the Sun or IBM JVM from the internal JIM site. Non-IBM users can download from java.sun.com. We recommend that you install the most recent fixpack for the JVM you want to use, so that you will have all of the latest fixes.

3. Configuring STAX

3.1. Requirements
3.2. Download
3.3. Unpackage
3.4. Configure
3.5. Running the STAX Service
3.6. STAX JVM Log

3.1. Requirements

- Java 1.4 or later must be installed.
- STAF V3 must be installed. See the installation instructions in the STAF documentation. Verify that the CLASSPATH environment variable contains the JSTAF.jar file (e.g. C:\STAF\bin\JSTAF.jar or /usr/local/staf/lib/JSTAF.jar). JSTAF.jar contains the STAF Java APIs to communicate with STAF from Java programs and is required to register STAF services written in Java.

3.2. Download

The STAX service is an external Java STAF service that you will download to your STAX service machine. You can download the latest version of STAX from http://staf.sourceforge.net/getstax.php. We provide a .zip file (for Windows) and a .tar file (for Unix). Note that the contents of the .zip and .tar files are identical.

3.3. Unpackage

It is recommended that you unzip/untar the STAX service file to a directory named "services" within your root STAF directory (usually C:/STAF/services on Windows or /usr/local/staf/services on Unix) on your STAX service machine.

After you unzip/untar the STAX service file, you will have the following files:

stax/STAX.jar	This is the STAX service jar file. This jar file does not need to be in your CLASSPATH.
stax/STAXMon.jar	This is the jar file for the STAX Monitor application. This jar file does not need to be in your CLASSPATH.
stax/STAFEvent.jar	This is the Event service jar file. The Event service can be downloaded from the staf.sourceforge.net website, but for convenience it is included in the STAX download. This jar file does not need to be in your CLASSPATH
stax/STAXDoc.jar	This is the jar file for the STAX Doc application. This jar file does not need to be in your CLASSPATH.
stax/samples/sample1.xml	A sample STAX job which demonstrates some of the capabilities of STAX.

stax/samples/FunctionList.xsl	An XSL Stylesheet that can be used to generate HTML documents that describe the functions in your STAX jobs.
stax/libraries/STAXUtil.html	Documentation for the library of common functions that you can call in your STAX jobs.
stax/libraries/STAXUtil.xml	A library of common functions that you can call in your STAX jobs.
stax/docs/History	A text file containing the complete history of STAX bugs/features.
stax/docs/staxgs.pdf	The Getting Started with STAX Guide.
stax/docs/staxug.pdf	The STAX User's Guide.
stax/docs/STAXDoc.pdf	The STAX Doc User's Guide. STAXDoc is a tool that lets you generate documentation for your STAX jobs.
stax/ext/stax-extensions.dtd	The DTD for a STAX extensions xml file. This DTD is used for files specified using the EXTENSIONXMLFILE parameter when registering the STAX service with extensions.
stax/readme.1st	The Readme file for this STAX service version.

3.4. Configure

Now it's time to configure the STAX service on the STAX service machine. Using any text editor, open the STAX service machine's STAF.cfg file (typically C:/STAF/bin/STAF.cfg on Windows or /usr/local/staf/bin/STAF.cfg on Unix).

You will need to configure the STAX and Event services, as well as increase the default setting for MAXQUEUESIZE (if you intend to run the STAX Monitor on the STAX service machine).

```
SERVICE STAX LIBRARY JSTAF EXECUTE {STAF/Config/STAFRoot}/services/stax/STAX.jar \
    OPTION J2=-Xmx384m
SERVICE EVENT LIBRARY JSTAF EXECUTE \
    {STAF/Config/STAFRoot}/services/stax/STAFEvent.jar
SET MAXQUEUESIZE 10000
```

It is recommended that you always specify the "J2=-Xmx384m" option for the STAX service JVM (changing the 384m value for a value appropriate for the amount of RAM on your STAX service machine). This option increases the maximum Java heap (memory) size and can alleviate OutOfMemory errors in the STAX JVM.

In this example we are assuming that the STAX service is the first service in your STAF.cfg file. If it was not, since you are changing the JVM options, you would also need to include the JVMName option, to specify a unique name for the STAX service's JVM.

Note that these service configuration statements assume that you have the JVM bin directory in your System PATH, so that when STAFProc starts, it will be able to start the JVM for these services. If you do not want to include the JVM bin directory in your PATH, then you can use the "OPTION JVM=xxx" to specify which Java executable to use for the services.

At this point, you should configure the additional STAF Java services (such as EventManager, Cron, and Email) by downloading them from http://staf.sourceforge.net/getservices.php, unpackaging them, and configuring them in your STAF.cfg as well:

```
SERVICE STAX LIBRARY JSTAF EXECUTE {STAF/Config/STAFRoot}/services/stax/STAX.jar \
    OPTION J2=-Xmx384m
SERVICE EVENT LIBRARY JSTAF EXECUTE \
    {STAF/Config/STAFRoot}/services/stax/STAFEvent.jar
SERVICE EVENTMANAGER LIBRARY JSTAF EXECUTE \
    {STAF/Config/STAFRoot}/services/eventmanager/STAFEventManager.jar
SERVICE CRON LIBRARY JSTAF EXECUTE {STAF/Config/STAFRoot}/services/cron/STAFCron.jar
SERVICE EMAIL LIBRARY JSTAF EXECUTE \
    {STAF/Config/STAFRoot}/services/email/STAFEmail.jar \
    PARMS MAILSERVER na.relay.ibm.com
SET MAXQUEUESIZE 10000
```

Note that the "na.relay.ibm.com" mail server is only avaiable within IBM. If you are not an IBM employee, you will need to change this to a valid SMTP server within your company.

Save the changes you have just made to the STAF.cfg file, and restart STAFProc in order to pick up the changes.

3.5. Running the STAX Service

Open a command prompt, and type **staf local service list**

Verify that the STAX and Event services (as well as EventManager, Cron, and Email) are listed in the output:

```
$ staf local service list
Response
--------
Name         Library    Executable
------------ ---------- ------------------------------------------------------
CRON         JSTAF      C:/STAF/services/cron/STAFCron.jar
DELAY        <Internal> <None>
DIAG         <Internal> <None>
ECHO         <Internal> <None>
EMAIL        JSTAF      C:/STAF/services/email/STAFEmail.jar
EVENT        JSTAF      C:/STAF/services/stax/STAFEvent.jar
EVENTMANAGER JSTAF      C:/STAF/services/eventmanager/STAFEventManager.jar
FS           <Internal> <None>
HANDLE       <Internal> <None>
HELP         <Internal> <None>
LOG          STAFLog    <None>
MISC         <Internal> <None>
PING         <Internal> <None>
PROCESS      <Internal> <None>
QUEUE        <Internal> <None>
SEM          <Internal> <None>
SERVICE      <Internal> <None>
SHUTDOWN     <Internal> <None>
STAX         JSTAF      C:/STAF/services/stax/STAX.jar
TRACE        <Internal> <None>
TRUST        <Internal> <None>
VAR          <Internal> <None>
```

Next, execute **staf local stax version** and make sure that you are running the correct version of STAX:

```
$ staf local stax version
Response
--------
3.0.0
```

Next, execute **staf local stax help** to see all of the command requests that the STAX service accepts:

```
$ staf local stax help
Response
--------
STAX Service Help:

EXECUTE   < <FILE <XML File Name> [MACHINE <Machine Name>]> | DATA <xml data> >
          [JOBNAME <Job Name>] [FUNCTION <Function ID>] [ARGS <Arguments>]
          [SCRIPTFILE <File Name>... [SCRIPTFILEMACHINE <machine name>]]
          [SCRIPT <Python Code>]... [CLEARLOGS [<Enabled | Disabled>]]
          [ HOLD | TEST | WAIT [Timeout] [RETURNRESULT] ]
          [LOGTCELAPSEDTIME <Enabled | Disabled>]
          [LOGTCNUMSTARTS <Enabled | Disabled>]
          [LOGTCSTARTSTOP <Enabled | Disabled>]

GET       DTD
```

```
LIST      JOBS | SETTINGS | EXTENSIONS | EXTENSIONJARFILES |
          JOB <Job ID> <THREADS | PROCESSES | STAFCMDS | SUBJOBS | BLOCKS |
TESTCASES>

QUERY     EXTENSIONJARFILE <Jar File Name> | EXTENSIONJARFILES |
          <JOB ob ID> [THREAD <Thread ID> | PROCESS <Location:Handle> | STAFCMD
          <Request#> | BLOCK <Block Name> | TESTCASE <Test Name>

HOLD      JOB <Job ID> [BLOCK <Block Name>]

RELEASE   JOB <Job ID> [BLOCK <Block Name>]
TERMINATE JOB <Job ID> [BLOCK <Block Name>]

UPDATE    JOB <Job ID> TESTCASE <Testcase name> STATUS <Status>
          [MESSAGE <Message text>] [FORCE]

START     JOB <Job ID> TESTCASE <Testcase name> [KEY <Key>]

STOP      JOB <Job ID> TESTCASE <Testcase name> [KEY <Key>]

SET       [CLEARLOGS <Enabled | Disabled>]
          [LOGTCELAPSEDTIME <Enabled | Disabled>]
          [LOGTCNUMSTARTS <Enabled | Disabled>]
          [LOGTCSTARTSTOP <Enabled | Disabled>]

VERSION   [JYTHON]

HELP
```

You can find more information on these commands in the STAX User's Guide.

3.6. STAX JVM Log

Errors that occur when running the STAX service will be stored in its JVM log. This log is data/STAF/lang/java/jvm/STAFJVM1/JVMLog.1 in your root STAF directory (typically C:/STAF/data/STAF/lang/java/jvm/STAFJVM1/JVMLog.1 for Windows, or /usr/local/staf/data/STAF/lang/java/jvm/STAFJVM1/JVMLog.1 for Unix).

If you edit this file, you should see something similar to:

```
*********************************************************************
*** 20050305-17:24:51 - Start of Log for JVMName: STAFJVM1
*** JVM Executable: java
*** JVM Options   : -Xmx384m
*********************************************************************

Registered Extensions for stax Version 3.0.0:
```

Note that the first time you configure the STAX service, you may see several "*sys-package-mgr*: processing new jar" messages in the JVM log. This is expected behavior, and does not indicate an error.

4. Running the STAX Monitor

4.1. Starting the STAX Monitor
4.2. Running a sample STAX job

4.1. Starting the STAX Monitor

Now you're ready to start the STAX Monitor and run a sample STAX job. From a command prompt, type **cd C:/STAF/services/stax** and then **java -jar STAXMon.jar**. If you would prefer to not have to have the C:/STAF/services/stax directory as your current directory, then fully-qualify the path to the STAXMon.jar file: **java -jar C:/STAF/services/stax/STAXMon.jar**

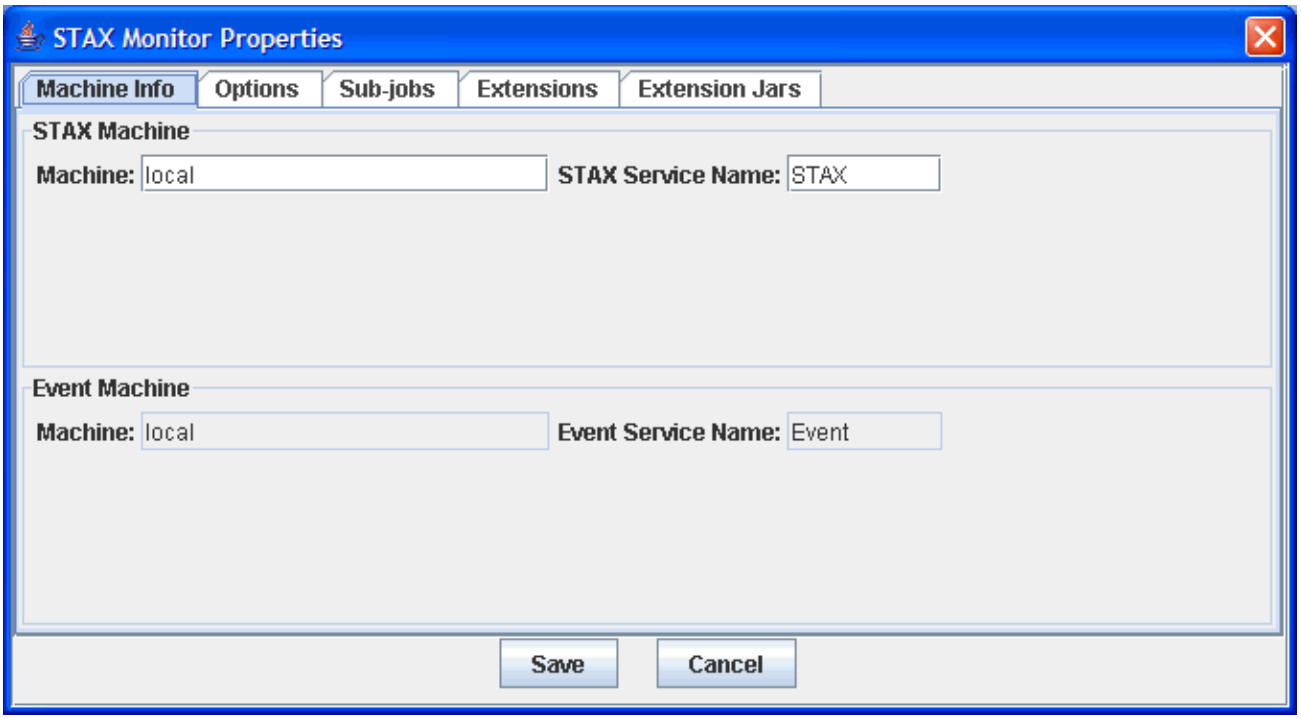
The first thing you will see is the STAX Monitor splash screen, which indicates which version of the STAX Monitor you are running:

Figure 2.



The very first time that you start the STAX Monitor, the STAX Monitor Properties dialog will be displayed:

Figure 3.



On subsequent restarts of the STAX Monitor, you can access the STAX Monitor Properties dialog via the "File" menu in the main STAX Monitor panel.

The most important information in the STAX Monitor Properties dialog is the STAX machine info. If you are running the STAX Monitor on the same machine as the STAX service machine, then leave the STAX Machine field set to "local". If you are running the STAX Monitor on a different machine than the STAX service machine, then change the STAX Machine field from "local" to the hostname of the STAX service machine. In most cases, the STAX service name is "STAX", so you don't need to change that field.

Note that the Event Service machine and name are determined by querying the STAX service machine, which is why these fields are disabled.

Next click on Save to close the STAX Monitor Properties. You will then see the main STAX Monitor panel:

Figure 4.

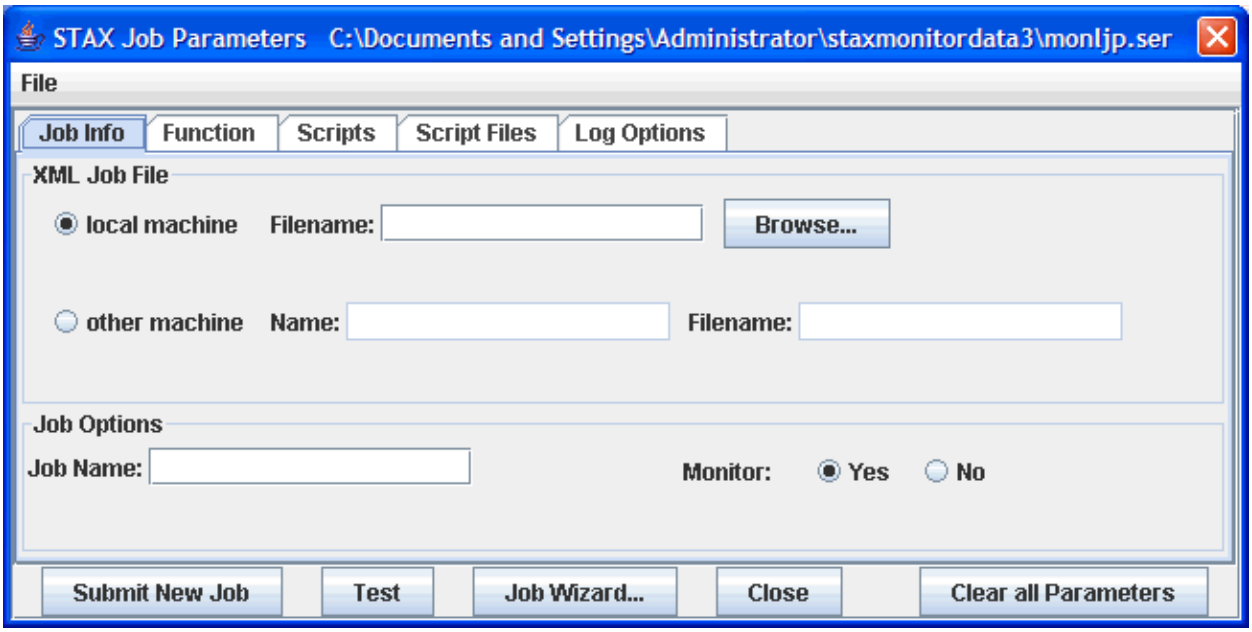


The Active Jobs table shows you all of the STAX jobs that the STAX service on the STAX service machine is running. Since you have not yet submitted any jobs, there are none listed in the table.

4.2. Running a sample STAX job

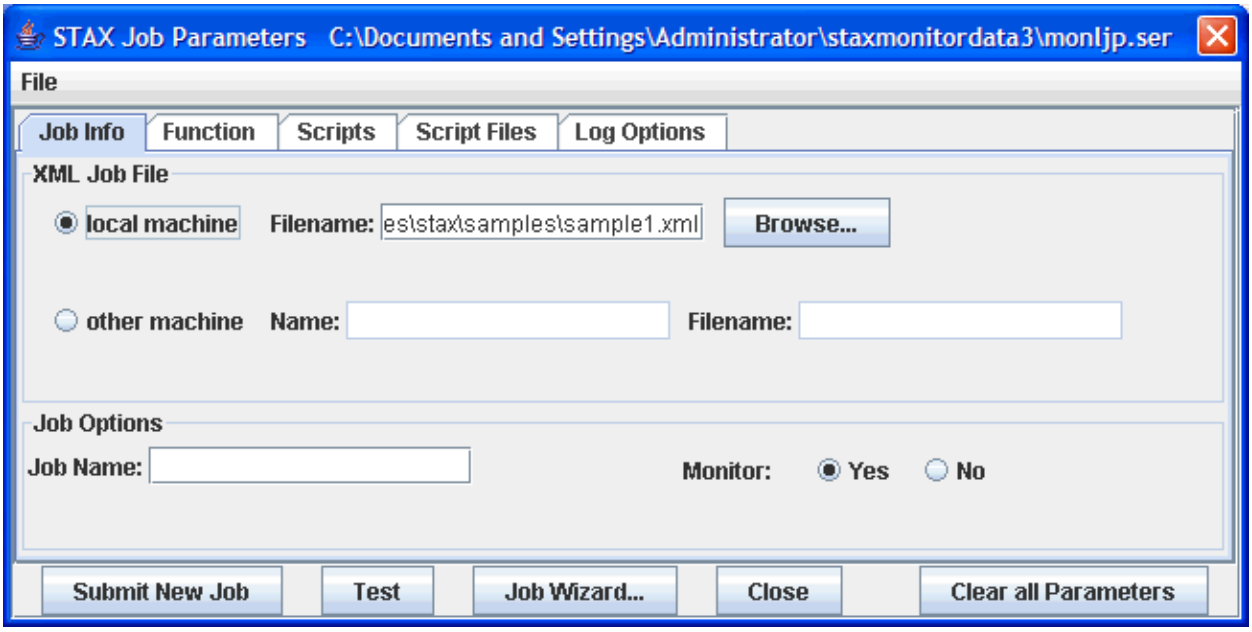
Next you will execute the sample STAX job, sample1.xml. Click on the "Submit New Job..." button in the main STAX Monitor panel. You will then see the "STAX Job Parameters" dialog:

Figure 5.



In this dialog you fill in information about the STAX job that you want to execute. In the "XML Job Info" section, leave "local machine" selected and specify the full path to the sample1.xml file (for example, C:/STAF/services/stax/samples/sample1.xml), or click on the "Browse..." button, and navigate to the samples directory and select sample1.xml.

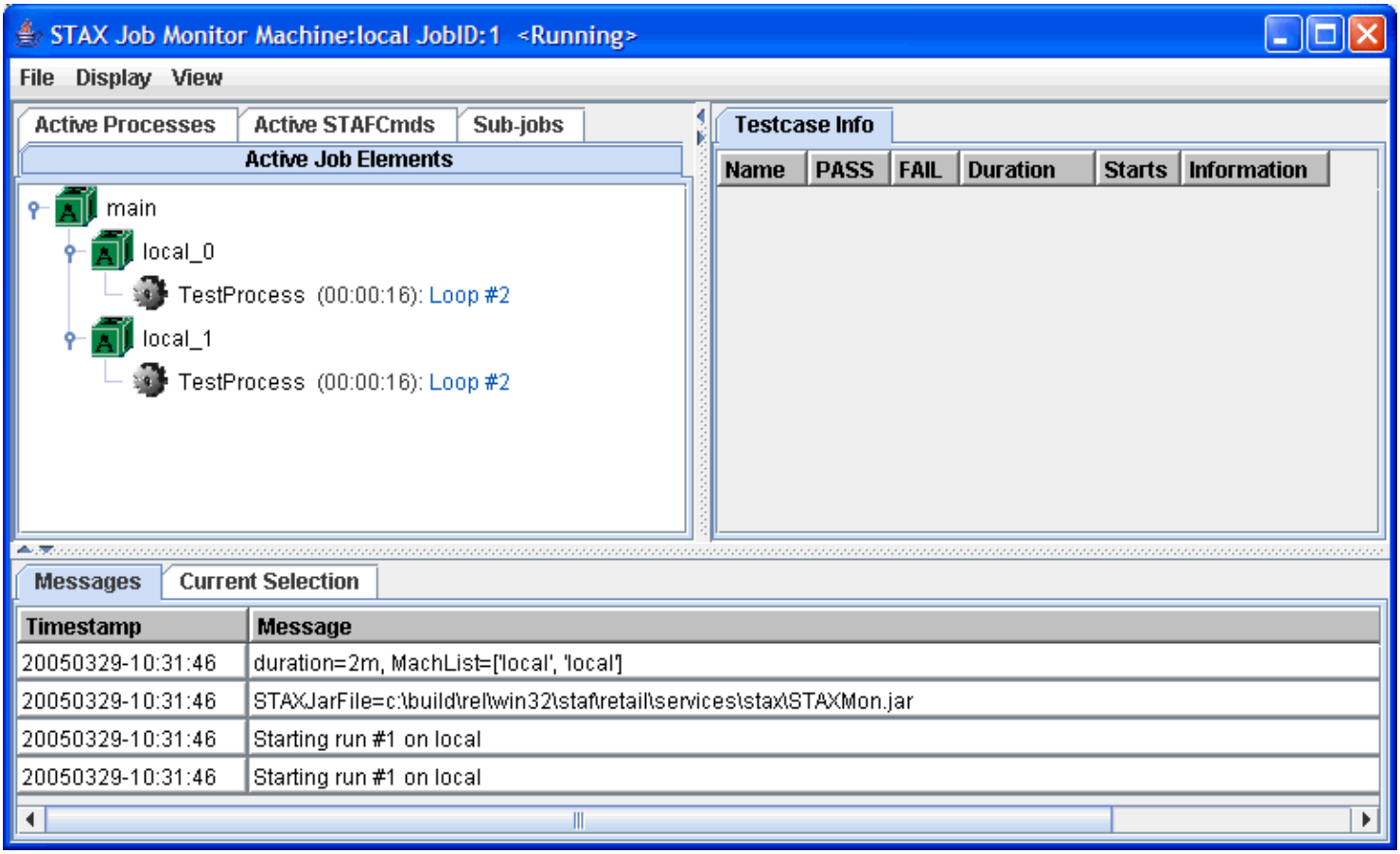
Figure 6.



Next click on the "Submit New Job" button. This will submit an EXECUTE request to the STAX service, passing along the job information you specified.

You will then see a separate window displayed that shows you a dynamic view of your STAX job as it is executing:

Figure 7.

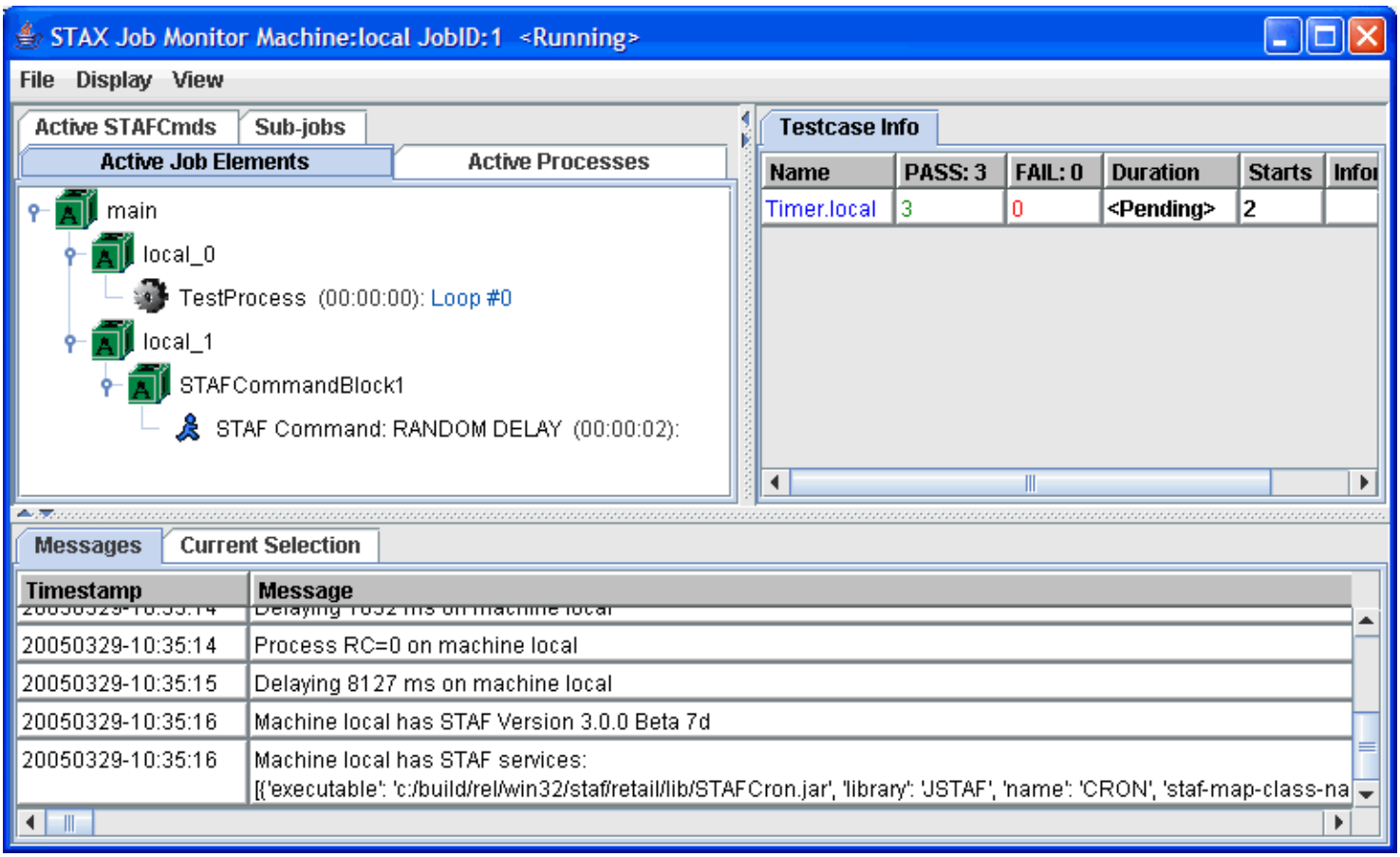


Notice in the title bar that the Job ID is 1. Every time the STAX service is started on the STAX service machine, the Job ID counter will reset to 1.

In the "Active Job Elements" tab, you will see the Processes and STAF Commands that are currently running. Note that they are displayed in a hierarchical format, depending on the blocks that are defined in your STAX job.

As the job executes, you will see that there are testcase passes/fails displayed in the "Testcase Info" tab:

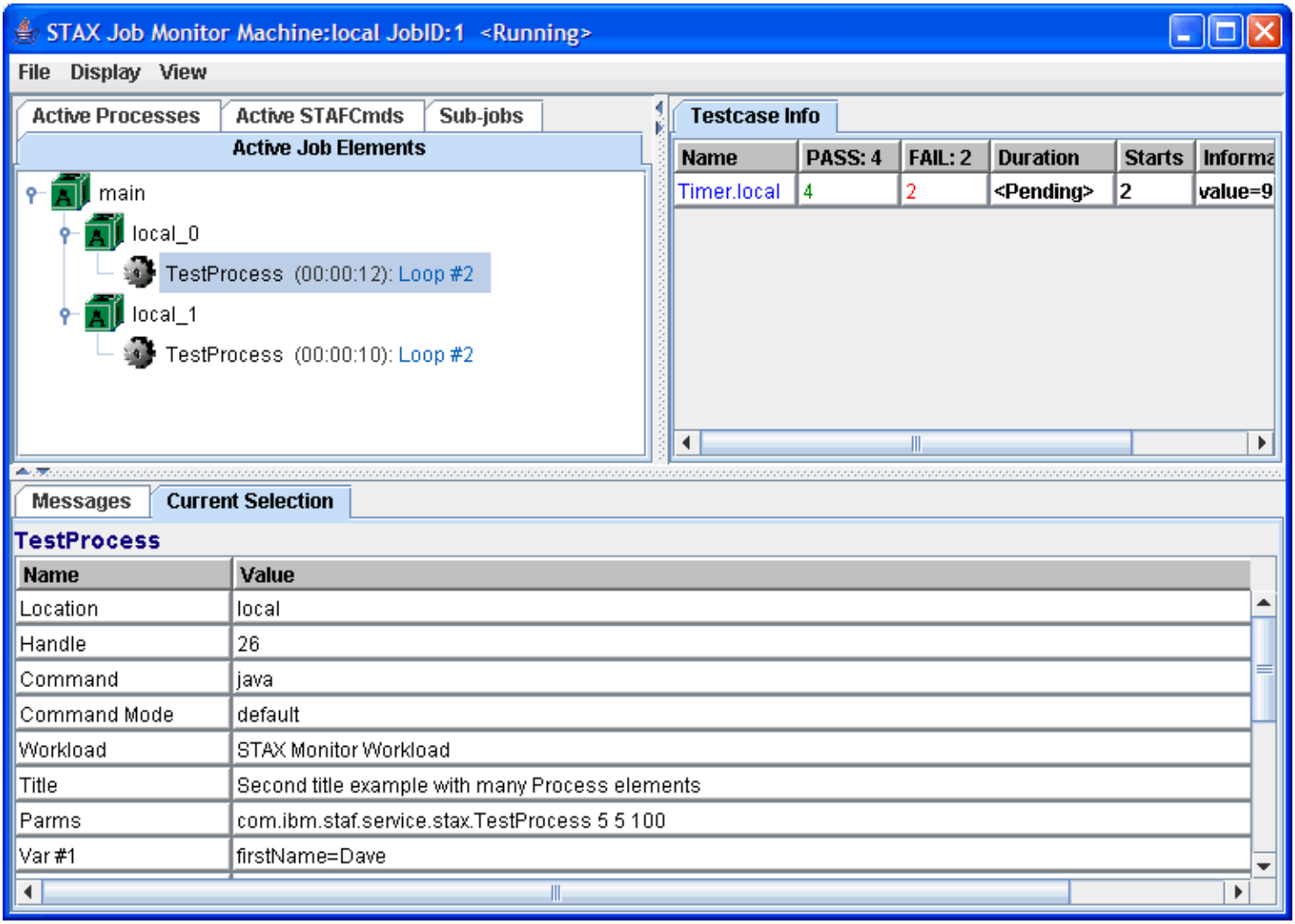
Figure 8.



The "Messages" tab displays informational messages that are written by your STAX job.

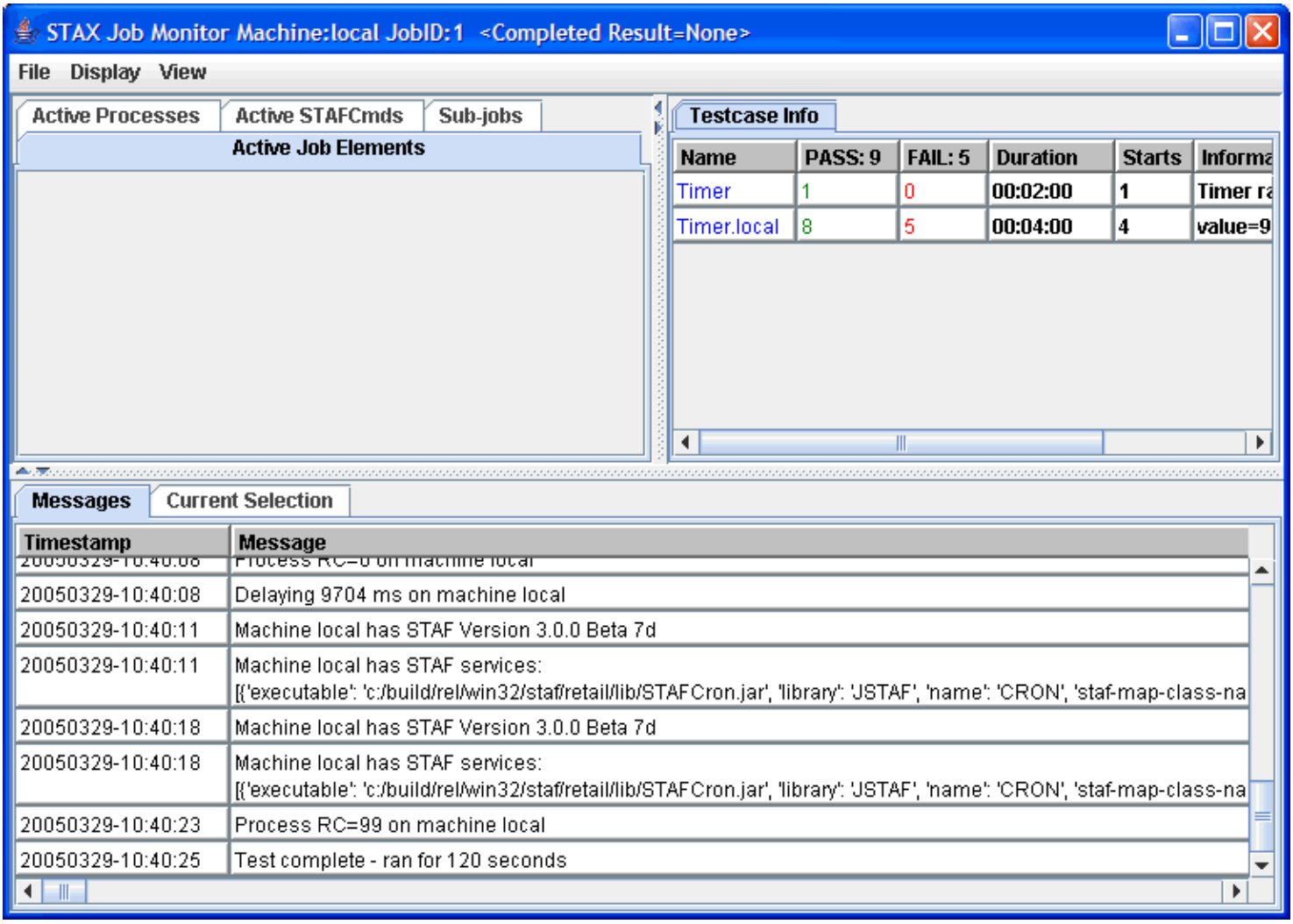
If you click on any entries in the "Active Job Elements" tree, you will see detailed information about that item in the "Current Selection" tab:

Figure 9.



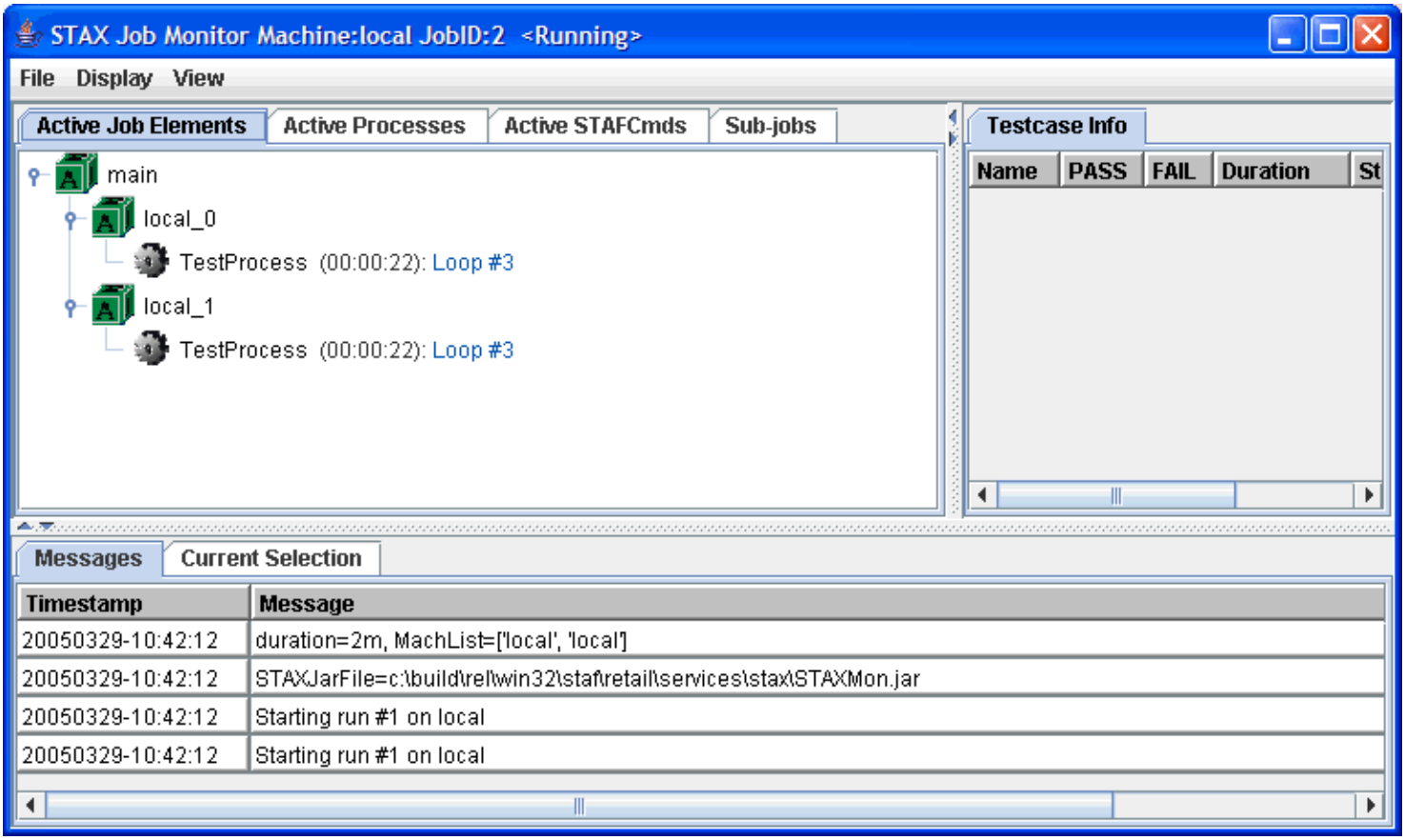
Note that this sample STAX job uses a timer to specify that the job should only execute for 2 minutes, so after 2 minutes the job will complete:

Figure 10.



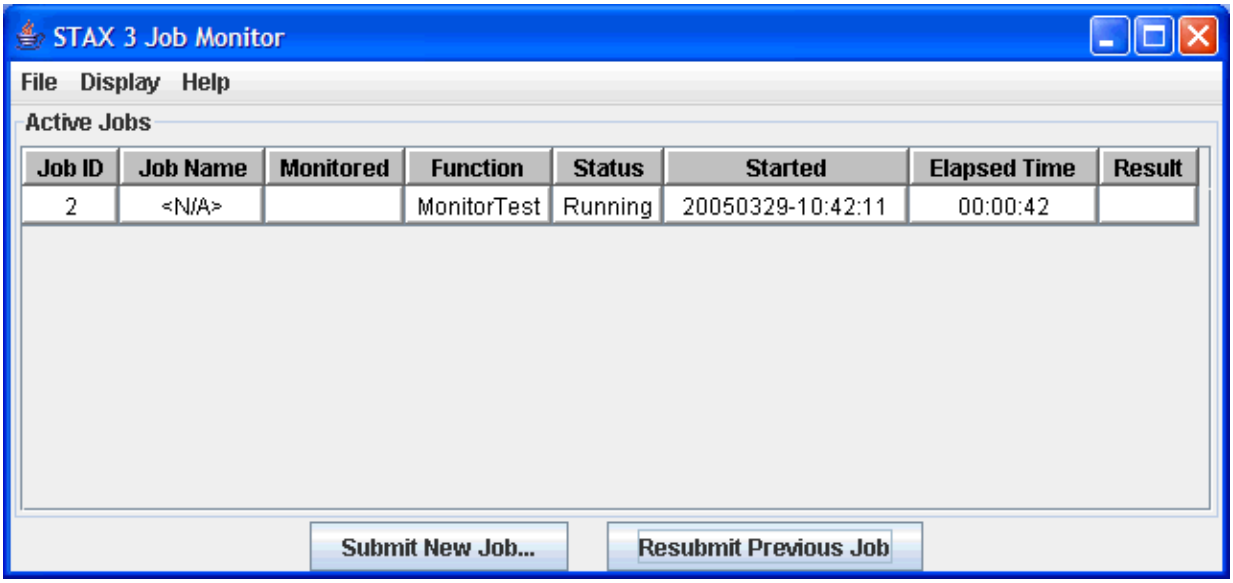
Click on "File" in the menu bar, and then "Exit and Resubmit Job". This will close this window, and submit the same STAX job again. Now you will see a new STAX Monitor window for Job ID 2:

Figure 11.



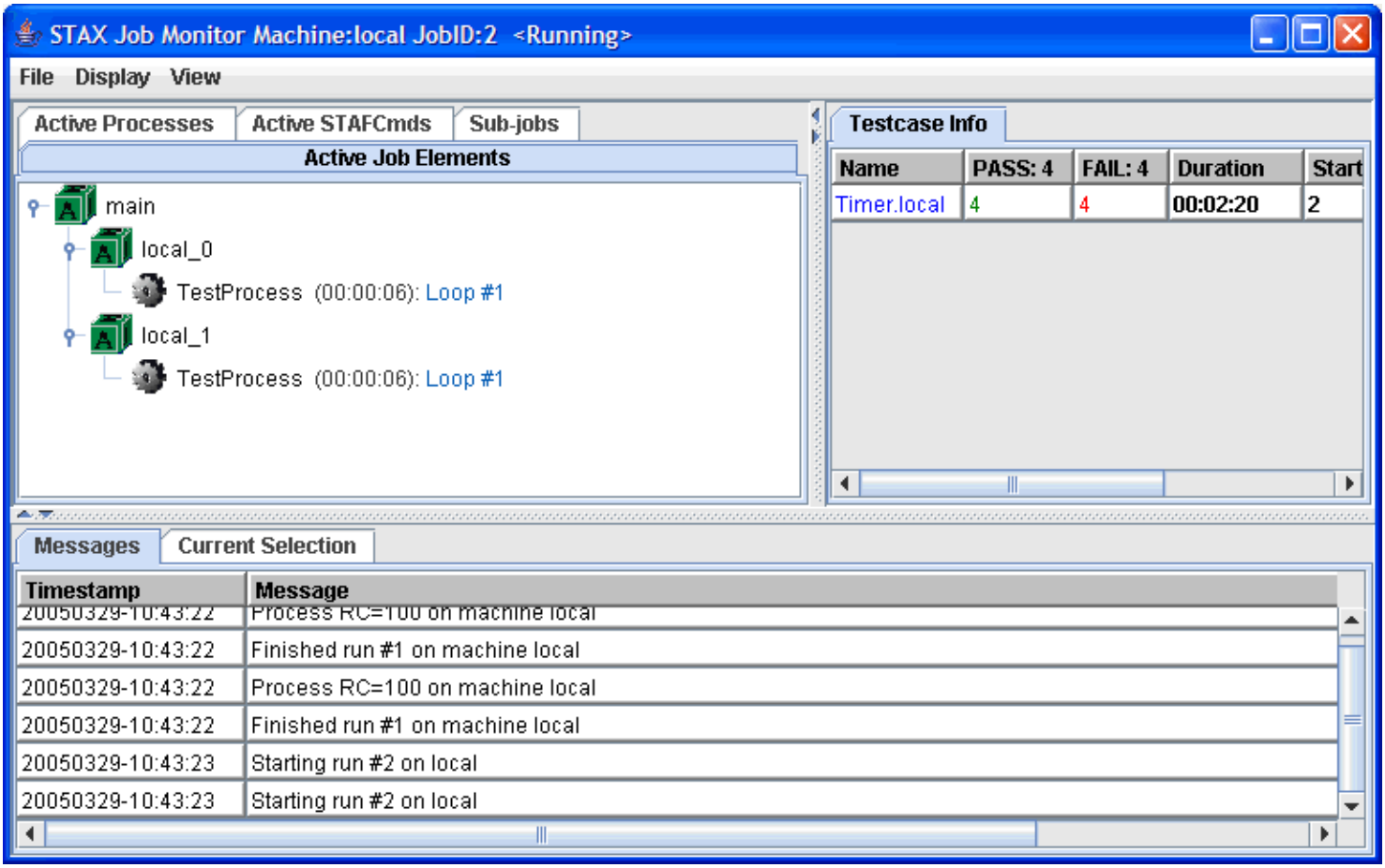
Now close this window and go back to the main STAX Monitor panel:

Figure 12.



Note that closing the STAX Monitor did not have any affect on the STAX Job. The STAX service is continuing to execute the job. Right-click on the Job ID 2 row in the "Active Jobs" table, and select "Start Monitoring". You will see a new window for Job 2 that displays its current execution:

Figure 13.



Now that you are familiar with some of the basics about the STAX Monitor, you are ready to start learning about how to create STAX jobs. But first, since these jobs use XML and Python, you need to learn some basic information about both in the next 2 sections of this document.

5. Understanding XML

5.1. XML Elements
5.2. XML Attributes

5.1. XML Elements

STAX uses XML to describe STAX job definitions. XML documents are made up of XML elements. You create XML elements with an opening tag, such as **<stax>**, followed by the element content (if any), such as text or other elements, and ending with the matching closing tag that starts with </, such as **</stax>**. Note that XML elements names are case-sensitive.

Here is an example of an XML element that contains text:

```
<log>'This is some text'</log>
```

Here is an example of an empty XML element:

```
<nop></nop>
```

Here is another example of an empty element that illustrates that you can close an empty element with />. This example is equivalent to the previous example:

```
<nop/>
```

Here is an example of an element that contains 2 sub-elements, each of which contain text:

```
<process>
  <location>'machineA'</location>
  <command>'C:/tests/stress1.exe'</command>
</process>
```

Sub-elements are indented to the right of the containing element. The number of indentation spaces is up to you, but the recommendation is 2 spaces.

XML comments start with <!-- and end with -->. For example:

```
<!-- Assign a list of machines to variable machList -->
```

Every STAX job has a root element, the **stax** element, so each STAX job will have the following basic template:

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<!DOCTYPE stax SYSTEM "stax.dtd">

<stax>
   .
   .
   .
</stax>
```

The first 2 lines are always the same. Within the <stax> element, you will define the workflow of your job.

5.2. XML Attributes

XML Attributes are name-value pairs that allow you to specify additional data in opening tags. To assign a value to an attribute, you use an equal sign. You must enclose the attribute value in quotes (we recommend using double quotes). Here is an example of specifying an attribute for the testcase element:

```
<testcase name="'Test1'">
```

Note the attribute name is **name**, the value is '**Test1**', and the value is enclosed in double quotes. You can also specify multiple attributes:

```
<testcase name="'Test1'" mode="'strict'">
```

6. Understanding Python

6.1. Python variable names
6.2. Using Python variables
6.3. Python Lists
6.4. Python Dictionaries
6.5. References

STAX uses Python for variable and expression evaluation. STAX uses Jython to execute the Python code. Jython is an implementation of the Python scripting language written in 100% pure Java that runs under any compliant JVM. Using Jython, you can write Python code that interacts with any Java code.

You use **script** elements within your STAX jobs to define Python variables and execute Python code. However, also note that in most cases, all of the element content and element attributes in your STAX jobs will also be evaluated as Python code.

6.1. Python variable names

STAX variable names must follow the Python variable naming conventions:

- Variable names must start with an underscore or letter, and be followed by any number of letters, numbers, or underscores.
- Python variables are case-sensitive. For example, **X** and **x** refer to two different variable names.
- Python reserved words cannot be used as variable names.
- STAX reserved words cannot be used as variable names. The STAX reserved words are **RC**, any word beginning with **STAX**, and any word beginning with **STAF**.

6.2. Using Python variables

Python string constants can be enclosed in single or double quotes, which allows embedded quotes of the opposite flavor. For example, the following two lines do exactly the same thing in a STAX job. They assign a literal string "CoolTest1" to the value of a variable named testName:

```
<script>testName = "CoolTest1"</script>
<script>testName = 'CoolTest1'</script>
```

However, the following line is not the same. It assigns the value of a variable named CoolTest1 to the value of a variable named testName. If this was not what you intended and a variable named CoolTest1 does not exist, you will get an error.

```
<script>testName = CoolTest1</script>
```

Note that since XML attribute values are evaluated as Python code, we recommend that you use double quotes to enclose the attribute value, and use single quotes for Python literal strings. For example:

```
<testcase name="'Test1'">
```

Using this convention makes your STAX jobs easier to read, compared to the following two examples, which use the same (single/double) quoting for both attributes and Python literal strings:

```
<testcase name=""Test1"">
<testcase name=''Test1''>
```

To specify an attribute value that is a Python variable (not a literal string), specify the following:

```
<testcase name="testName">
```

Since we only have one set of quotes (for the attribute value), Python will treat testName as a variable reference, and since we defined this variable in a **script** element earlier in this section, the attribute value will be the literal string "CoolTest1".

You can also include Python variable references within literal strings by using **%s** string substitution. For example:

```
<testcase name="'%s Part A' % testName">
```

Within the literal string you can include any number of %s variable references. After closing the literal string, you specify **%** followed by the variable reference(s). If there is more than one variable reference, then they must be enclosed by parenthesis and separated by commas:

```
<testcase name="'%s Part A on machine %s' % (testName, machineName)">
```

6.3. Python Lists

Python lists are objects that contain any number of other Python objects, including other Python lists. Python lists are enclosed with square braces, are comma-separated, and are 0-indexed. For example:

```
<script>L1 = ['abc', [0, 'xyz'], 8]</script>
```

Here we have created a Python variable, L1, that is set to a list of 3 objects. L1[0] is the literal string '**abc**'. L1[1] is a nested sublist **[0, 'xyz']**. L1[2] is the number **8**.

If we examine the nested sublist, L1[1][0] is the number **0**, and L1[1][1] is the literal string '**xyz**'.

6.4. Python Dictionaries

Python dictionaries are similar to maps in other languages. Python dictionaries are enclosed with curly braces, and contain a series of comma-separated key/value pairs. For example:

```
<script>M1 = {'machine': 'server1', 'duration': '2h'}</script>
```

Here we have created a Python variable, M1, that is set to a dictionary. This dictionary contains two keys: '**machine**' and '**duration**'. The value for 'machine' is '**server1**' and the value for 'duration' is '**2h**'.

6.5. References

For more information about Python, Jython, and XML:

- See the http://www.jython.org website for more information about Jython.
- See the http://www.python.org website for more information about Python.
- See the http://www.w3c.org website for more information about XML.
- See an XML Beginner's Guide.

7. Writing and Executing STAX Jobs

7.1. Basic STAX job template
7.2. Starting a process
7.3. Starting a stafcmd
7.4. Checking the stafcmd return code and the result
7.5. Checking the process return code and the result
7.6. Using script elements to define Python variables
7.7. Specifying scripts in the STAX Monitor
7.8. Adding parameters to a function
7.9. Adding logging to the STAX job
7.10. Importing and calling a STAX function
7.11. Adding execution control into a STAX job
7.12. Running tasks in parallel
7.13. Looping in a STAX job
7.14. Adding testcases into a STAX job
7.15. Adding time constraints into a STAX job
7.16. Sending email in a STAX job
7.17. Starting STAX jobs via EventManager
7.18. Starting STAX jobs via Cron

In these sections you will be executing some sample STAX jobs. To better understand the STAX jobs, these STAX jobs are shown with line numbers. For easier cut/paste of these STAX jobs, they are also included, without the line numbers, in section 8 of this doucment

7.1. Basic STAX job template

Now you are ready to start writing and executing some simple STAX jobs. Let's take a look at the basic template of a STAX job:

```
1:   <?xml version="1.0" encoding="UTF-8" standalone="no"?>
2:   <!DOCTYPE stax SYSTEM "stax.dtd">
3:
4:   <stax>
5:
6:     <defaultcall function="main"/>
7:
8:     <function name="main">
9:       <nop/>
10:    </function>
11:
12:  </stax>
```

Note that lines 1 and 2 are always the same, on line 4 we have the opening **stax** tag, and on line 12 we have the closing **stax** tag. **function** elements are used as the main structuring mechanism in STAX jobs. You can only define **function** elements within the root **stax** element.

In this example we have defined one **function** named "main". In the root **stax** element you are required to have a **defaultcall** element which indicates which function should be called when the job is submitted, if the submitter does not specify a particular function. In this example, we have defined the "main" function as the default function.

The attributes for **function** and **defaultcall** are always literal strings (you cannot use Python variables). That is why you see the double quotes, which enclose the attribute value, but you don't see any other quotes for Python, even though they are literal strings.

In our "main" **function**, we have defined an empty **nop** element. This is the simplest STAX job that you could create and execute. Let's execute it now.

Open your favorite text editor, or use an XML editor such as Cooktop, copy/paste lines 1-12 (delete the first five characters on each line, to get rid of the line numbers and whitespace), and save the file as DoesNothing.xml. Go to the main STAX Monitor panel, and click on the "Submit New Job..." button. For "XML Job File" "Filename:", enter the full path to the DoesNothing.xml file. Next, click on the "Test" button to check the STAX job for XML and Python errors. You should see a "Validation Successful" popup. Next, click on the "Submit New Job" button. You should see a new STAX Monitor window for this STAX job, and it should complete immediately.

7.2. Starting a process

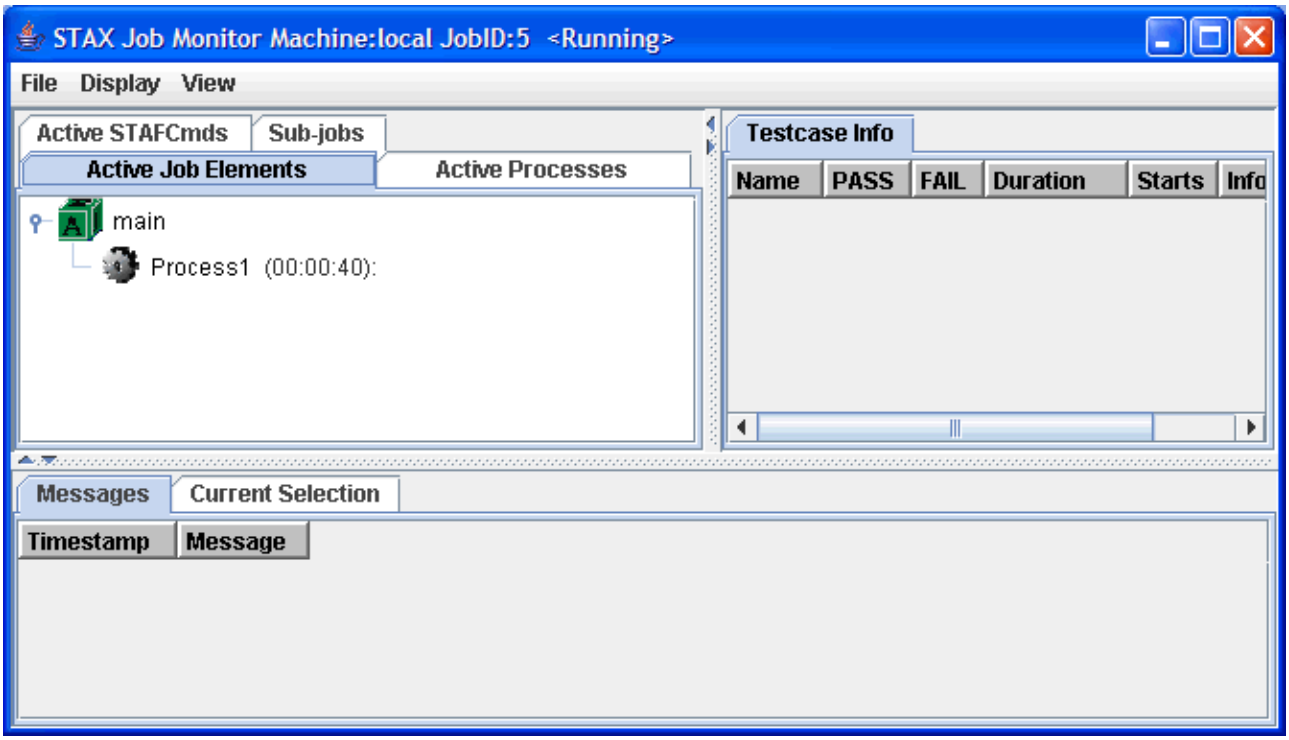
Now let's create a STAX job that starts a process. This example will use "notepad" on Windows. If you are on Unix, you can substitute some other GUI application (like "xclock"):

```
1:   <?xml version="1.0" encoding="UTF-8" standalone="no"?>
2:   <!DOCTYPE stax SYSTEM "stax.dtd">
3:
4:   <stax>
5:
6:     <defaultcall function="main"/>
7:
8:     <function name="main">
9:
10:      <process>
11:        <location>'local'</location>
12:        <command>'notepad'</command>
13:      </process>
14:
15:    </function>
16:
17:  </stax>
```

Notice that we have replaced the **nop** element with a **process** element, which has 2 sub-elements: **location** and **command**. We've specified that we want the process executed on the local machine (the STAX service machine), and the command to execute is notepad. Note that for location and command, the text is surrounded by single quotes, indicating that these are Python literal strings.

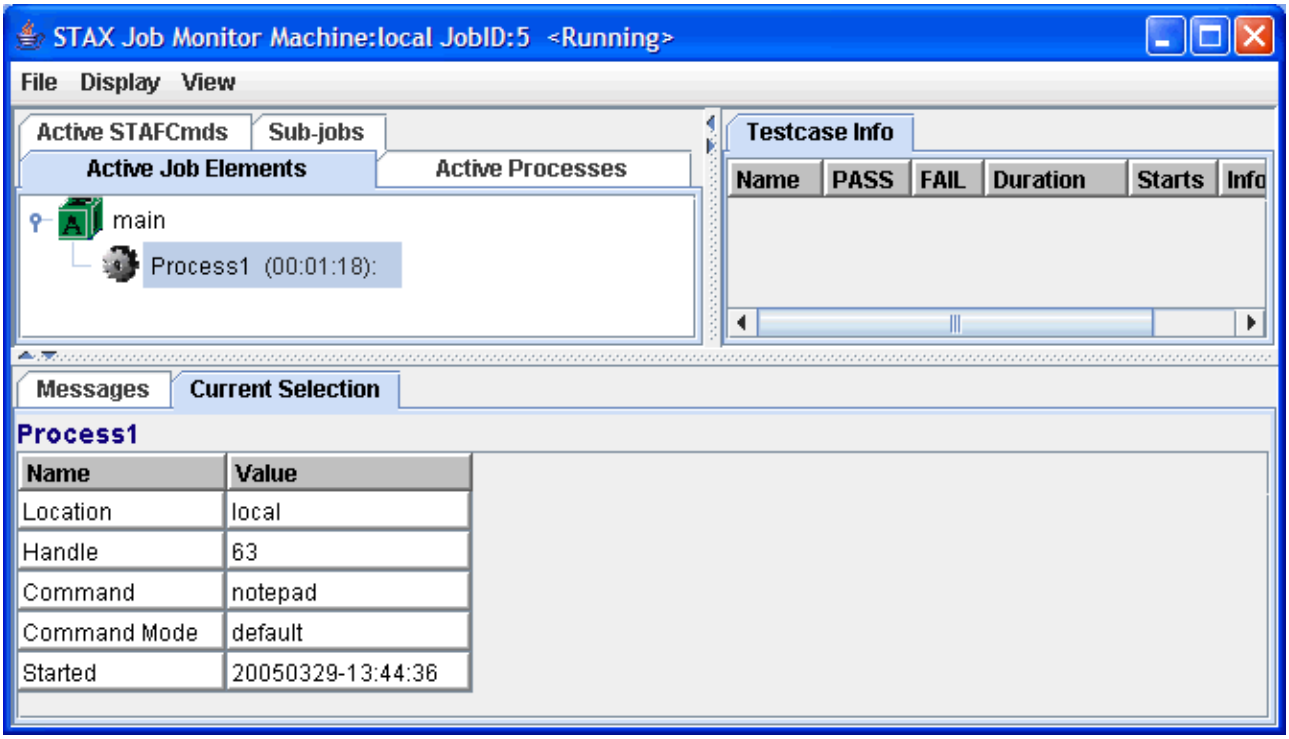
Open your favorite text editor, or use an XML editor such as Cooktop, copy/paste lines 1-17 (delete the first five characters on each line, to get rid of the line numbers and whitespace), and save the file as RunNotepadProcess.xml. Go to the main STAX Monitor panel, and click on the "Submit New Job..." button. For "XML Job File" "Filename:", enter the full path to the RunNotepadProcess.xml file. Next, click on the "Test" button to check the STAX job for XML and Python errors. You should see a "Validation Successful" popup. Next, click on the "Submit New Job" button. You should see a new STAX Monitor window for this STAX job, and you should see a Notepad process on your system. Your STAX Monitor window should look like:

Figure 14.



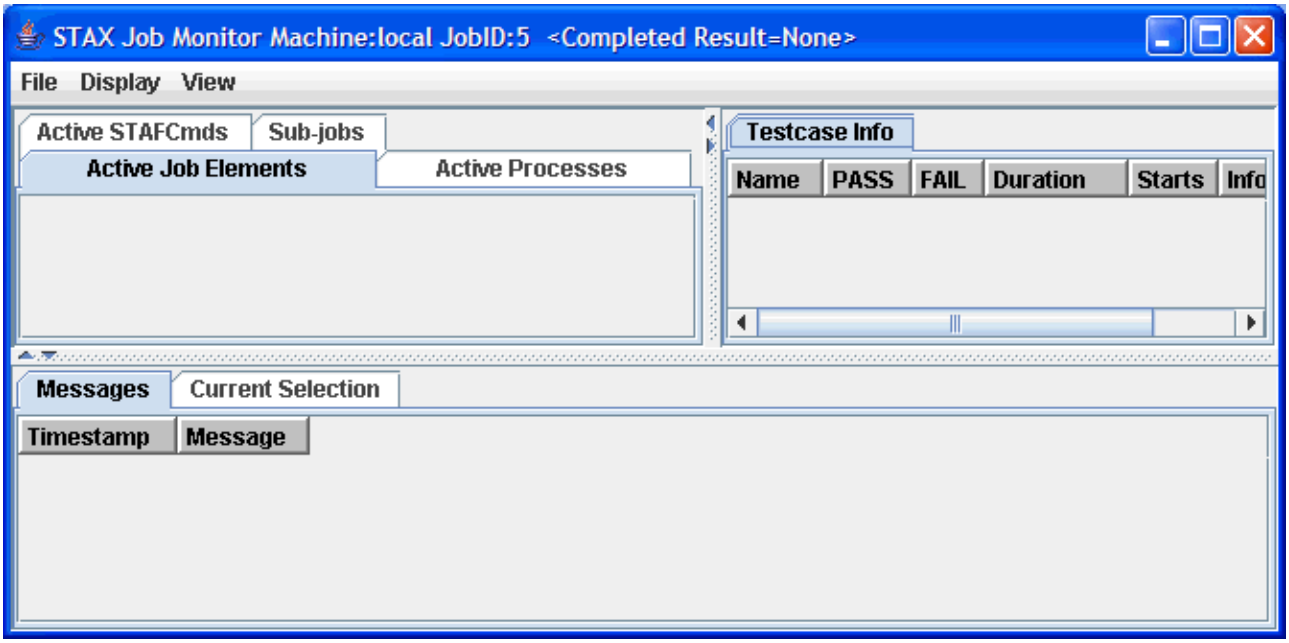
Notice that in the "Active Job Elements" tree, you always have a "main" block (note that this is not related to the function named "main"). The "main" block shows one process running. Click on the process in the tree to see the details about the process:

Figure 15.



At this point the STAX job is waiting for the process to complete. Bring the Notepad application to the foreground, and close it. The STAX Monitor should now show the job as completed.

Figure 16.



7.3. Starting a stafcmd

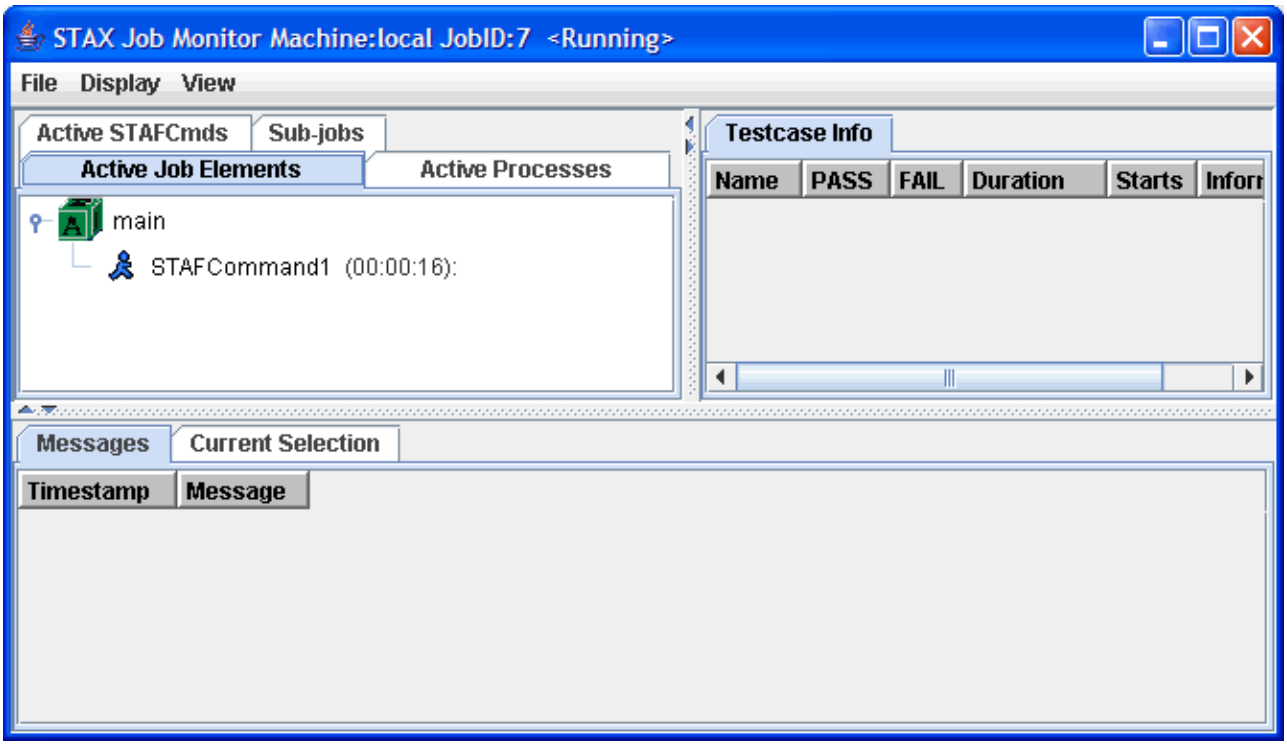
Now let's run a STAF service command:

```
1:   <?xml version="1.0" encoding="UTF-8" standalone="no"?>
2:   <!DOCTYPE stax SYSTEM "stax.dtd">
3:
4:   <stax>
5:
6:     <defaultcall function="main"/>
7:
8:     <function name="main">
9:
10:      <stafcmd>
11:        <location>'local'</location>
12:        <service>'delay'</service>
13:        <request>'delay 30000'</request>
14:      </stafcmd>
15:
16:    </function>
17:
18:  </stax>
```

In this example we now have a **stafcmd** element which allows us to call any STAF service. It has sub-elements **location**, **service**, and **request**. We have indicated that we want to call the DELAY service on the local machine, and have it delay for 30 seconds.

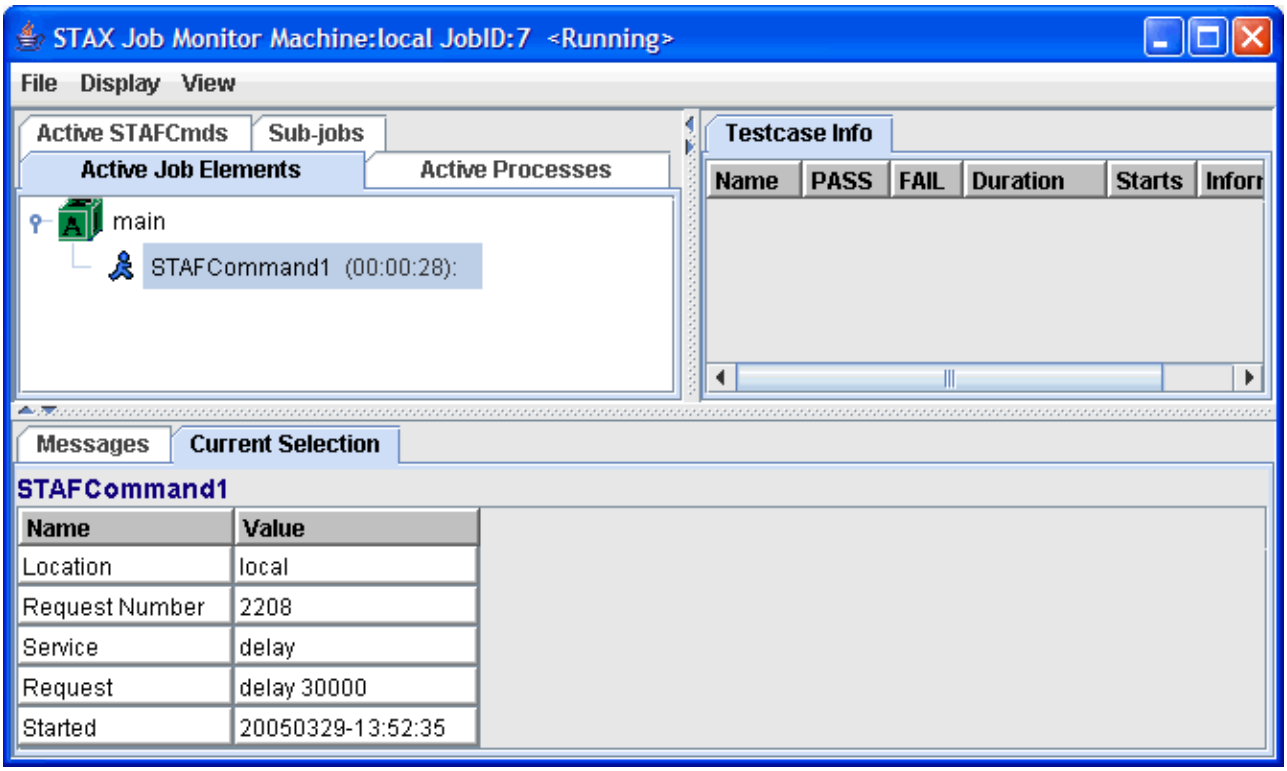
Open your favorite text editor, or use an XML editor such as Cooktop, copy/paste lines 1-18 (delete the first five characters on each line, to get rid of the line numbers and whitespace), and save the file as RunDelayRequest.xml. Go to the main STAX Monitor panel, and click on the "Submit New Job..." button. For "XML Job File" "Filename:", enter the full path to the RunDelayRequest.xml file. Next, click on the "Test" button to check the STAX job for XML and Python errors. You should see a "Validation Successful" popup. Next, click on the "Submit New Job" button. You should see a new STAX Monitor window for this STAX job. Your STAX Monitor window should look like:

Figure 17.



The "Active Job Elements" tree now shows the STAF Command executing. Click on the command to see the details about it:

Figure 18.



After the STAF Command completes (in 30 seconds), the job should complete.

7.4. Checking the stafcmd return code and the result

Now let's run another STAF service command, and check the return code and the result value.

```
1:   <?xml version="1.0" encoding="UTF-8" standalone="no"?>
2:   <!DOCTYPE stax SYSTEM "stax.dtd">
3:
4:   <stax>
5:
6:     <defaultcall function="main"/>
7:
8:     <function name="main">
9:
10:      <sequence>
11:
12:        <stafcmd>
13:          <location>'local'</location>
14:          <service>'var'</service>
15:          <request>'resolve string {STAF/Config/OS/Name}'</request>
16:        </stafcmd>
17:
18:        <if expr="RC != 0">
19:          <message>'Oops, RC = %s, Result = %s' % (RC, STAFResult)</message>
20:          <else>
21:            <message>'Great!  STAF/Config/OS/Name = %s' % (STAFResult)</message>
22:          </else>
23:        </if>
24:
25:      </sequence>
26:
27:    </function>
28:
29:  </stax>
```

In this STAX job, we have a **stafcmd** at line 12 that resolves the STAF/Config/OS/Name variable. Notice that within the **function**, we have added a **sequence** element at line 10. This is needed because the wrapper elements, such as **function**, can only contain a single sub-element. In RunDelayRequest.xml, we only had the single **stafcmd** within the function, so the sequence was not needed. Now, however, we have a **stafcmd** element and an **if** element, which is why we need to enclose those with the **sequence**.

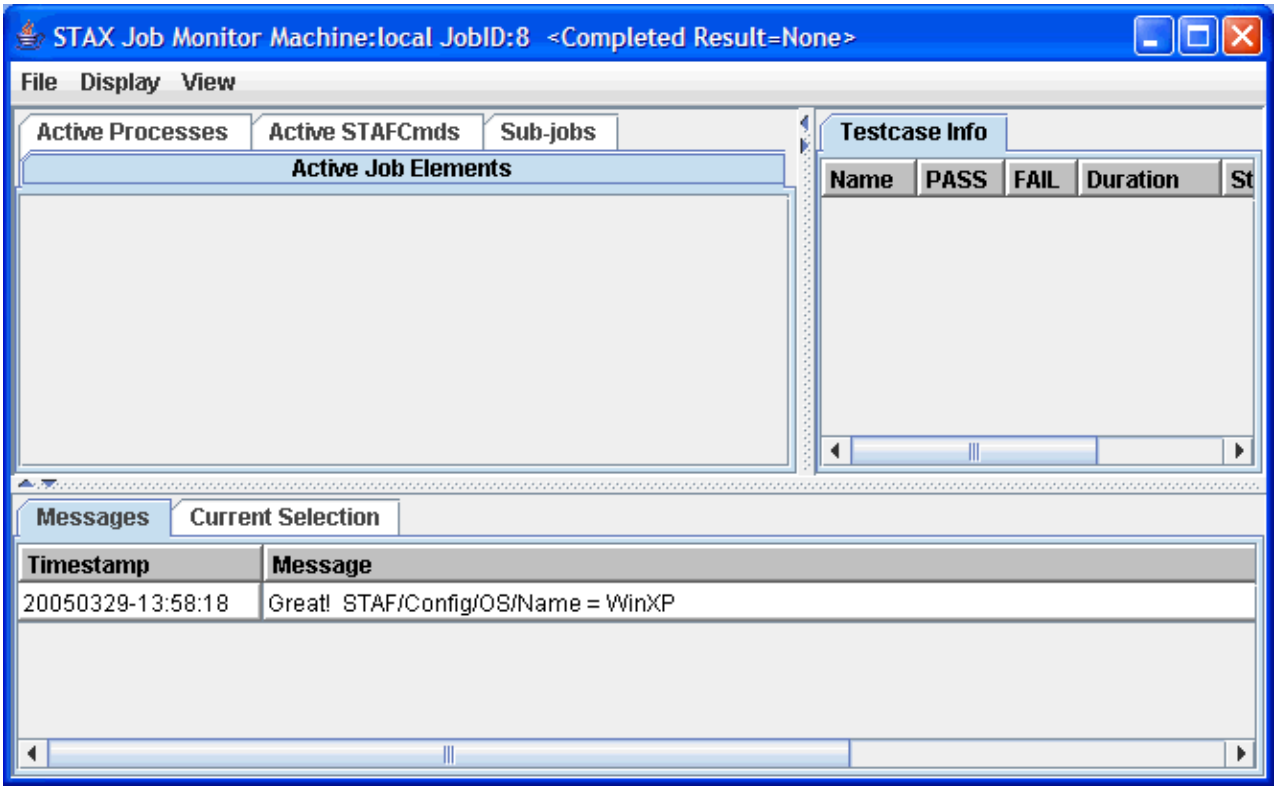
It is highly recommended that after every **stafcmd** you check with return code from the service request. The return code is accessible via the RC variable. Notice that on line 18 we have an **if** element to check the return code. For its **expr** attribute, we specify a Python expression that will be evaluated as either true or false.

This STAX job also introduces the **message** element. You use this element to write information to the STAX Monitor. So, if **RC != 0**, then we will write the "Oops" message to the STAX Monitor, otherwise, we will write the "Great!" message to the STAX Monitor.

Also notice that in the **message** elements, we are using the %s string substitution to include the RC and STAFResult variables in the messages. The STAFResult variable we be set to the result string returned by the service (in this case, the STAF/Config/OS/Name variable).

Open your favorite text editor, or use an XML editor such as Cooktop, copy/paste lines 1-29 (delete the first five characters on each line, to get rid of the line numbers and whitespace), and save the file as CheckSTAFCmdRC.xml. Go to the main STAX Monitor panel, and click on the "Submit New Job..." button. For "XML Job File" "Filename:", enter the full path to the CheckSTAFCmdRC.xml file. Next, click on the "Test" button to check the STAX job for XML and Python errors. You should see a "Validation Successful" popup. Next, click on the "Submit New Job" button. You should see a new STAX Monitor window for this STAX job. Your STAX Monitor window should look like:

Figure 19.



Notice in the Messages tab, you see the message written by the STAX job.

7.5. Checking the process return code and the result

This next STAX job will run a Java class that is provided in the STAXMon.jar file. Here is the syntax for this Java class:

```
java -cp C:\STAF\bin\JSTAF.jar;C:\STAF\services\stax\STAXMon.jar
com.ibm.staf.service.stax.TestProcess

Usage: java TestProcess loopCount incrementSeconds returnCode
```

Here are the parameters the class accepts:

loopCount	The number of loops the class should perform.
incrementSeconds	The number of seconds the class should wait during each loop iteration.
returnCode	The return code the class should return.

This STAX job will execute this process with the parameters "10 3 99":

```
1:   <?xml version="1.0" encoding="UTF-8" standalone="no"?>
2:   <!DOCTYPE stax SYSTEM "stax.dtd">
3:
4:   <stax>
5:
6:     <defaultcall function="main"/>
7:
8:     <function name="main">
9:
10:      <sequence>
11:
12:        <process name="'My Test Process'">
13:          <location>'local'</location>
14:          <command>
15:            'java com.ibm.staf.service.stax.TestProcess 10 3 99'
16:          </command>
17:          <env>
18:            'CLASSPATH=C:/STAF/bin/JSTAF.jar;C:/STAF/services/stax/STAXMon.jar'
19:          </env>
20:          <stderr mode="'stdout'"/>
21:          <returnstdout/>
22:        </process>
23:
24:        <if expr="RC != 0">
25:          <message>'Error: RC=%s, STAXResult=%s' % (RC, STAXResult)</message>
26:          <else>
27:            <message>'Process RC was 0.  STAXResult=%s' % STAXResult</message>
28:          </else>
29:        </if>
30:
31:      </sequence>
32:
33:    </function>
34:
35:  </stax>
```

For this **process** element, we are running the "java" executable on the local machine, and we are using the **env** element to set the CLASSPATH for the process. Also notice that we have specified the "name" attribute for the **process**. For both **process** and **stafcmd**, if you don't specify the "name" attribute, they will be given names like Process1, Process2, STAFCmd1, STAFCmd2, for example. Using the "name" attribute allows you to specify more detailed names that can self-document what is happening in your STAX jobs.

Also notice that we are specifying the **stderr** and **returnstdout** elements. The **returnstdout** element indicates that we want the standard output of the process to be returned when the process completes. The **stderr** element indicates that we want the standard error information to be returned as

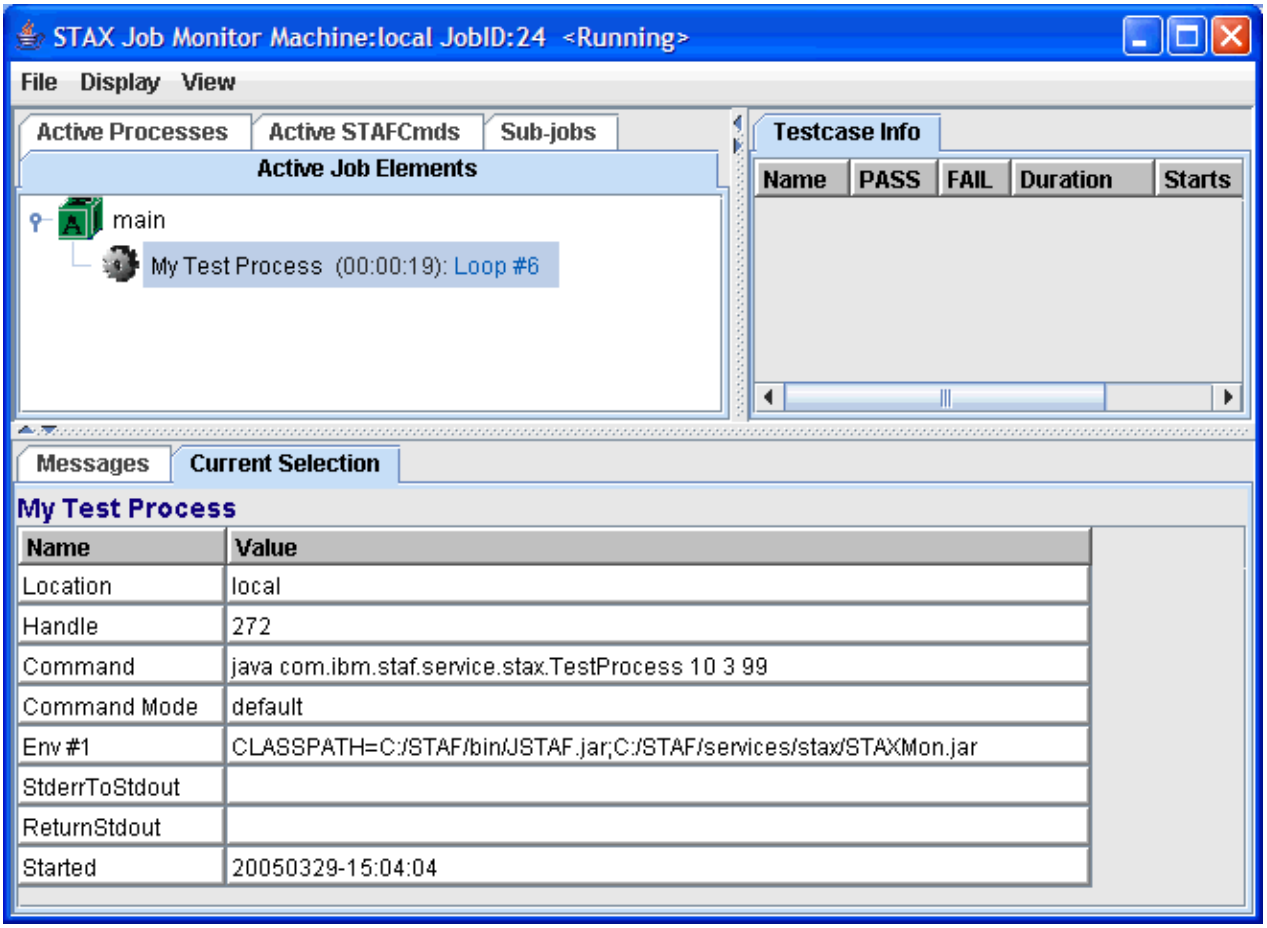
part of the standard output. This allows you to examine the information in your STAX job after the process completes, and it is a good practice to specify these elements for your processes.

Next, we have an **if** element to check if the RC variable is set. The RC variable will be set to the error code returned by the process. In this STAX job, the process will be returning error code 99. It is highly recommended that you check the return code after every **process**.

Notice in the **message** element that we specify the STAXResult variable. This variable will contain any files, such as stdout or stderr, that we specified to return when the job completes.

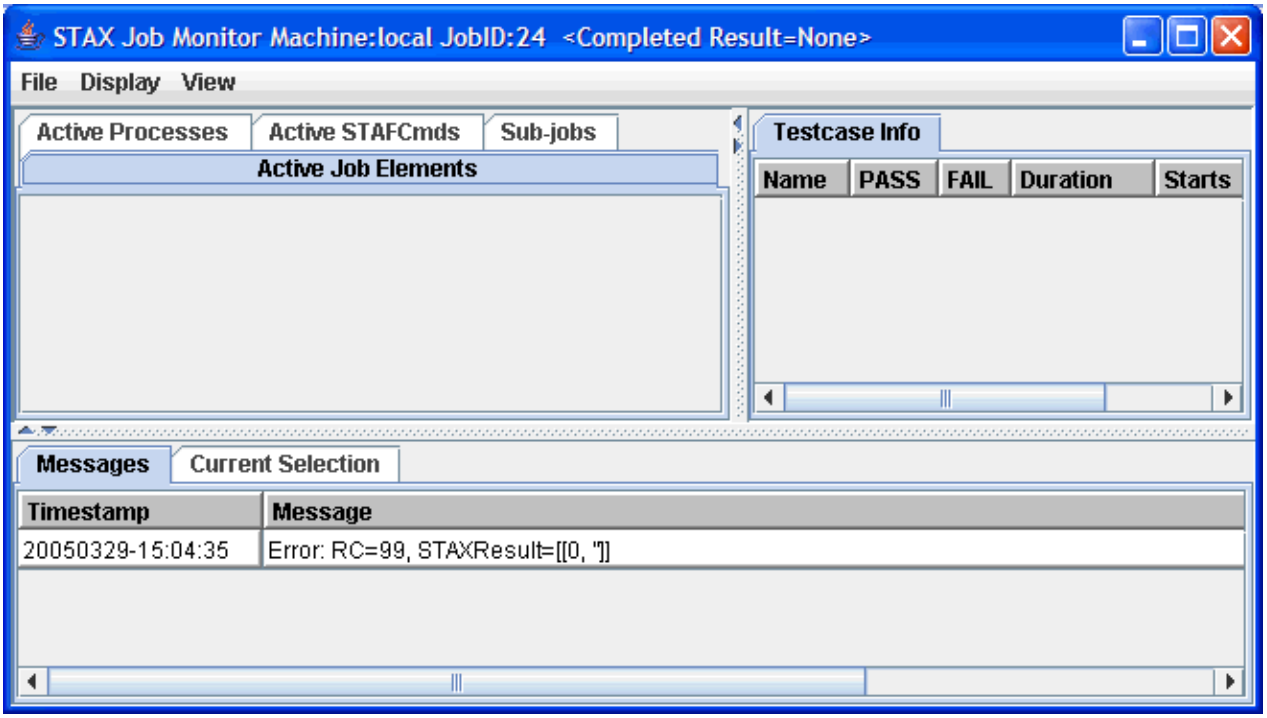
Open your favorite text editor, or use an XML editor such as Cooktop, copy/paste lines 1-35 (delete the first five characters on each line, to get rid of the line numbers and whitespace), and save the file as RunTestProcess.xml. Go to the main STAX Monitor panel, and click on the "Submit New Job..." button. For "XML Job File" "Filename:", enter the full path to the RunTestProcess.xml file. Next, click on the "Test" button to check the STAX job for XML and Python errors. You should see a "Validation Successful" popup. Next, click on the "Submit New Job" button. You should see a new STAX Monitor window for this STAX job. Your STAX Monitor window should look like the following (click on "My Test Process" in the Active Job Elements tree to get the details about the process):

Figure 20.



After the job completes, click on the Messages tab:

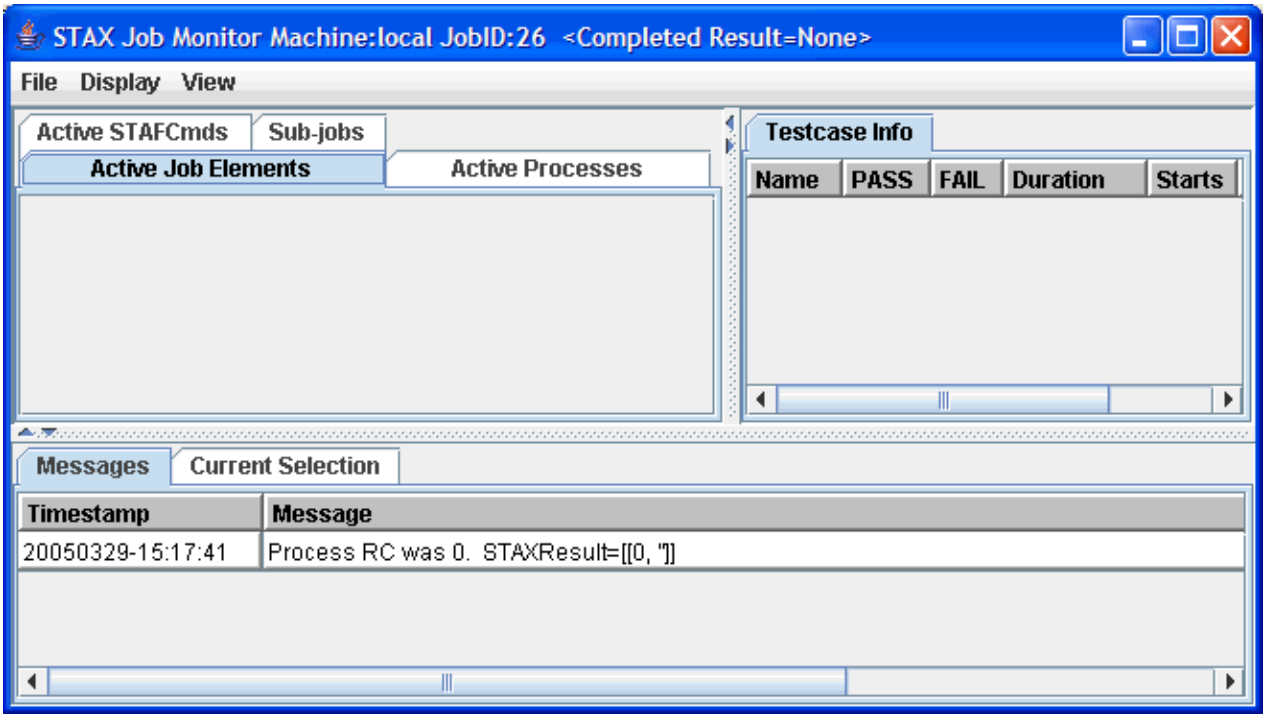
Figure 21.



Notice that STAXResult variable is set to [[0, '']]. This variable contains a list of all of the files you specified to have returned after the process completes. In this example we only specified one file to return, the standard output (which will include the standard error). The first item (in the first sublist), 0, is the RC from retrieving the file. The second item (in the first sublist), '', is the content of the file (in this example the process did not write anything to stdout or stderr).

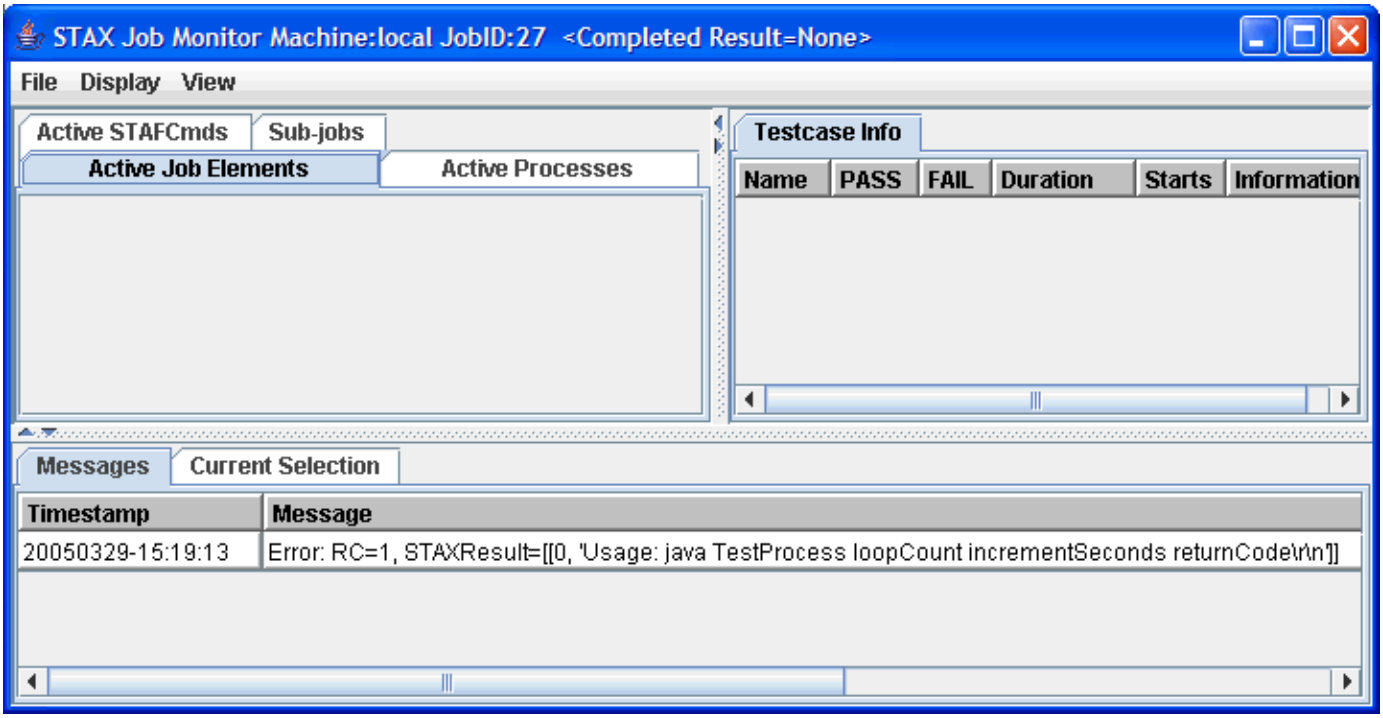
Now change the 99 in the **parms** element to 0, and rerun the job. Verify that when the job completes, you get the "Process RC was 0" message:

Figure 22.



Now change the **parms** element and remove the last parameter, 0, so that the TestProcess writes to its standard output (to indicate that it was not passed the correct nubmer of parameters). Verify that you get the correct STAXResult:

Figure 23.



7.6. Using script elements to define Python variables

This next STAX job is similar to the previous job, but in this job we will add a **script** element to define some Python variables:

```
1:   <?xml version="1.0" encoding="UTF-8" standalone="no"?>
2:   <!DOCTYPE stax SYSTEM "stax.dtd">
3:
4:   <stax>
5:
6:     <script>
7:       jstafJar = '{STAF/Config/STAFRoot}/bin/JSTAF.jar'
8:       staxmonJar = '{STAF/Config/STAFRoot}/services/stax/STAXMon.jar'
9:       machine = 'local'
10:      java_command = 'java'
11:      java_class = 'com.ibm.staf.service.stax.TestProcess'
12:      loopCount = '10'
13:      incSeconds = '3'
14:      returnCode = '50'
15:      parms = '%s %s %s' % (loopCount, incSeconds, returnCode)
16:      cp = 'CLASSPATH=%s;%s' % (jstafJar, staxmonJar)
17:    </script>
18:
19:    <defaultcall function="main"/>
20:
21:    <function name="main">
22:
23:      <sequence>
24:
25:        <process name="'My Test Process'">
26:          <location>machine</location>
27:          <command>'%s %s %s' % (java_command, java_class, parms)</command>
28:          <env>cp</env>
29:          <stderr mode="'stdout'"/>
30:          <returnstdout/>
31:        </process>
32:
33:        <if expr="RC != 0">
```

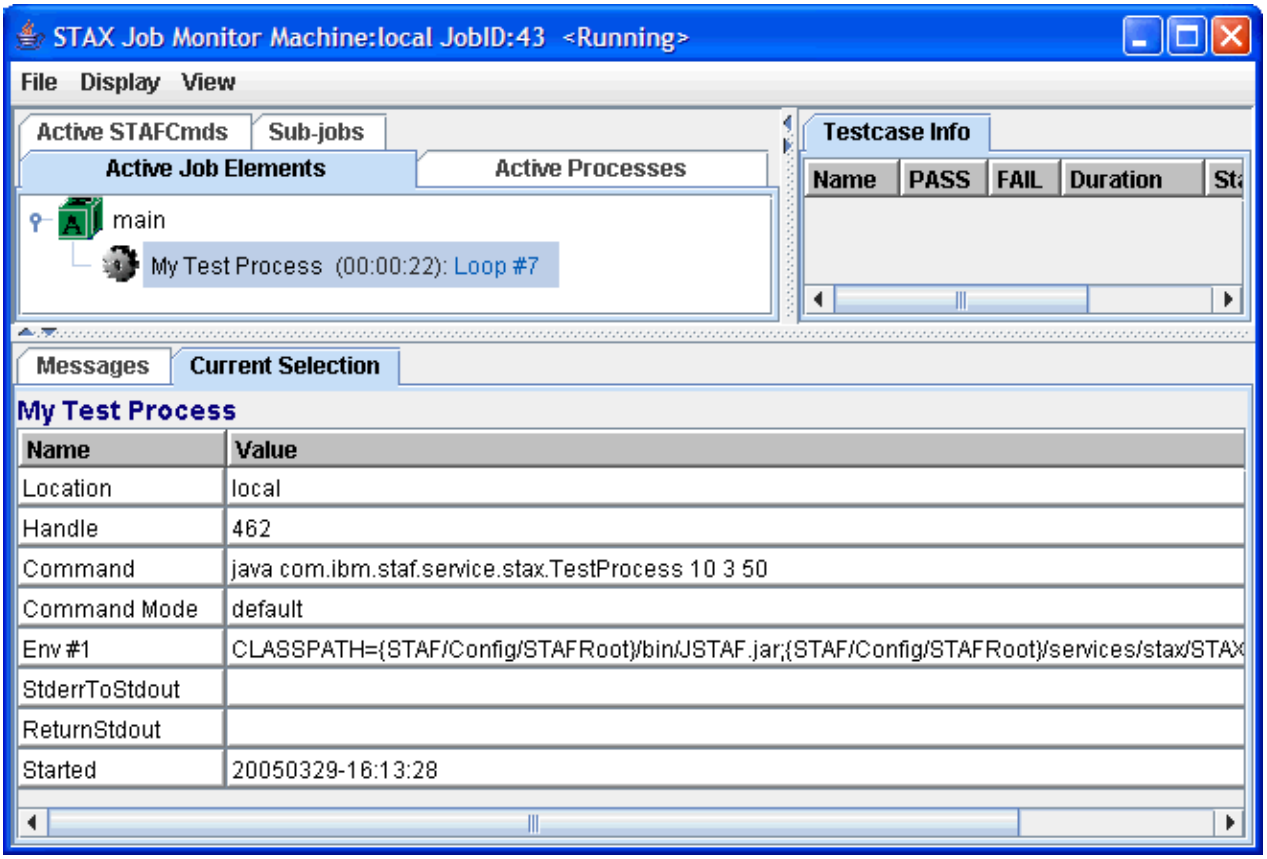
```
34:          <message>'Error: RC=%s, STAXResult=%s' % (RC, STAXResult)</message>
35:          <else>
36:            <message>'Process RC was 0.  STAXResult=%s' % STAXResult</message>
37:          </else>
38:        </if>
39:
40:      </sequence>
41:
42:    </function>
43:
44:  </stax>
```

On line 6 we have a **script** element to define 10 variables, some of which reference other variables using %s, and on lines 26-28 we use these variables for **location**, **command**, and **env**. Note that we don't use single quotes for **location** and **env** elements, since we are just using the variables instead of literal strings. For **command**, we use the single quotes to build up the string, using the %s substitution.

Since this **script** element has been defined in the root **stax** element, it will be executed at the beginning of the STAX job, prior to calling the "main" function.

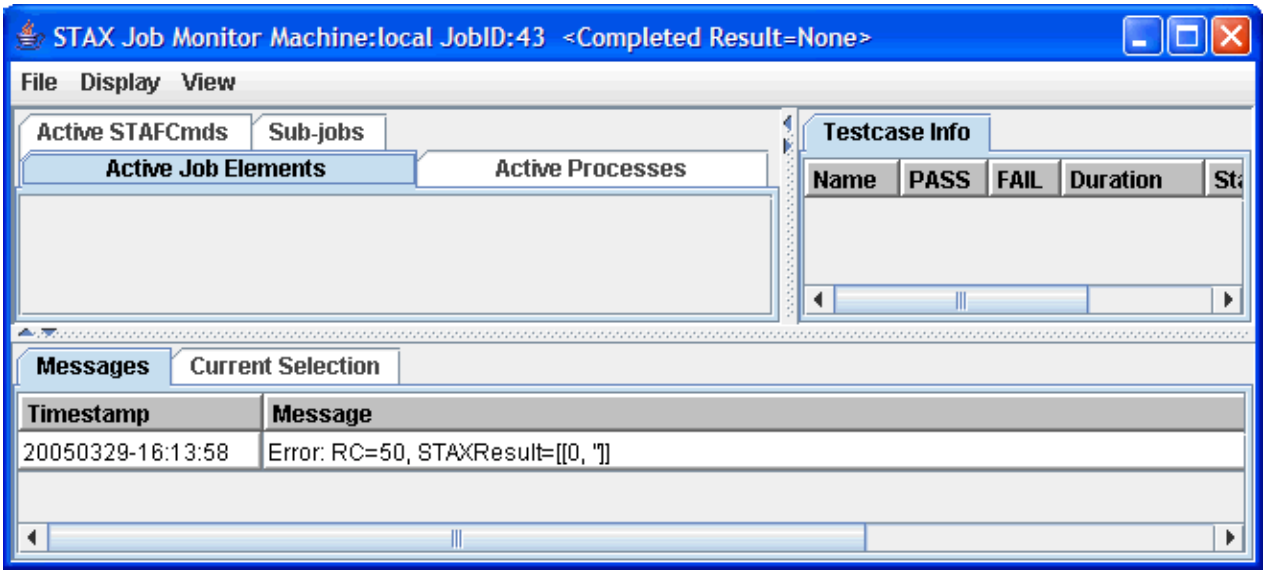
Open your favorite text editor, or use an XML editor such as Cooktop, copy/paste lines 1-44 (delete the first five characters on each line, to get rid of the line numbers and whitespace), and save the file as UsingScripts.xml. Go to the main STAX Monitor panel, and click on the "Submit New Job..." button. For "XML Job File" "Filename:", enter the full path to the UsingScripts.xml file. Next, click on the "Test" button to check the STAX job for XML and Python errors. You should see a "Validation Successful" popup. Next, click on the "Submit New Job" button. You should see a new STAX Monitor window for this STAX job. Your STAX Monitor window should look like the following (click on "My Test Process" in the Active Job Elements tree to get the details about the process):

Figure 24.



After the job completes, click on the Messages tab:

Figure 25.



7.7. Specifying scripts in the STAX Monitor

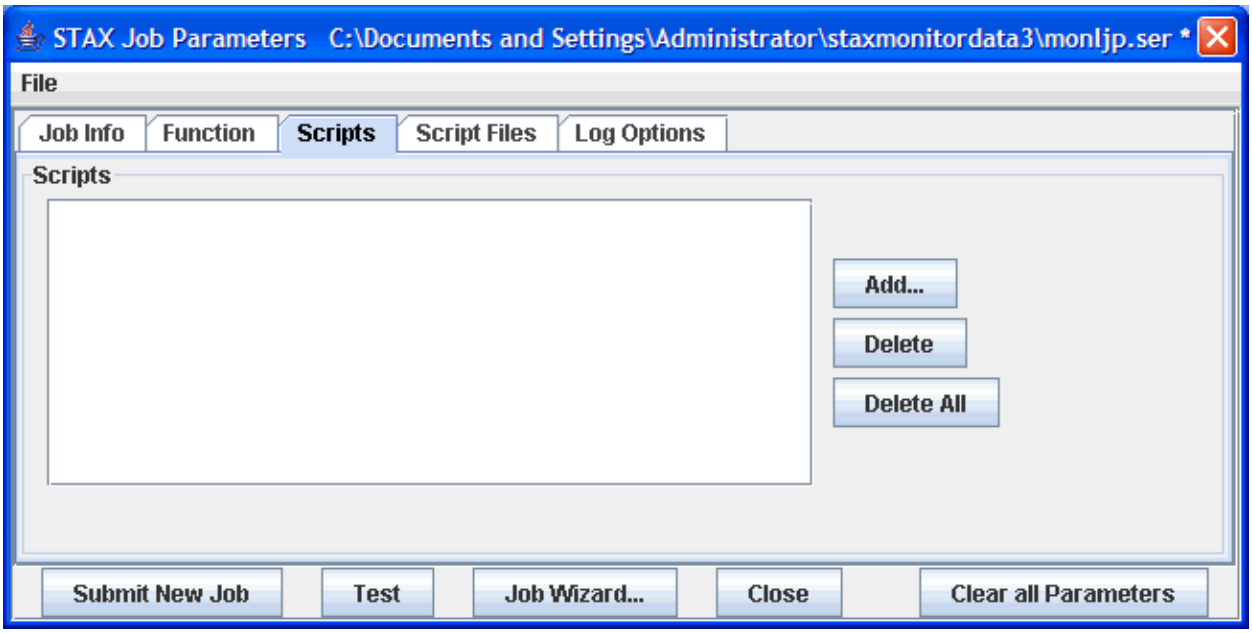
When starting a job via the STAX Monitor (or from the command line), you can specify any numbers of scripts. These scripts override any **script** elements that are defined in the root **stax** element. So, if we use the exact same job as the last example:

```
1:   <?xml version="1.0" encoding="UTF-8" standalone="no"?>
2:   <!DOCTYPE stax SYSTEM "stax.dtd">
3:
4:   <stax>
5:
6:     <script>
7:       jstafJar = '{STAF/Config/STAFRoot}/bin/JSTAF.jar'
8:       staxmonJar = '{STAF/Config/STAFRoot}/services/stax/STAXMon.jar'
9:       machine = 'local'
10:      java_command = 'java'
11:      java_class = 'com.ibm.staf.service.stax.TestProcess'
12:      loopCount = '10'
13:      incSeconds = '3'
14:      returnCode = '50'
15:      parms = '%s %s %s' % (loopCount, incSeconds, returnCode)
16:      cp = 'CLASSPATH=%s;%s' % (jstafJar, staxmonJar)
17:    </script>
18:
19:    <defaultcall function="main"/>
20:
21:    <function name="main">
22:
23:      <sequence>
24:
25:        <process name="'My Test Process'">
26:          <location>machine</location>
27:          <command>'%s %s %s' % (java_command, java_class, parms)</command>
28:          <env>cp</env>
29:          <stderr mode="'stdout'"/>
30:          <returnstdout/>
31:        </process>
32:
33:        <if expr="RC != 0">
34:          <message>'Error: RC=%s, STAXResult=%s' % (RC, STAXResult)</message>
35:          <else>
36:            <message>'Process RC was 0.  STAXResult=%s' % STAXResult</message>
```

```
37:          </else>
38:        </if>
39:
40:      </sequence>
41:
42:    </function>
43:
44:  </stax>
```

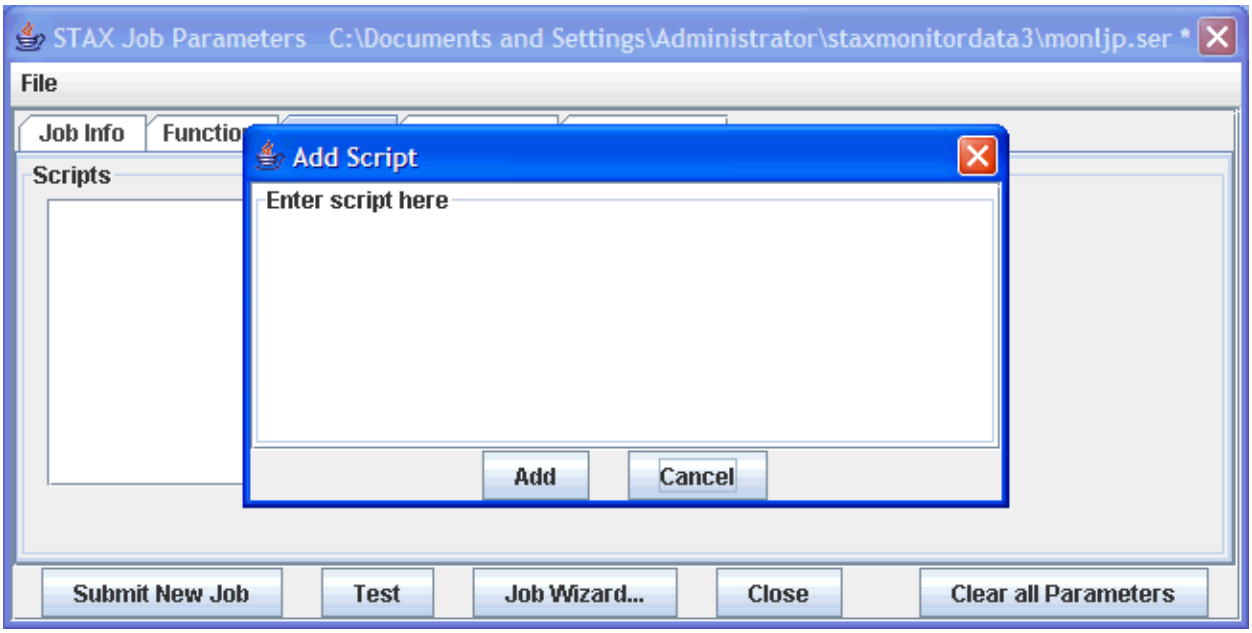
But let's say that we want to override the parameters, and specify parameters "30 1 10" (30 loops, 1 second per loop, return code 10). In the main STAX Monitor window, click on the "Submit New Job..." button, and leave UsingScripts.xml as the "Filename:". Click on the "Scripts" tab:

Figure 26.



Then click on the "Add" button:

Figure 27.

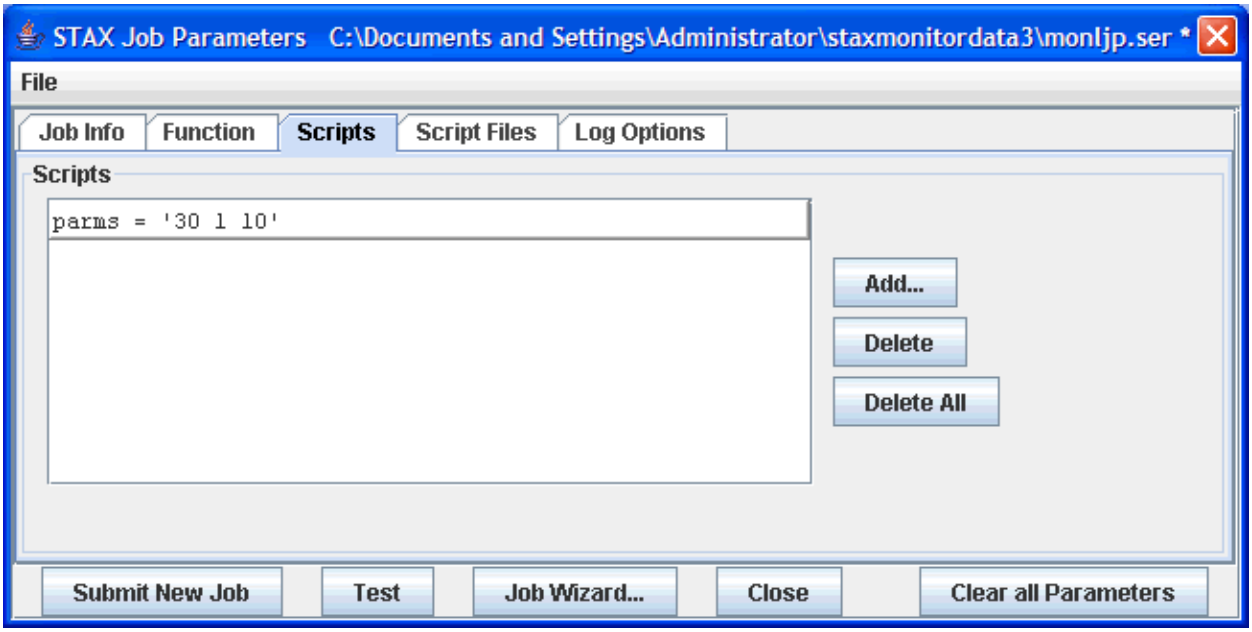


In the "Enter script here" field, enter:

```
parms = '30 1 10'
```

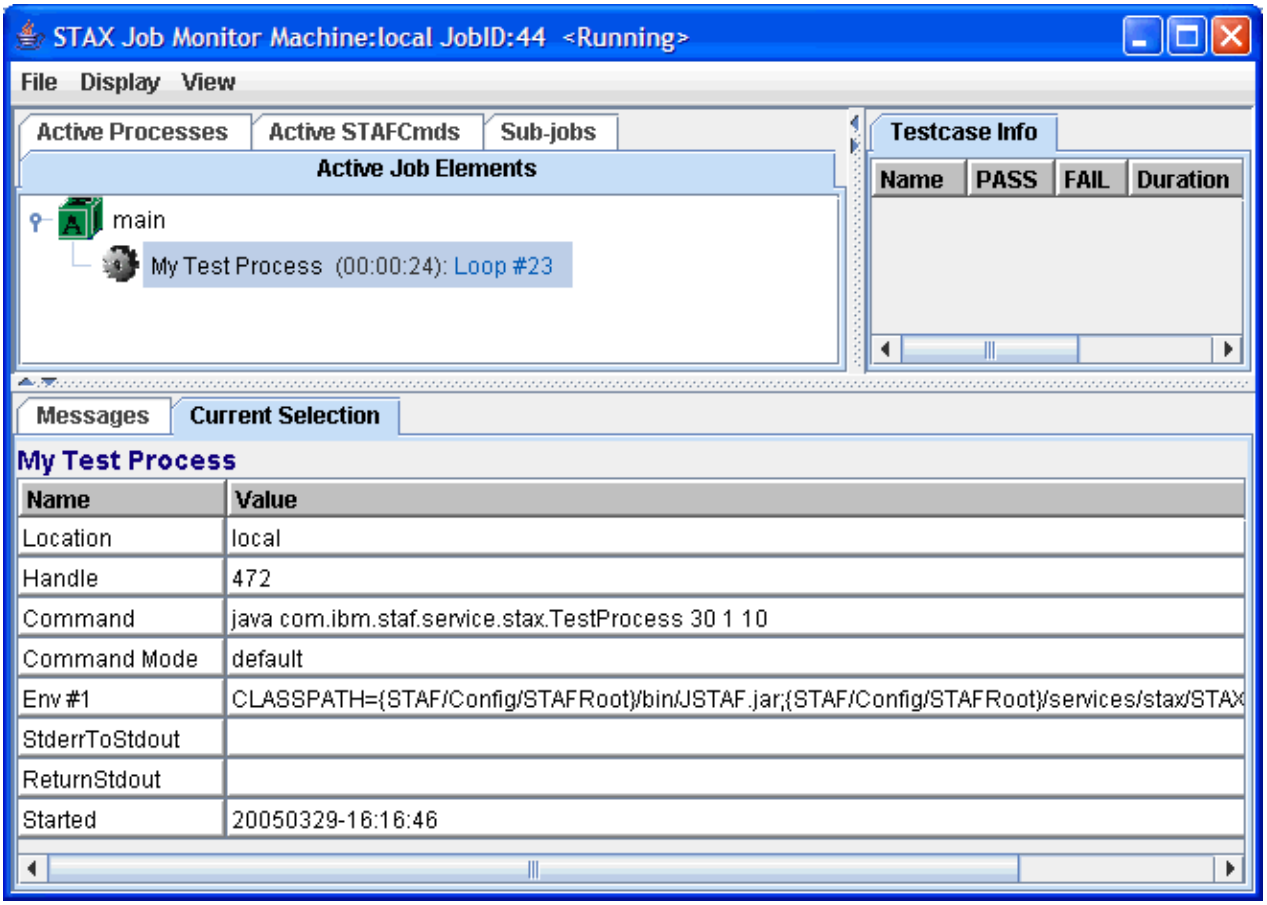
and click on the "Add" button:

Figure 28.



Next, click on the "Test" button to check the STAX job for XML and Python errors. You should see a "Validation Successful" popup. Next, click on the "Submit New Job" button. You should see a new STAX Monitor window for this STAX job. Your STAX Monitor window should look like the following (click on "My Test Process" in the Active Job Elements tree to get the details about the process):

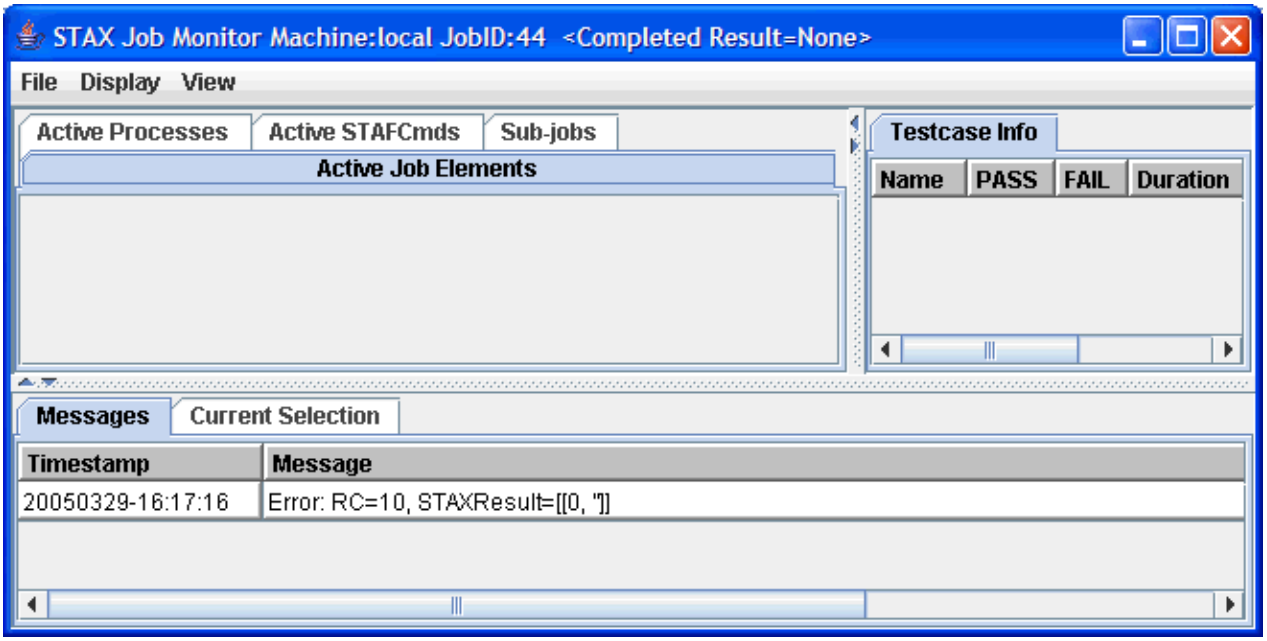
Figure 29.





Notice that the parameters that you specified in the "Scripts" tab in the STAX Monitor are being used. When the job starts executing, the **script** element in the root **stax** element is executed, and the parms variable has the value of 'com.ibm.staf.service.stax.TestProcess 10 3 50'. Next, any scripts that were specified in the STAX Monitor are executed, so the parms variable now has the value of 'com.ibm.staf.service.stax.TestProcess 30 1 10'. Next, the "main" function is called and the process is executed. When the job is complete, click on the "Messages" tab:

Figure 30.



In the main STAX Monitor window, click on the "Submit New Job..." button, and click on the "Scripts" tab. Select the "parms = '30 1 10'" entry, and click on the "Delete" button, and then click on the "Close" button.

7.8. Adding parameters to a function

In this next STAX job, we will now pass the "machine", "java_command", "parms", and "classpath" values as parameters to the function, rather than defining them in a **script** element or in the STAX Monitor "Scripts" tab.

```
1:   <?xml version="1.0" encoding="UTF-8" standalone="no"?>
2:   <!DOCTYPE stax SYSTEM "stax.dtd">
3:
4:   <stax>
5:
6:     <defaultcall function="main"/>
7:
8:     <function name="main">
9:
10:      <function-prolog>
11:        This function is used as an example in the "Getting Started with STAX"
12:        document.  It starts the TestProcess, and allows the parms, machine,
13:        java_command, java_class, processName, and classpath to be passed as
14:        arguments to the function.
15:      </function-prolog>
16:
17:      <function-map-args>
18:        <function-required-arg name="parms">
19:          The three parameters to pass to the process.
20:        </function-required-arg>
21:        <function-optional-arg name="machine" default="'local'">
22:          The name of machine where the test process should run.
```

```
23:        </function-optional-arg>
24:        <function-optional-arg name="java_command" default="'java'">
25:          The name of java executable that should be used to execute the test
26:          process.
27:        </function-optional-arg>
28:        <function-optional-arg name="java_class"
29:            default="'com.ibm.staf.service.stax.TestProcess'">
30:          The name of java class for the test process.
31:        </function-optional-arg>
32:        <function-optional-arg name="processName" default="'My Test Process'">
33:          The name of the process.
34:        </function-optional-arg>
35:        <function-optional-arg name="classpath"
36:
default="'{STAF/Config/STAFRoot}/bin/JSTAF.jar;{STAF/Config/STAFRoot}/services/stax/STAXMon.jar'">
37:          The CLASSPATH that should be used when the test process is started..
38:        </function-optional-arg>
39:      </function-map-args>
40:
41:      <sequence>
42:
43:        <process name="processName">
44:          <location>machine</location>
45:          <command>'%s %s %s' % (java_command, java_class, parms)</command>
46:          <env>'CLASSPATH=%s' % classpath</env>
47:          <stderr mode="'stdout'"/>
48:          <returnstdout/>
49:        </process>
50:
51:        <if expr="RC != 0">
52:          <message>'Error: RC=%s, STAXResult=%s' % (RC, STAXResult)</message>
53:          <else>
54:            <message>'Process RC was 0.  STAXResult=%s' % STAXResult</message>
55:          </else>
56:        </if>
57:
58:      </sequence>
59:
60:    </function>
61:
62:  </stax>
```

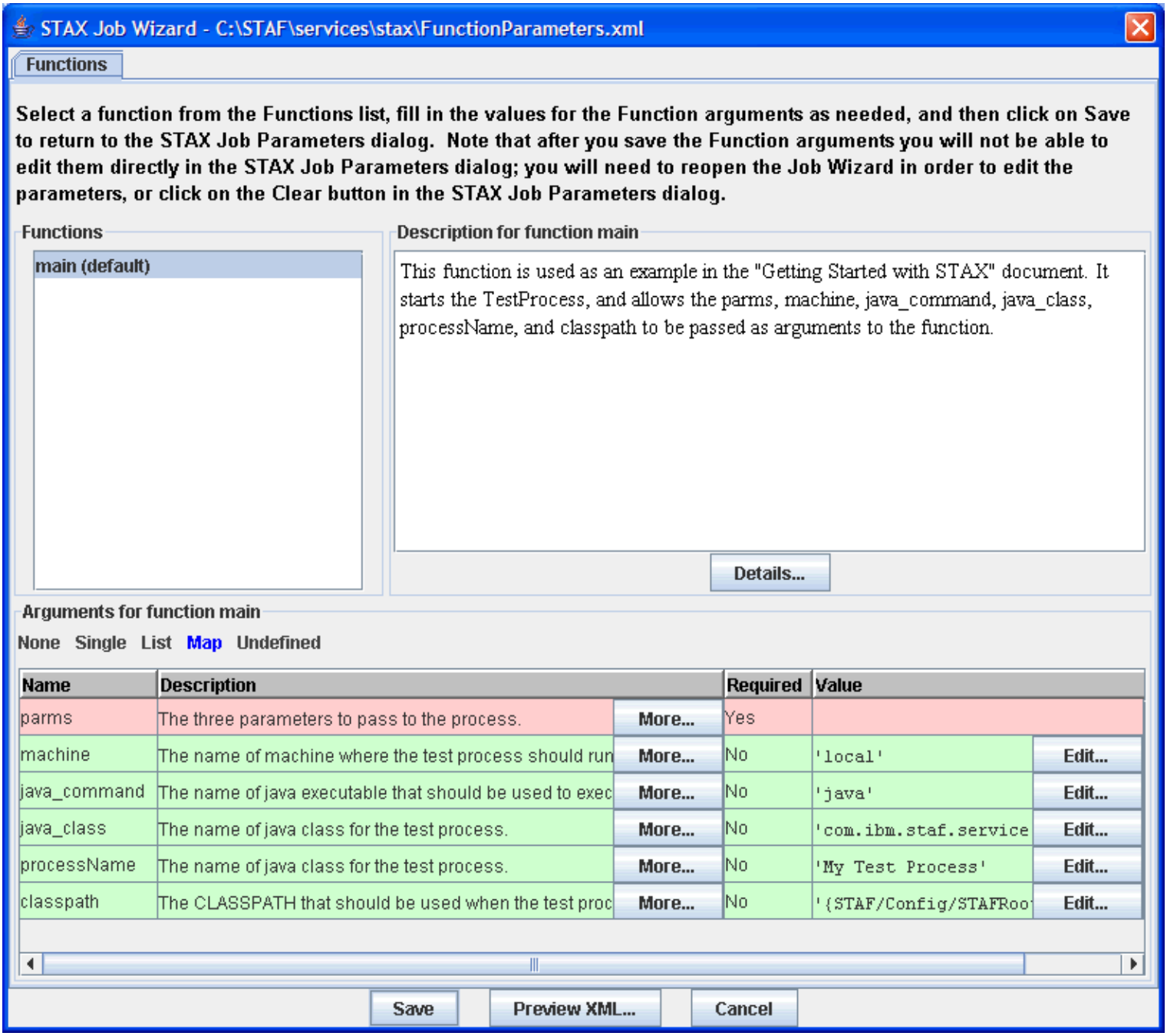
Notice that after the **function** opening tag, on line 10 we have added a **function-prolog** element. This is where you specify any plain text or HTML that describes the function. Note that this text is not evaluated as Python code, and so we do not enclose it in quotes.

Next, on line 17 we have a **function-map-args** element that indicates that you can pass a Python map for the arguments to the function. You can pass 6 arguments to the function; 1 is required, and 5 are optional, with default values. Within the **function-map-args** element we have one **function-required-arg** and five **function-optional-arg**. All six have a "name" attribute that is the name of the Python variable for the argument. For the **function-optional-arg** elements, you must specify a "default" argument, which is the value of the argument if it is not specified by the caller. For the character data for all six, you can specify plain text or HTML that describes the argument. Note that this text is not evaluated as Python code, and so we do not enclose it in quotes.

Open your favorite text editor, or use an XML editor such as Cooktop, copy/paste lines 1-62 (delete the first five characters on each line, to get rid of the line numbers and whitespace), and save the file as FunctionParameters.xml. Go to the main STAX Monitor panel, and click on the "Submit New Job..." button. For "XML Job File" "Filename:", enter the full path to the FunctionParameters.xml file.

Next click on the "Job Wizard..." button. You should see the following dialog:

Figure 31.



The STAX Job Wizard reads in the STAX job, and displays information about the functions in the STAX job. Since we have only defined one function with this STAX job, we only see the "main" function in the "Functions" list.

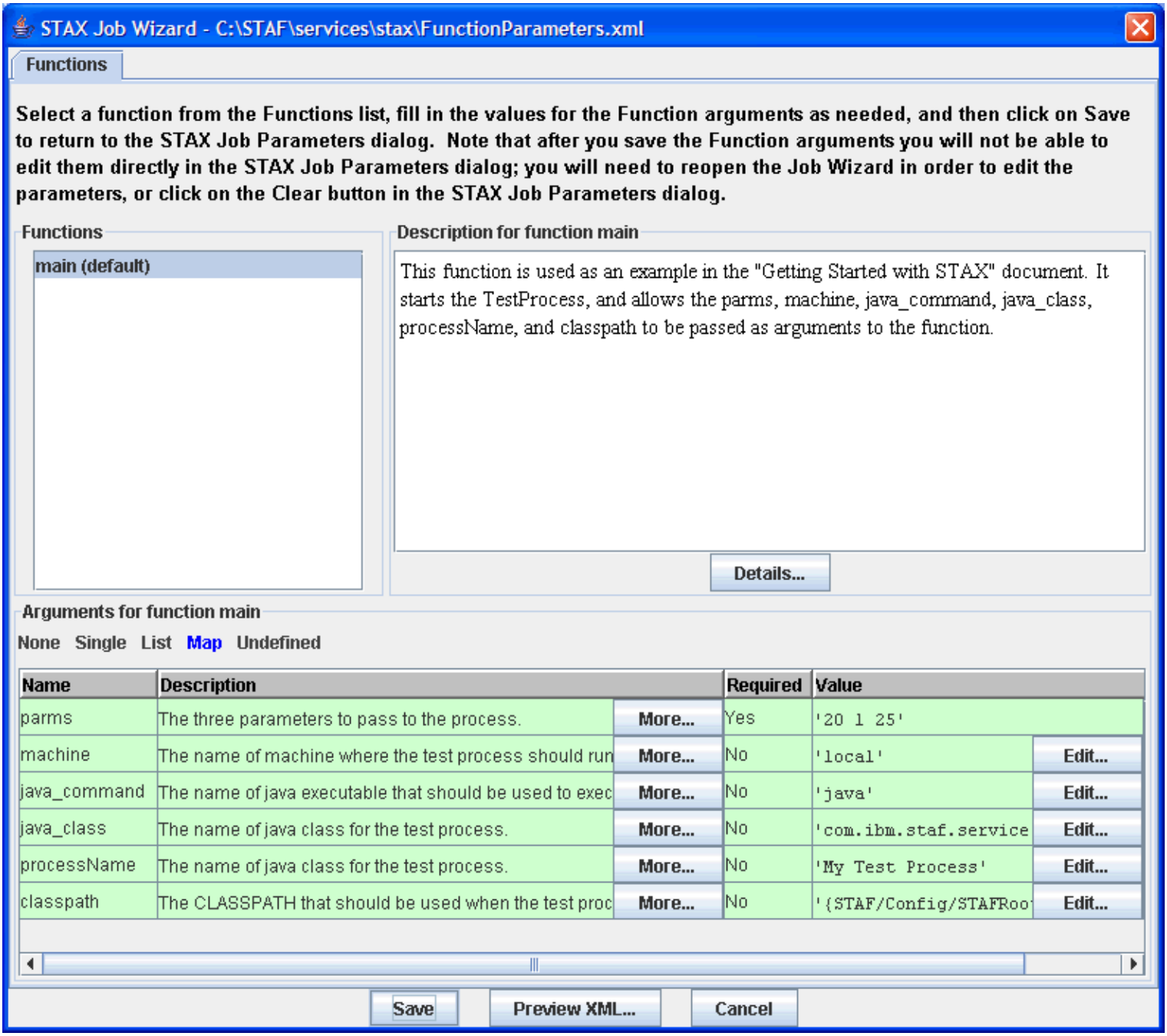
In the "Description for function main" section, we see the text that we specified for **function-prolog**.

In the "Arguments for function main" section, we see that the function accepts a Map of arguments, and the six arguments are shown in the table. Arguments with a light red background are required, while arguments with a light green background are optional, and the default value is shown.

Next, click in the "Value" column for the "parms" argument and type the string (and then press enter to commit the changes):

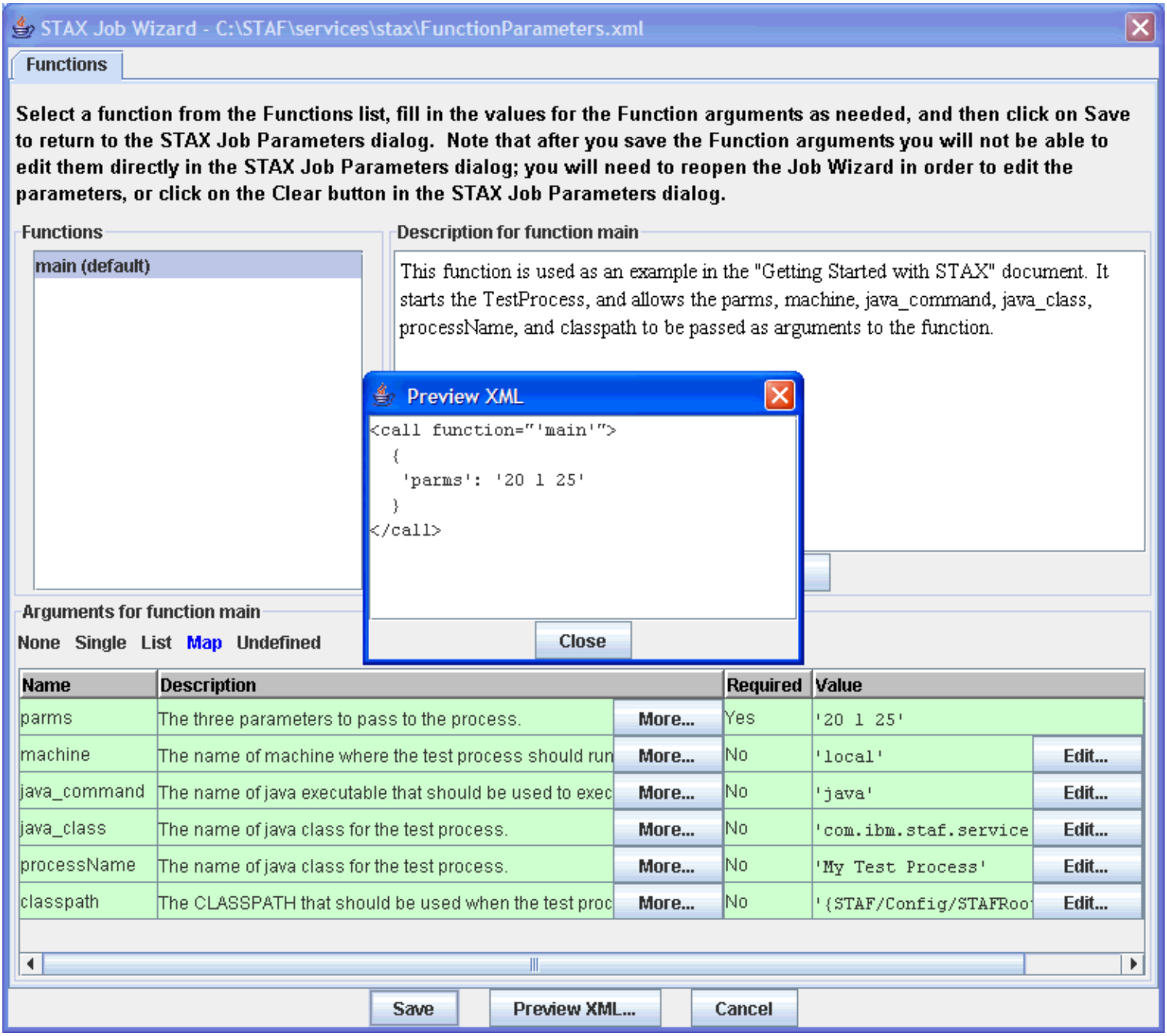
```
'20 1 25'
```

Figure 32.



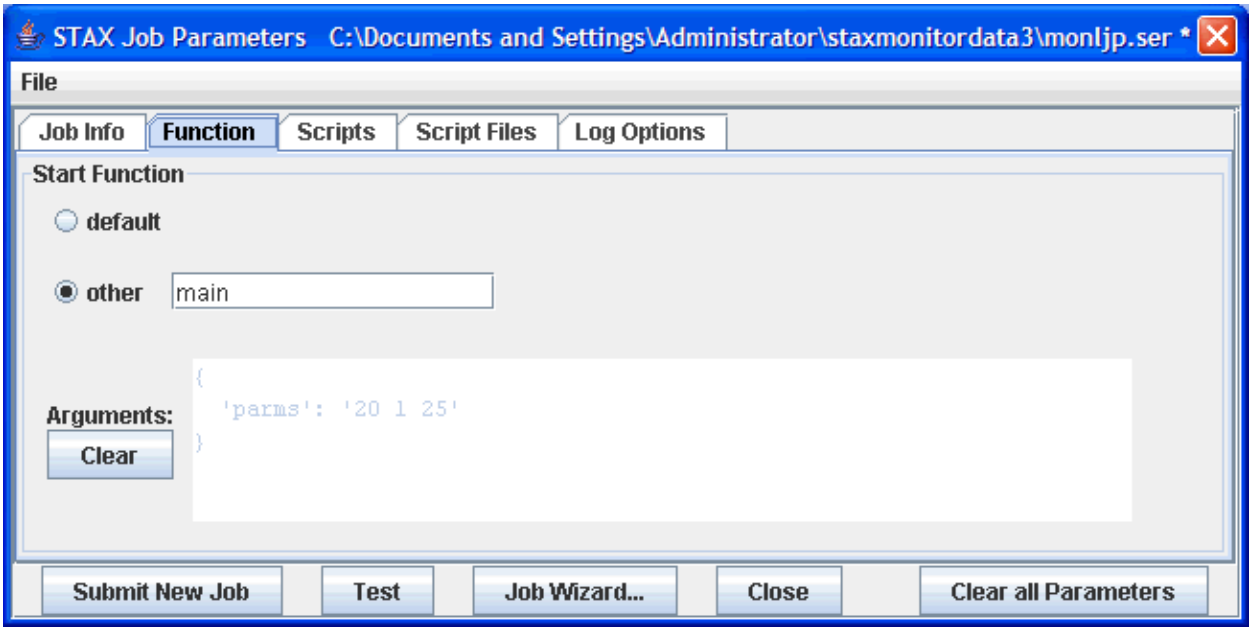
Notice that after you fill in the argument for "parms", it now has a light green background. Click on the "Preview XML" button, which displays the **call** information for the function:

Figure 33.



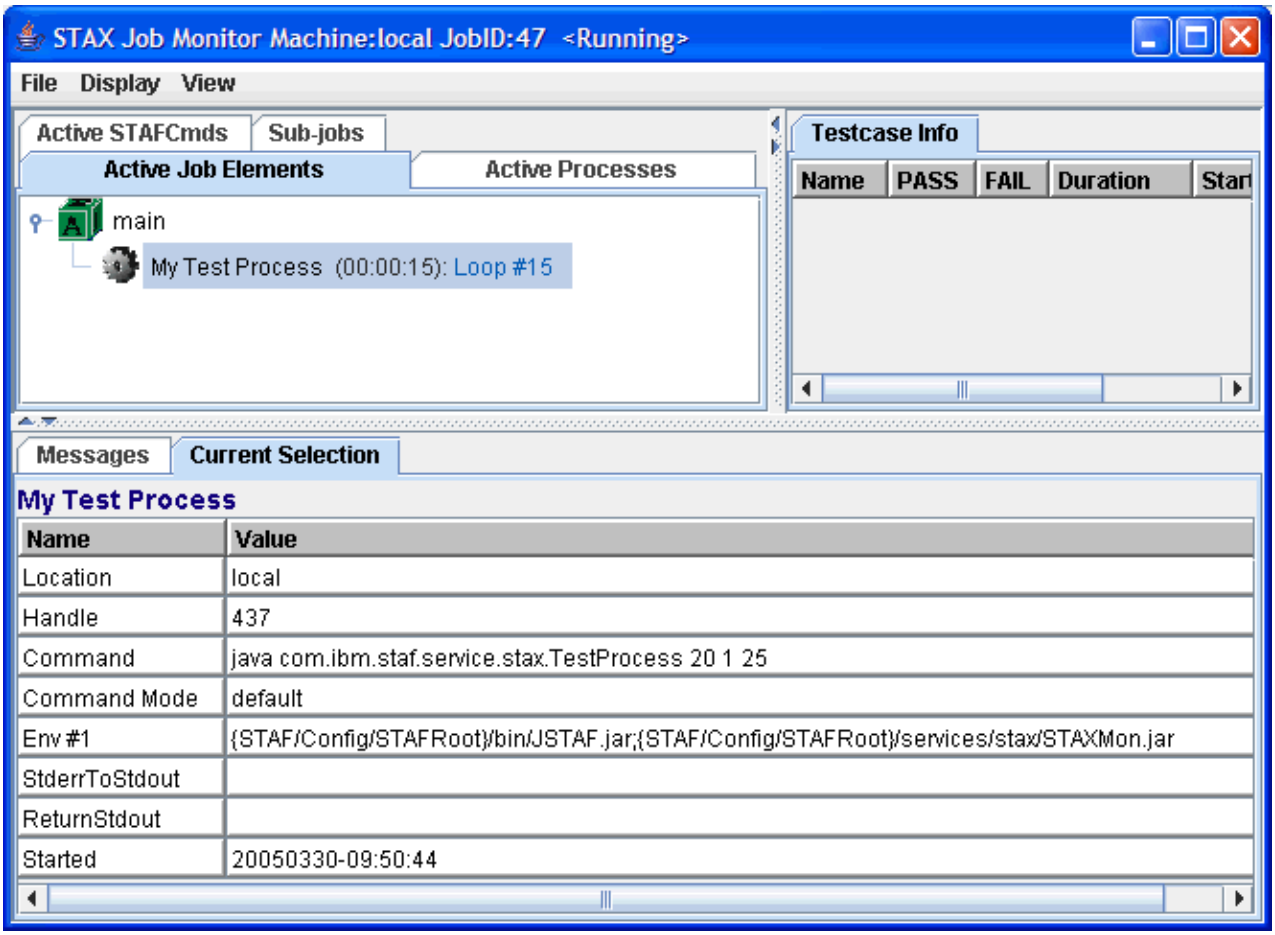
Click on the "Close" button. Now you are ready to save the arguments, so click on the "Save" button, and then click on the "Yes" button in the confirmation dialog. Next, click on the "Function" tab, and note that these are the arguments that will be passed to the "main" function:

Figure 34.



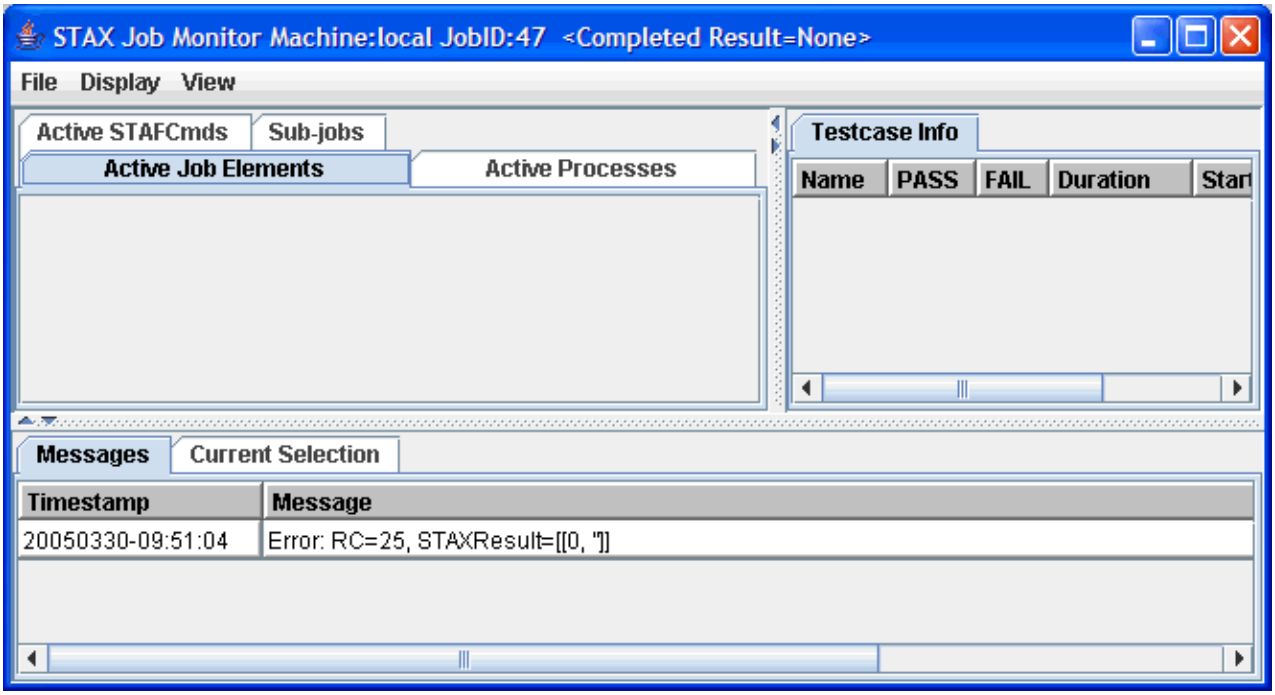
Next, click on the "Test" button to check the STAX job for XML and Python errors. You should see a "Validation Successful" popup. Next, click on the "Submit New Job" button. You should see a new STAX Monitor window for this STAX job. Your STAX Monitor window should look like the following (click on "My Test Process" in the Active Job Elements tree to get the details about the process):

Figure 35.



After the job completes, click on the Messages tab:

Figure 36.



7.9. Adding logging to the STAX job

Next let's add logging to the STAX job. At the same time we'll add more information to the messages and logs, so that we can more easily distinguish between the processes.

```
1:   <?xml version="1.0" encoding="UTF-8" standalone="no"?>
2:   <!DOCTYPE stax SYSTEM "stax.dtd">
3:
4:   <stax>
5:
6:     <defaultcall function="main"/>
7:
8:     <function name="main">
9:
10:      <function-prolog>
11:        This function is used as an example in the "Getting Started with STAX"
12:        document.  It starts the TestProcess, and allows the parms, machine,
13:        java_command, java_class, processName, and classpath to be passed as
14:        arguments to the function.
15:      </function-prolog>
16:
17:      <function-map-args>
18:        <function-required-arg name="parms">
19:          The three parameters to pass to the process.
20:        </function-required-arg>
21:        <function-optional-arg name="machine" default="'local'">
22:          The name of machine where the test process should run.
23:        </function-optional-arg>
24:        <function-optional-arg name="java_command" default="'java'">
25:          The name of java executable that should be used to execute the test
26:          process.
27:        </function-optional-arg>
28:        <function-optional-arg name="java_class"
29:            default="'com.ibm.staf.service.stax.TestProcess'">
30:          The name of java class for the test process.
31:        </function-optional-arg>
32:        <function-optional-arg name="processName" default="'My Test Process'">
```

```
33:          The name of the process.
34:        </function-optional-arg>
35:        <function-optional-arg name="classpath"
36:
default="'{STAF/Config/STAFRoot}/bin/JSTAF.jar;{STAF/Config/STAFRoot}/services/stax/STAXMon.jar'">
37:          The CLASSPATH that should be used when the test process is started..
38:        </function-optional-arg>
39:      </function-map-args>
40:
41:      <sequence>
42:
43:        <process name="'%s with parms %s' % (processName, parms)">
44:          <location>machine</location>
45:          <command>'%s %s %s' % (java_command, java_class, parms)</command>
46:          <env>'CLASSPATH=%s' % classpath</env>
47:          <stderr mode="'stdout'"/>
48:          <returnstdout/>
49:        </process>
50:
51:        <if expr="RC != 0">
52:          <message log="1" level="'Error'">
53:            '%s with parms %s Error: RC=%s, STAXResult=%s' % \
54:              (processName, parms, RC, STAXResult)
55:          </message>
56:          <else>
57:            <message log="1">
58:              'SUCCESS: %s with parms %s\nSTAXResult=%s' % \
59:                (processName, parms, STAXResult)
60:            </message>
61:          </else>
62:        </if>
63:
64:        <return>RC</return>
65:
66:      </sequence>
67:
68:    </function>
69:
70:  </stax>
```

Notice on lines 52 and 57, we have added the "log" attribute, and set the value to 1 (true). This means that the message will also be written to the STAX job user log.

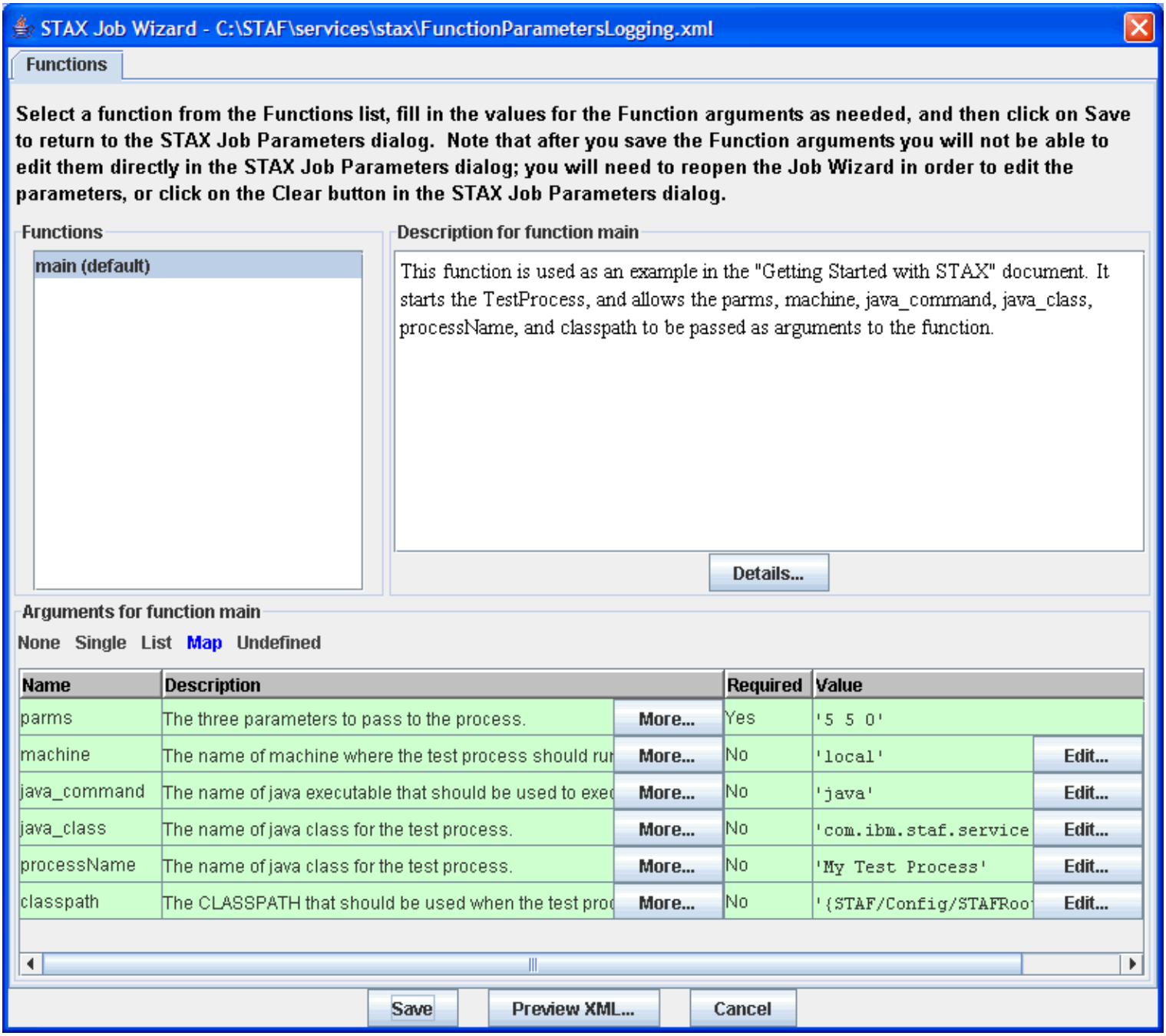
Each time you run a STAX job, there are 2 log associated with it. The first is the STAX job log, and it is only written to by the STAX service itself. It contains information such as start and stop messages, signals, and a summary of testcase passes/fails when the job completes. The second is the STAX job user log, and this is where you get to log information. Note that there is a separate **log** element that you can use, but if you want the information to be written to the STAX Monitor and the STAX job user log, it is simpler to just use the "log" attribute with the **message** element.

Also notice on line 64 we have added a **return** element, which will return the RC to the caller.

Open your favorite text editor, or use an XML editor such as Cooktop, copy/paste lines 1-70 (delete the first five characters on each line, to get rid of the line numbers and whitespace), and save the file as FunctionParametersLogging.xml. Go to the main STAX Monitor panel, and click on the "Submit New Job..." button. For "XML Job File" "Filename:", enter the full path to the FunctionParametersLogging.xml file.

Next click on the "Job Wizard..." button. The parms value should still be '20 1 25'. Click in the text area and change the value to '5 5 0' and press enter to commit the changes.

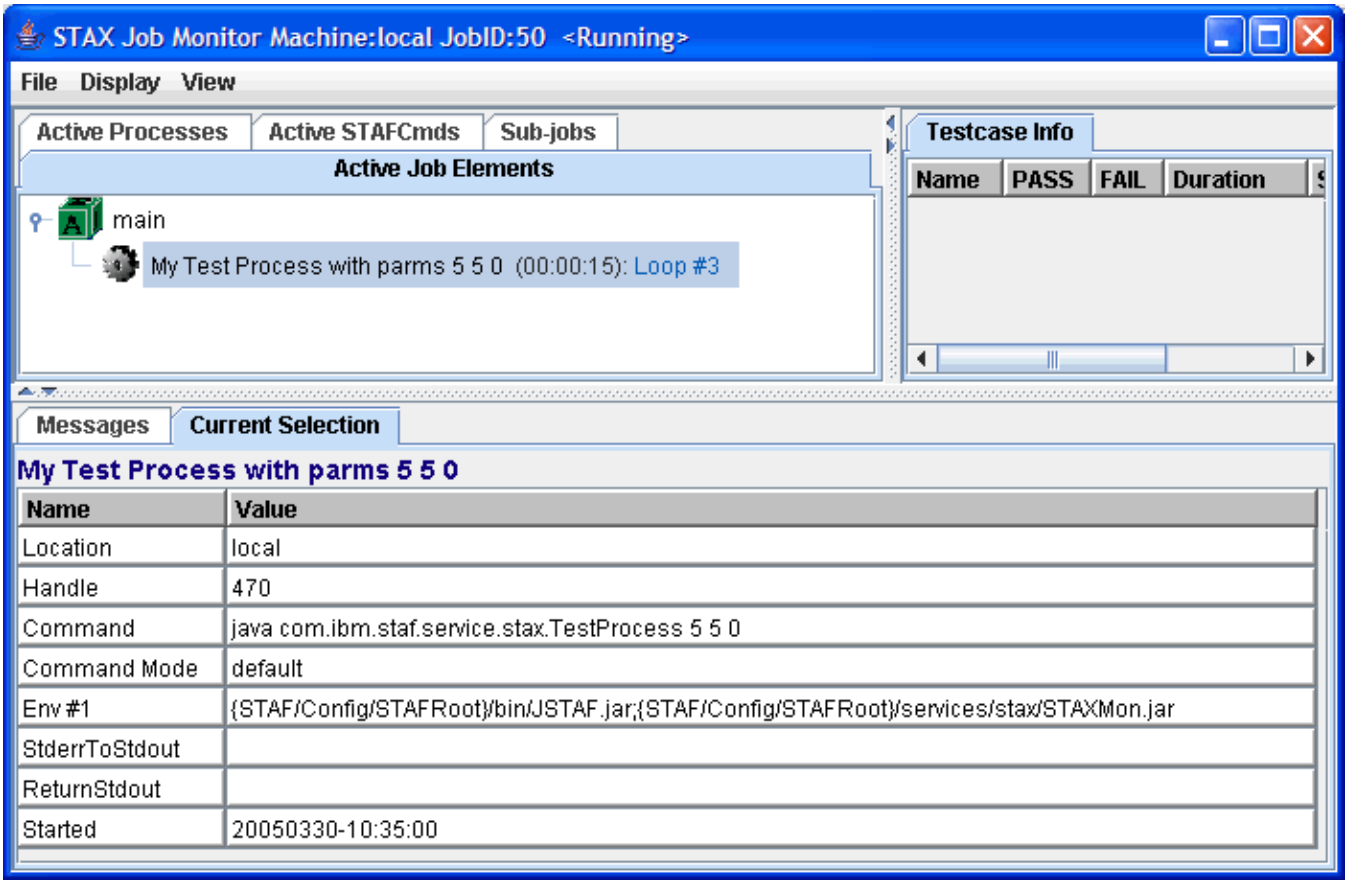
Figure 37.



Click on the "Save" button, and then click on the "Yes" button in the confirmation dialog.

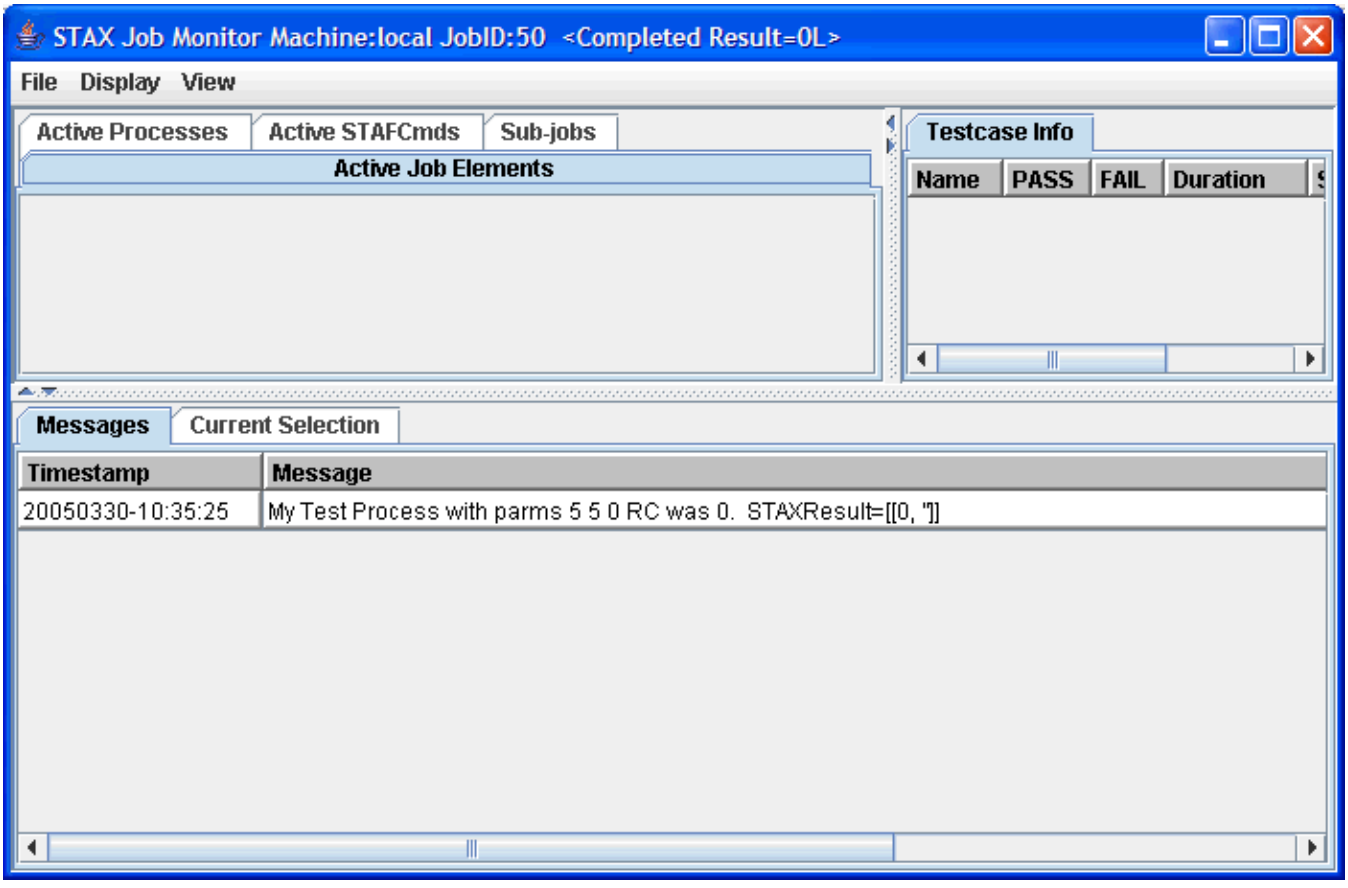
Next, click on the "Test" button to check the STAX job for XML and Python errors. You should see a "Validation Successful" popup. Next, click on the "Submit New Job" button. You should see a new STAX Monitor window for this STAX job. Your STAX Monitor window should look like the following (click on "My Test Process with parms 5 5 0" in the Active Job Elements tree to get the details about the process):

Figure 38.



After the job completes, click on the Messages tab:

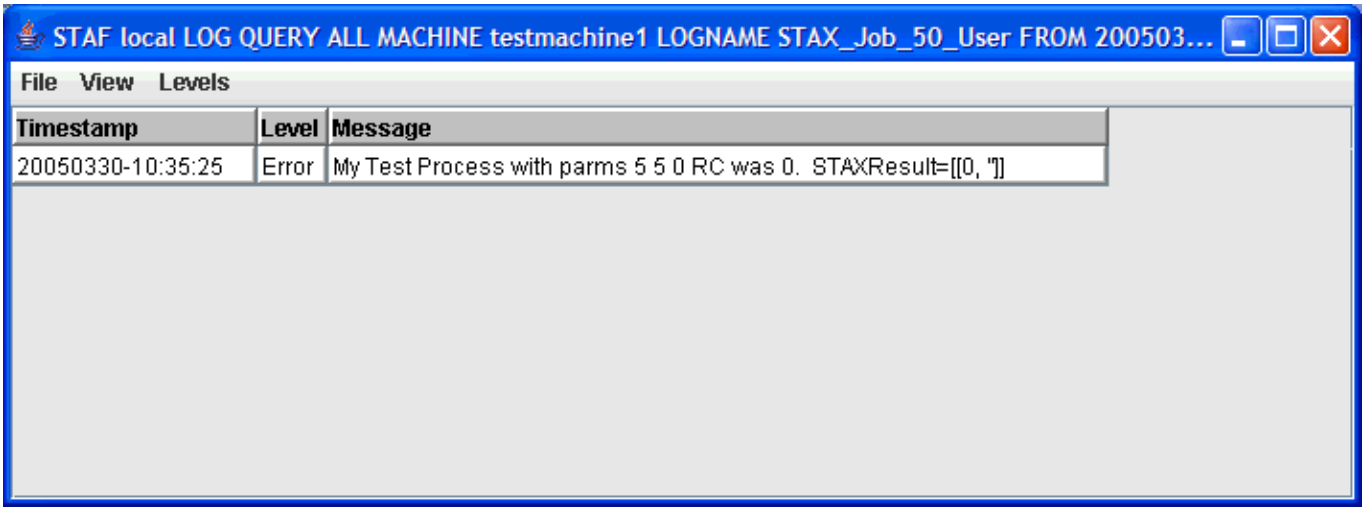
Figure 39.



Notice in the title bar that the Result is included. This is because we added the **return** to the job. The RC, returned from the **process**, is a Python Long object, which is why it is displayed as "0L". We could simply convert it to a string object, but we will leave it as it is since we will be calling this function from another STAX job.

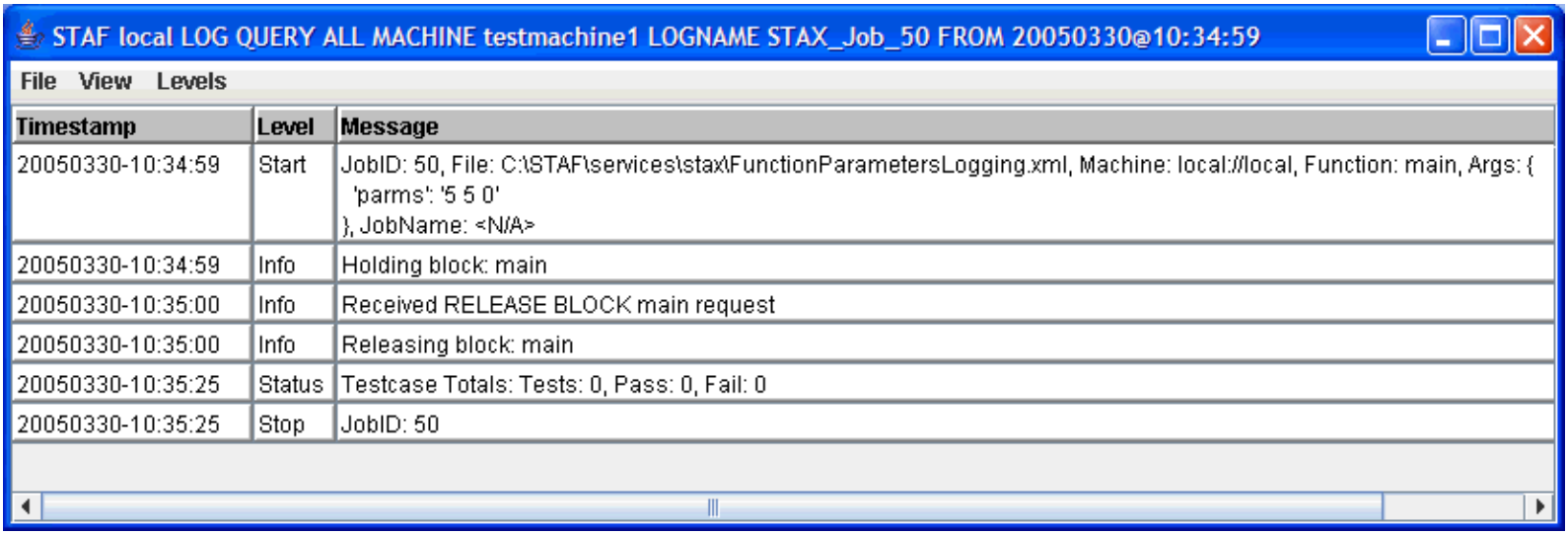
To view the STAX job user log, you can click on "Display" in the menu bar, and then "Display Job User Log". You should see the following dialog:

Figure 40.



To view the STAX job log, you can click on "Display" in the menu bar, and then "Display Job Log". You should see the following dialog:

Figure 41.



7.10. Importing and calling a STAX function

Now we will create a STAX job that imports FunctionParametersLogging.xml (the previous example) and calls the "main" function several times:

```
1:   <?xml version="1.0" encoding="UTF-8" standalone="no"?>
2:   <!DOCTYPE stax SYSTEM "stax.dtd">
3:
4:   <stax>
5:
6:     <defaultcall function="begin_tests"/>
7:
8:     <script>
9:       ImportMachine = 'local'
10:      ImportDirectory = '{STAF/Config/STAFRoot}/services/stax'
11:    </script>
12:
13:    <function name="begin_tests">
14:
15:      <sequence>
16:
17:        <import machine="ImportMachine"
18:          file="'%s/FunctionParametersLogging.xml' % ImportDirectory"/>
19:
20:        <call function="'main'">{ 'parms' : '9 2 7' }</call>
21:
22:        <call function="'main'">{ 'parms' : '2 9 15' }</call>
23:
24:      </sequence>
25:
26:    </function>
27:
28:  </stax>
```

On line 17 we have an **import** element which is used to import functions from other STAX xml files. In this example we are importing all of the functions from file {STAF/Config/STAFRoot}/services/stax/FunctionParametersLogging.xml (on the local machine).

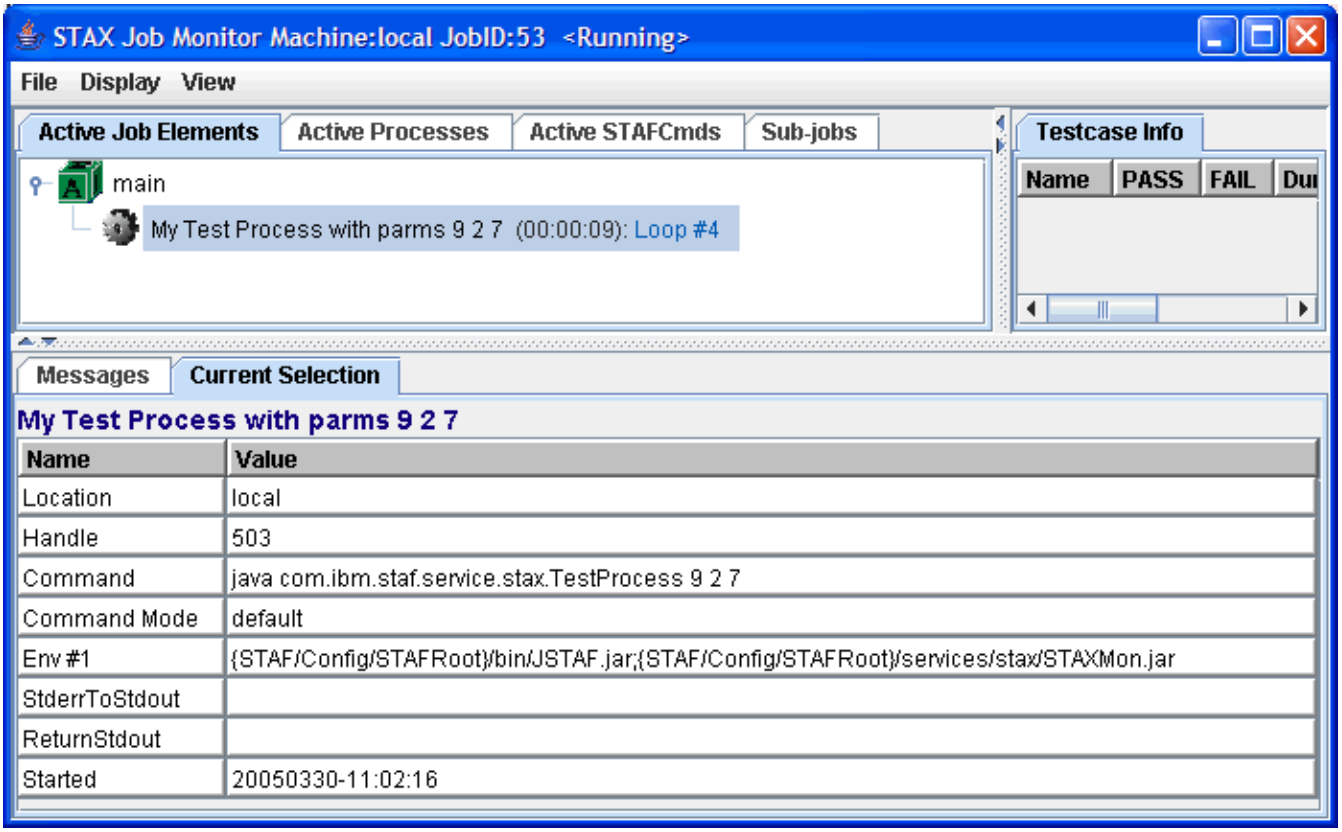
On line 20 we have a **call** element which calls the "main" function, and passes '9 2 7' as the 'parms' argument.

On line 22 we have a **call** element which calls the "main" function, and passes '2 9 15' as the 'parms' argument.

Open your favorite text editor, or use an XML editor such as Cooktop, copy/paste lines 1-28 (delete the first five characters on each line, to get rid of the line numbers and whitespace), and save the file as ImportFunction.xml. Go to the main STAX Monitor panel, and click on the "Submit New Job..." button. For "XML Job File" "Filename:", enter the full path to the ImportFunction.xml file. In the "Functions" tab, click on the "Default" radio button, and click on the "Arguments:" "Clear" button.

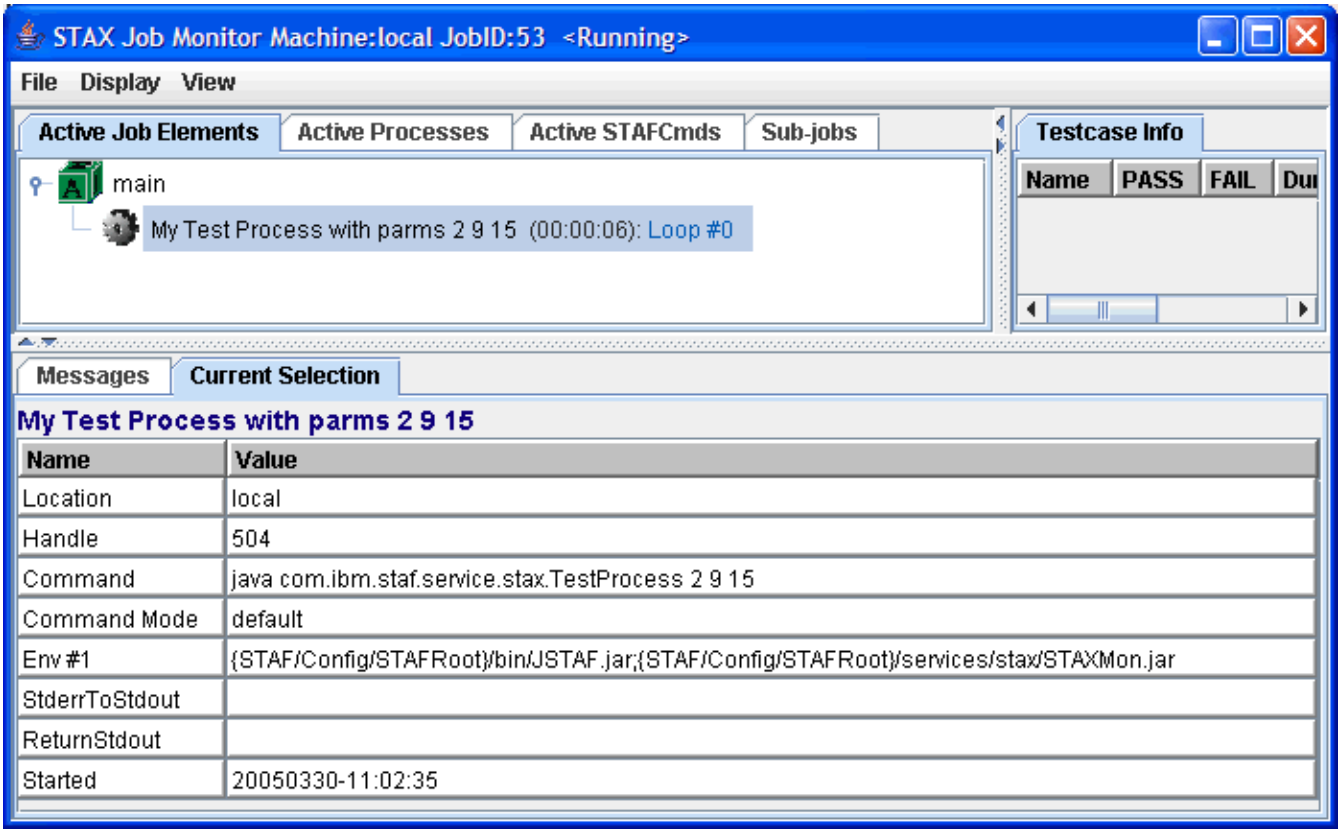
Next, click on the "Test" button to check the STAX job for XML and Python errors. You should see a "Validation Successful" popup. Next, click on the "Submit New Job" button. You should see a new STAX Monitor window for this STAX job. Your STAX Monitor window should look like the following

Figure 42.



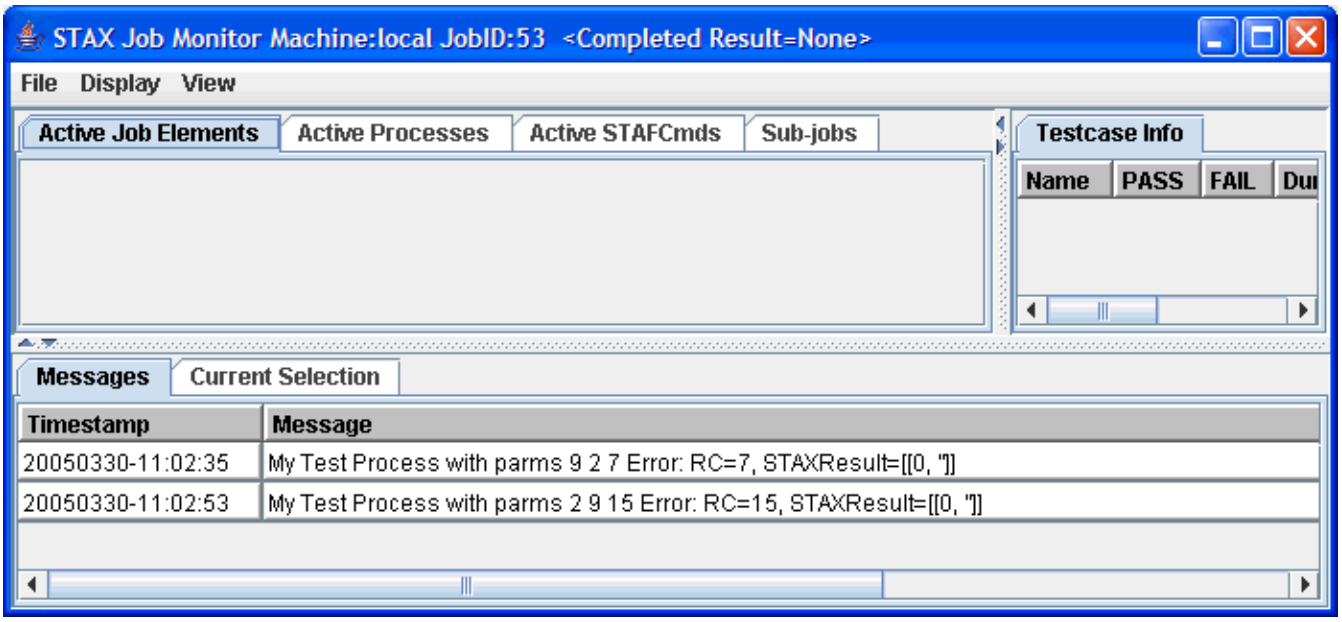
First, the process with parms '9 2 7' is executed, and completes in 18 seconds. Next, the process with parms '2 9 15' is executed:

Figure 43.



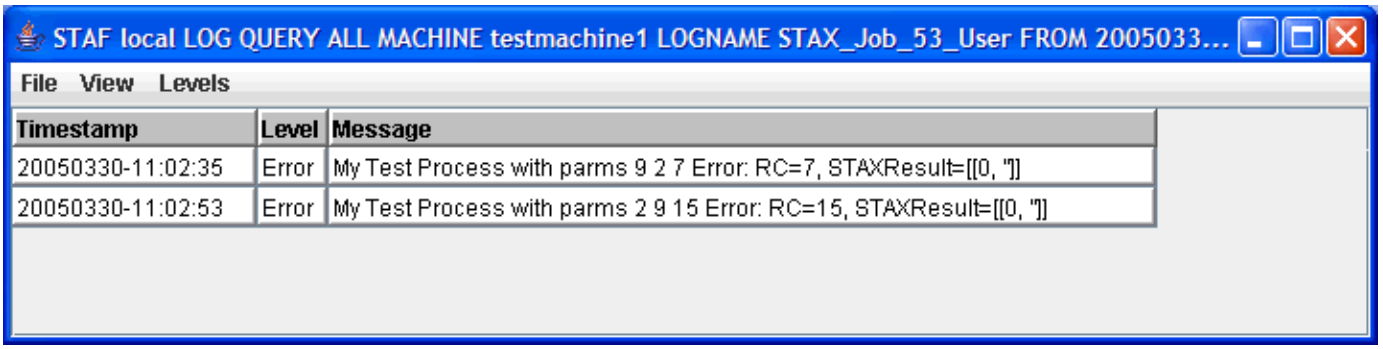
After the second process completes the job is completed:

Figure 44.



To view the STAX job user log, you can click on "Display" in the menu bar, and then "Display Job User Log". You should see the following dialog:

Figure 45.



7.11. Adding execution control into a STAX job

Now we will add execution control to this STAX job:

```
1:   <?xml version="1.0" encoding="UTF-8" standalone="no"?>
2:   <!DOCTYPE stax SYSTEM "stax.dtd">
3:
4:   <stax>
5:
6:     <defaultcall function="begin_tests"/>
7:
8:     <script>
9:       ImportMachine = 'local'
10:      ImportDirectory = '{STAF/Config/STAFRoot}/services/stax'
11:    </script>
12:
13:    <function name="begin_tests">
14:
15:      <block name="'SVT_Regression'">
16:
17:        <sequence>
18:
```

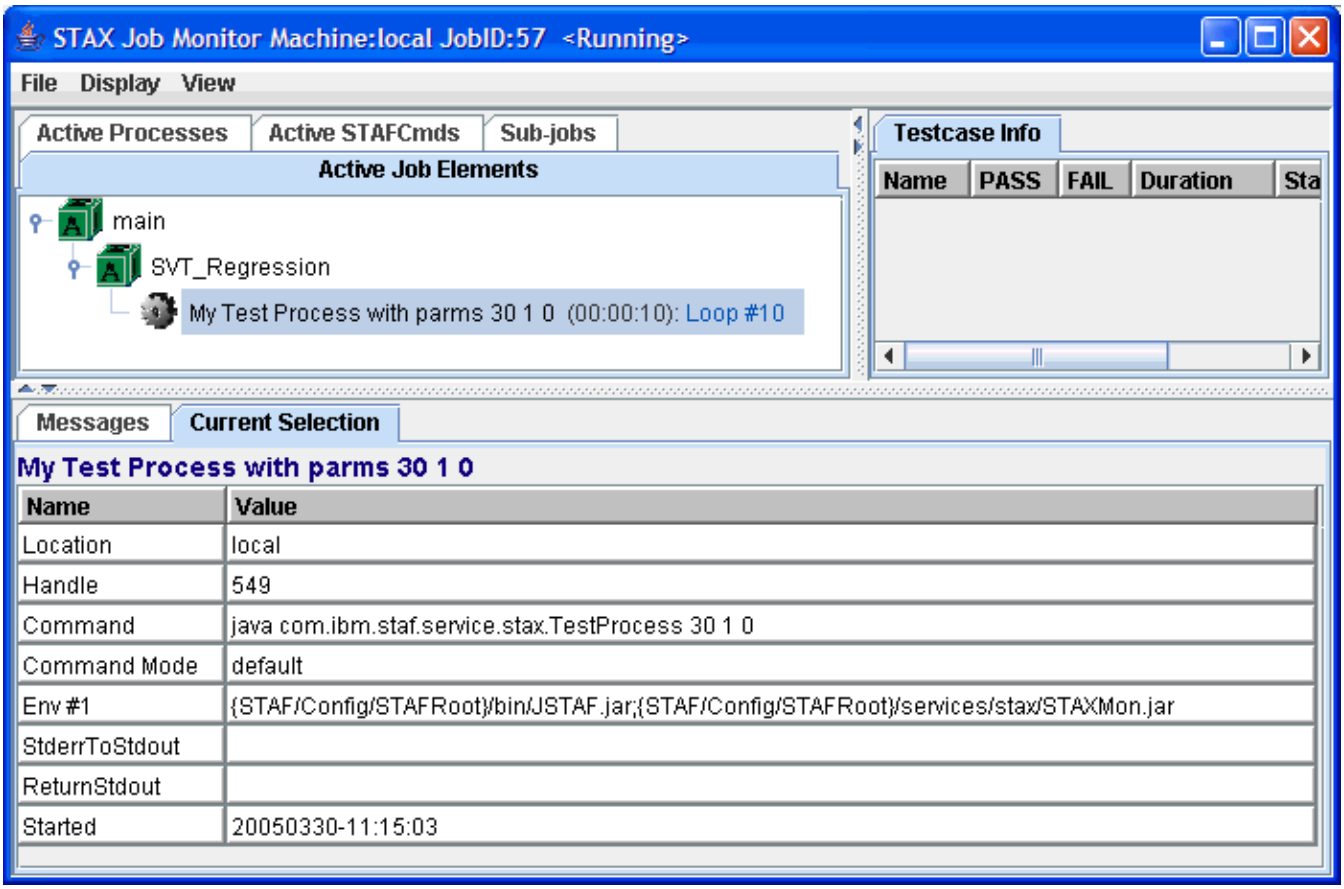
```
19:          <import machine="ImportMachine"
20:            file="'%s/FunctionParametersLogging.xml' % ImportDirectory"/>
21:
22:          <call function="'main'">{ 'parms' : '30 1 0' }</call>
23:
24:          <call function="'main'">{ 'parms' : '15 2 0' }</call>
25:
26:        </sequence>
27:
28:      </block>
29:
30:    </function>
31:
32:  </stax>
```

On line 15 we have defined a **block** element named "SVT_Regression". This will be a child block of the "main" block, which every STAX job will have.

Open your favorite text editor, or use an XML editor such as Cooktop, copy/paste lines 1-32 (delete the first five characters on each line, to get rid of the line numbers and whitespace), and save the file as Block.xml. Go to the main STAX Monitor panel, and click on the "Submit New Job..." button. For "XML Job File" "Filename:", enter the full path to the Block.xml file.

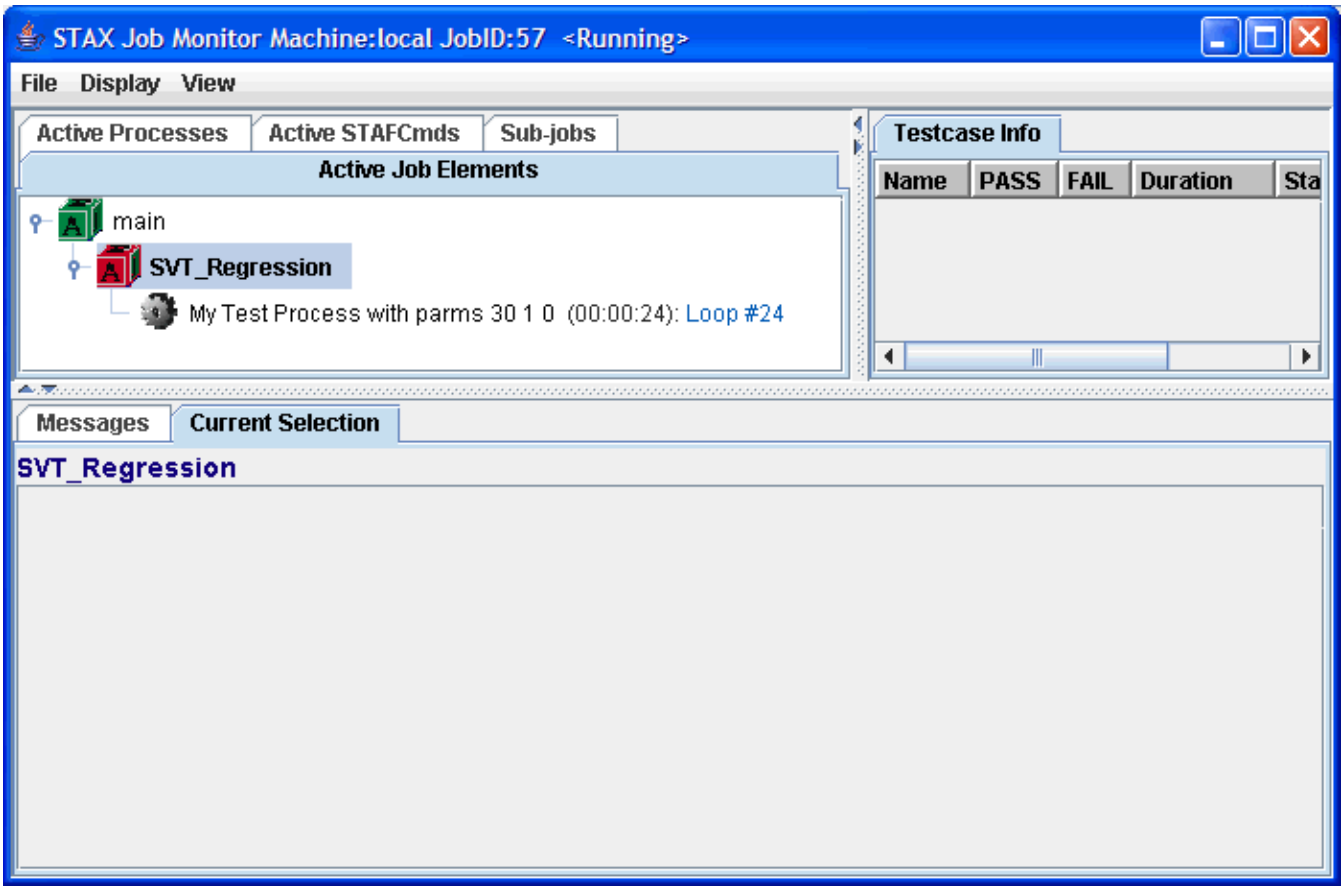
Next, click on the "Test" button to check the STAX job for XML and Python errors. You should see a "Validation Successful" popup. Next, click on the "Submit New Job" button. You should see a new STAX Monitor window for this STAX job. Your STAX Monitor window should look like the following

Figure 46.



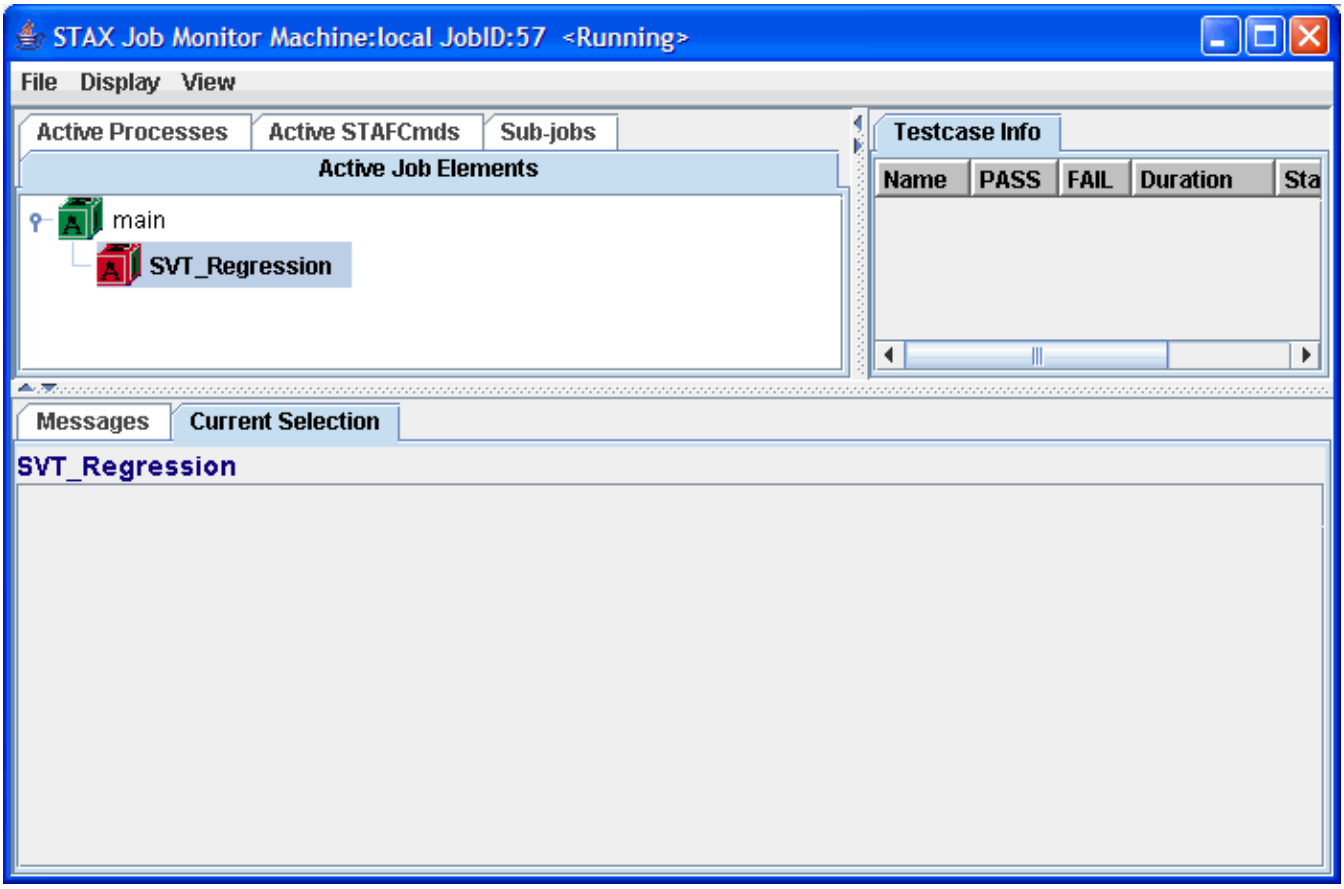
Notice that you see the "SVT_Regression" block in the tree view. The first process will execute for 30 seconds. At around 15-20 seconds into its execution, right click on the "SVT_Regression" block and select "Hold":

Figure 47.



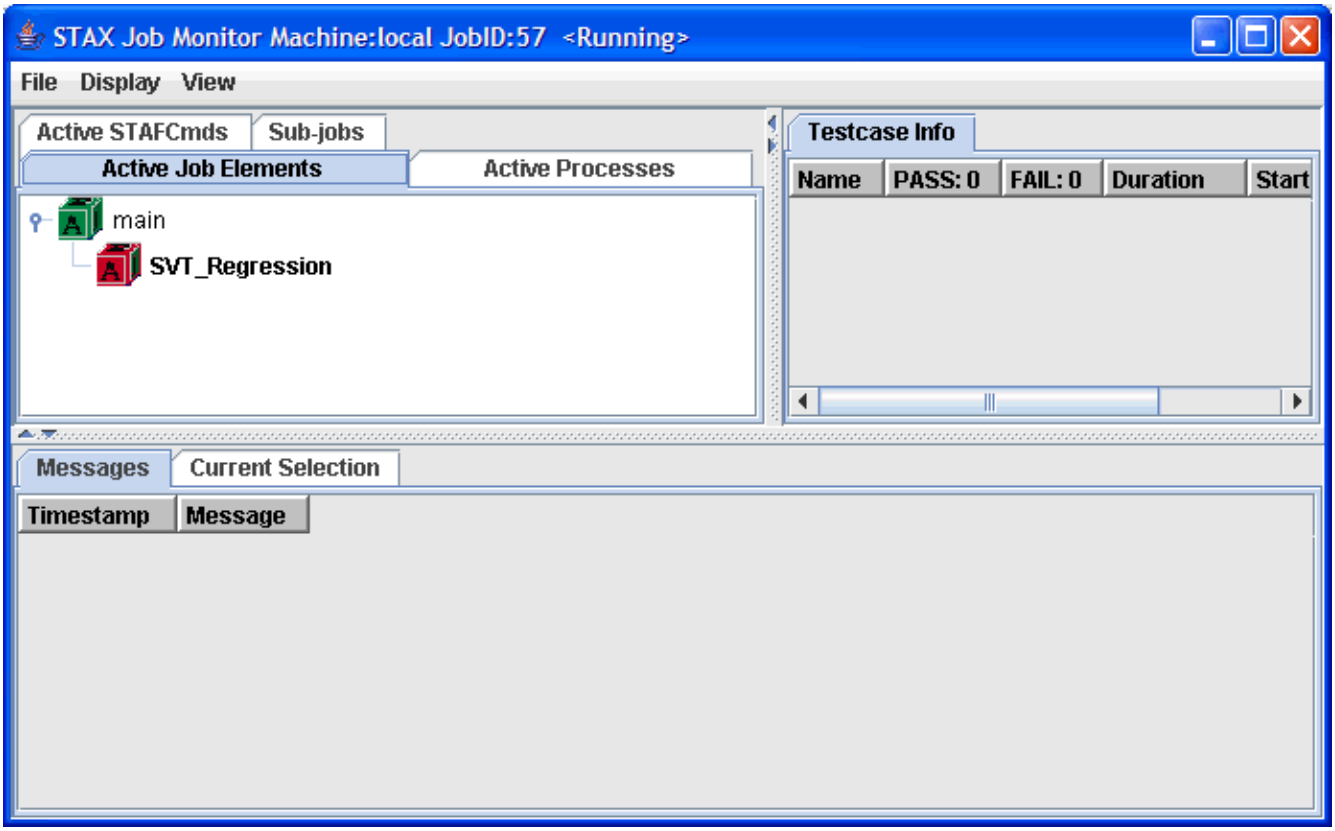
Notice that the "SVT_Regression" block now is red, indicated that it is blocked. But also notice that the first process is still executing. This demonstrates that when a block is held, any currently running processes of stafcmds will complete, but nothing new will begin after they have completed. After 30 seconds of execution, the first process will complete:

Figure 48.



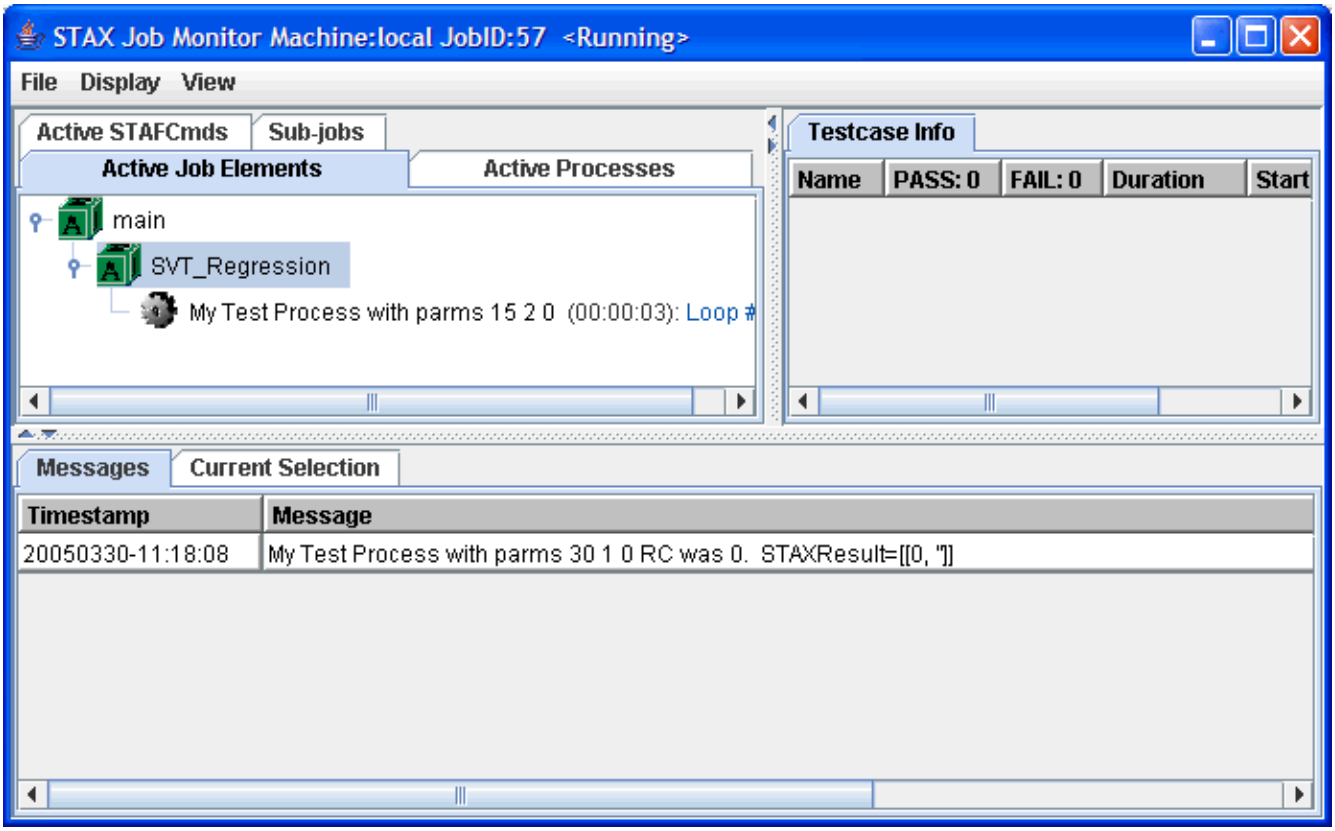
Now close this STAX Monitor window. Back in the main STAX Monitor panel, right click on the job, and select "Start Monitoring". You should see that the job is still held:

Figure 49.



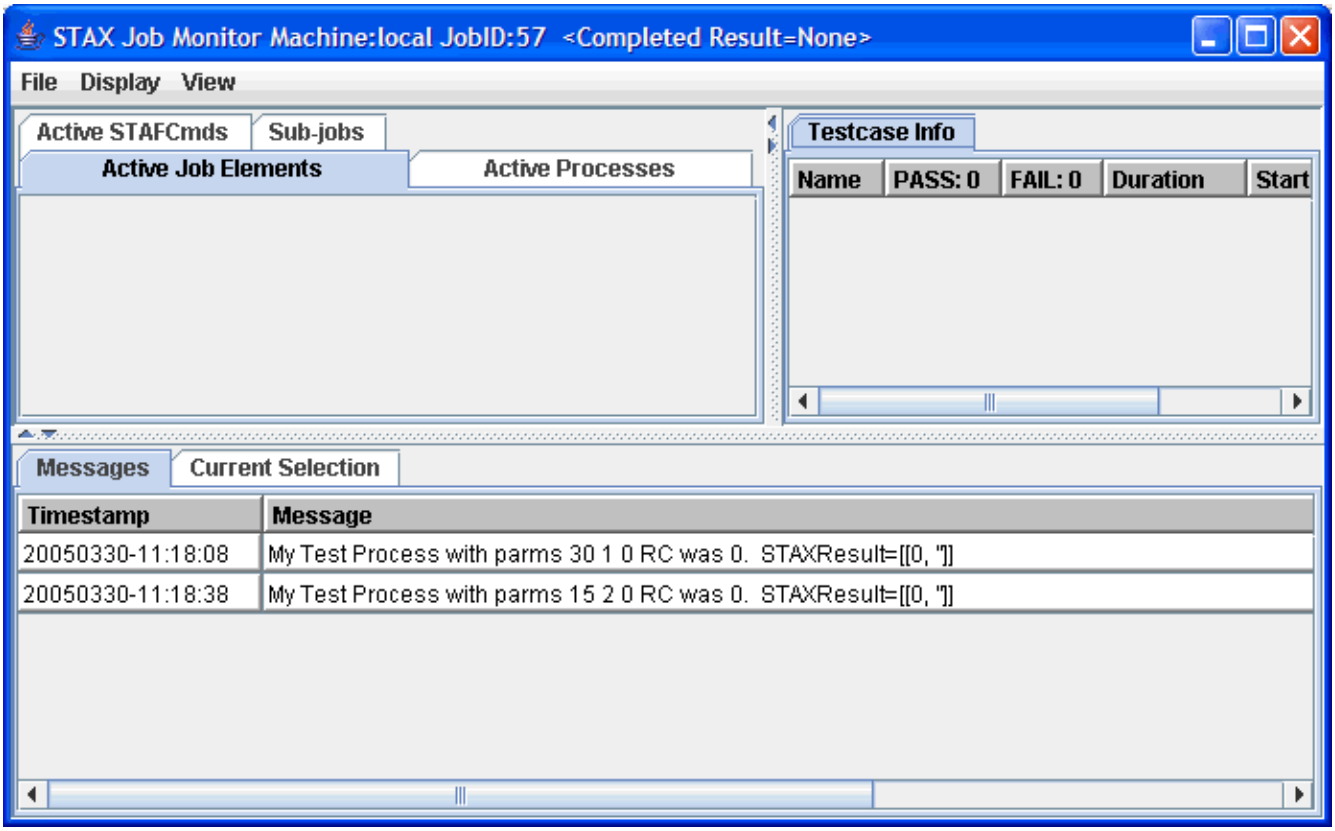
Right click on the "SVT_Regression" block and select "Release". Notice that the second process is now executing:

Figure 50.



After the second process completes, the job is complete:

Figure 51.



7.12. Running tasks in parallel

So far we have only executed tasks sequentially, so now let's look at how you can run tasks in parallel in STAX jobs:

```
1:   <?xml version="1.0" encoding="UTF-8" standalone="no"?>
2:   <!DOCTYPE stax SYSTEM "stax.dtd">
3:
4:   <stax>
5:
6:     <defaultcall function="begin_tests"/>
7:
8:     <script>
9:       ImportMachine = 'local'
10:      ImportDirectory = '{STAF/Config/STAFRoot}/services/stax'
11:    </script>
12:
13:    <function name="begin_tests">
14:
15:      <sequence>
16:
17:        <import machine="ImportMachine"
18:          file="'%s/FunctionParametersLogging.xml' % ImportDirectory"/>
19:
20:        <block name="'Run Processes in Parallel'">
21:
22:          <parallel>
23:
24:            <call function="'main'">{ 'parms' : '40 1 0' }</call>
25:            <call function="'main'">{ 'parms' : '15 2 0' }</call>
26:            <call function="'main'">{ 'parms' : '10 2 0' }</call>
27:
28:          </parallel>
29:
30:        </block>
31:
32:        <call function="'main'">{ 'parms' : '5 3 0' }</call>
33:
34:      </sequence>
35:
36:    </function>
37:
38:  </stax>
```

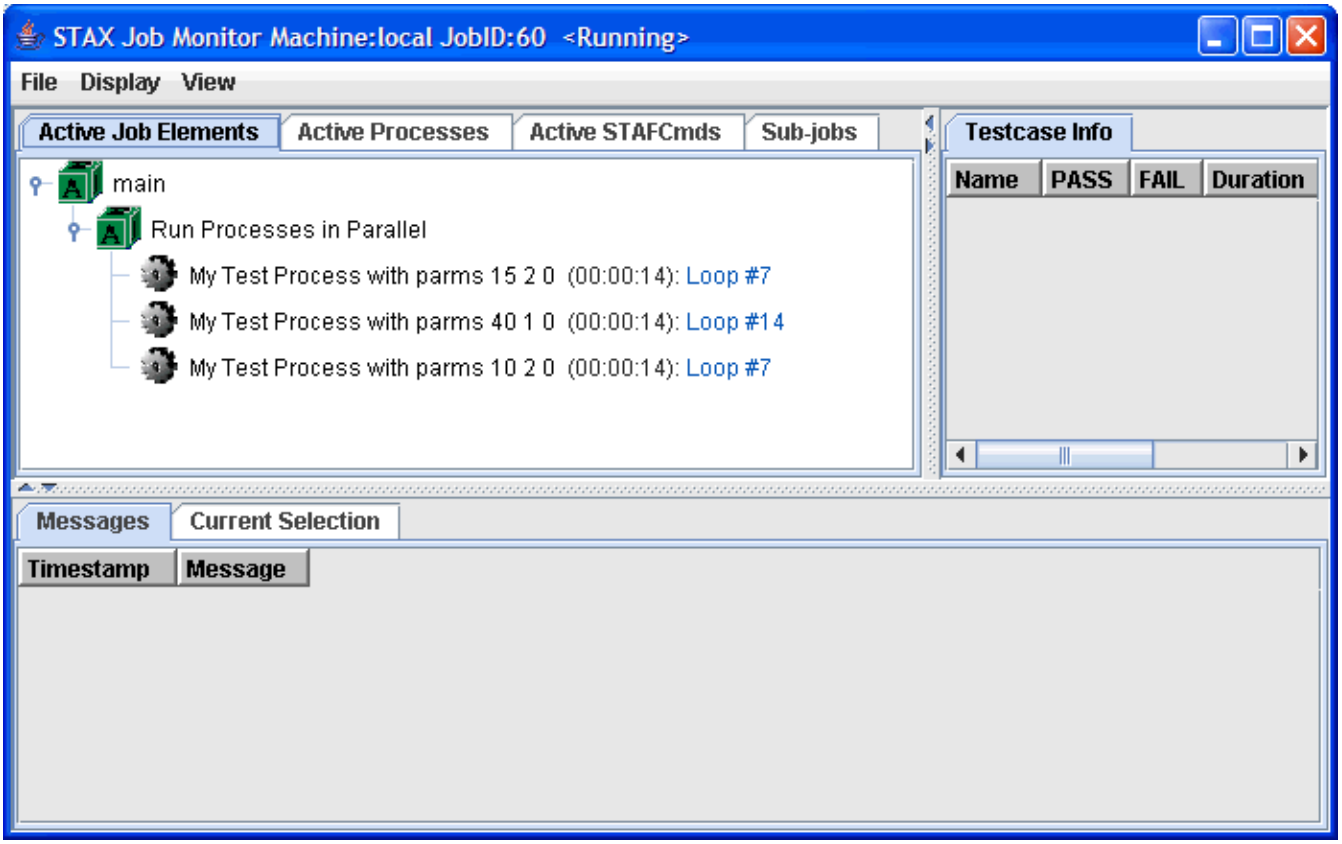
On line 15 we have defined a **sequence** that will first execute an **import**, then execute a **block**, and then execute a **call**.

In the "Run Processes in Parallel" block, at line 20 we have a **parallel** element. It contains three **call** elements that will be executed in parallel.

Open your favorite text editor, or use an XML editor such as Cooktop, copy/paste lines 1-38 (delete the first five characters on each line, to get rid of the line numbers and whitespace), and save the file as Parallel.xml. Go to the main STAX Monitor panel, and click on the "Submit New Job..." button. For "XML Job File" "Filename:", enter the full path to the Parallel.xml file.

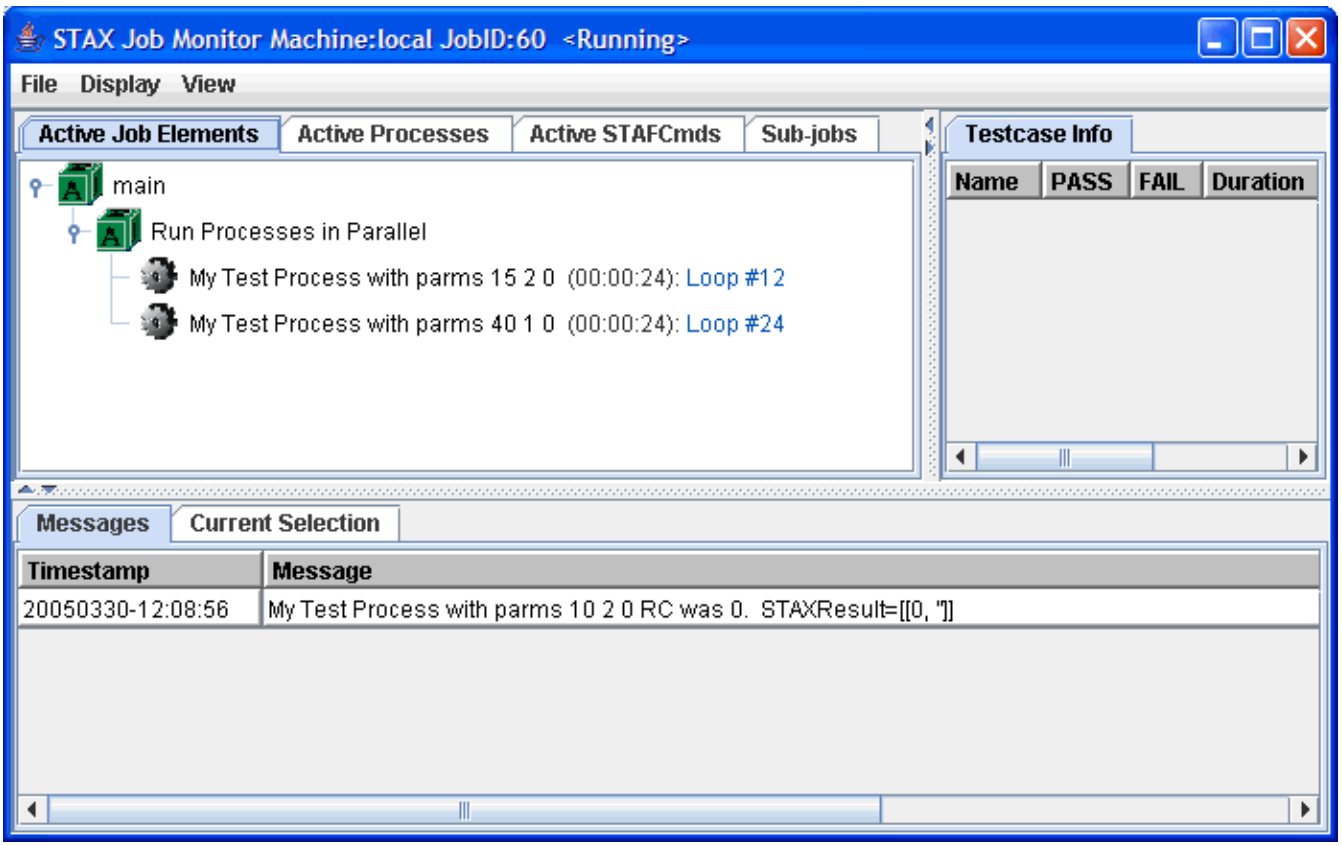
Next, click on the "Test" button to check the STAX job for XML and Python errors. You should see a "Validation Successful" popup. Next, click on the "Submit New Job" button. You should see a new STAX Monitor window for this STAX job. Your STAX Monitor window should look like the following

Figure 52.



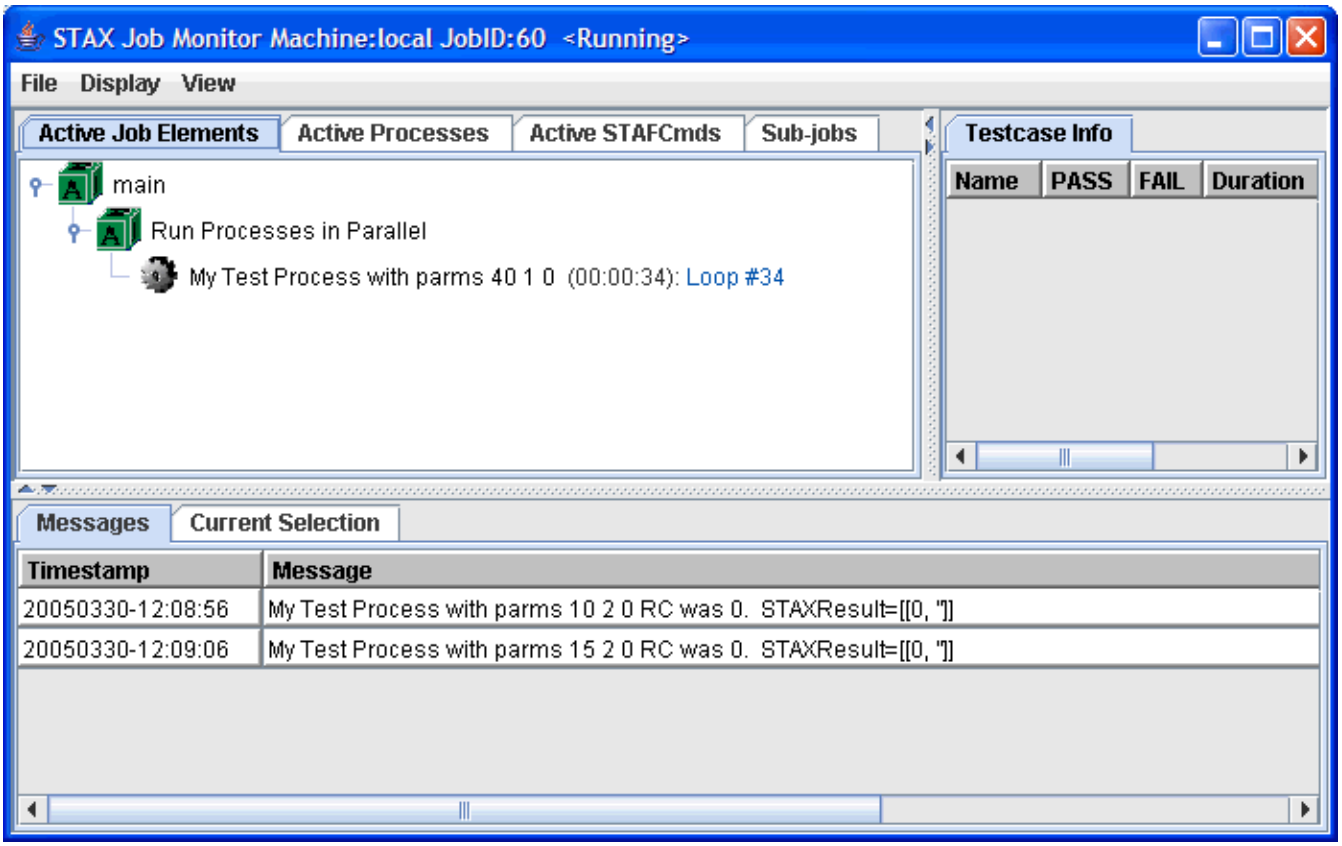
Notice that in the "Run Processes in Parallel" block, the 3 processes are running in parallel. The "10 2 0" process will complete first:

Figure 53.



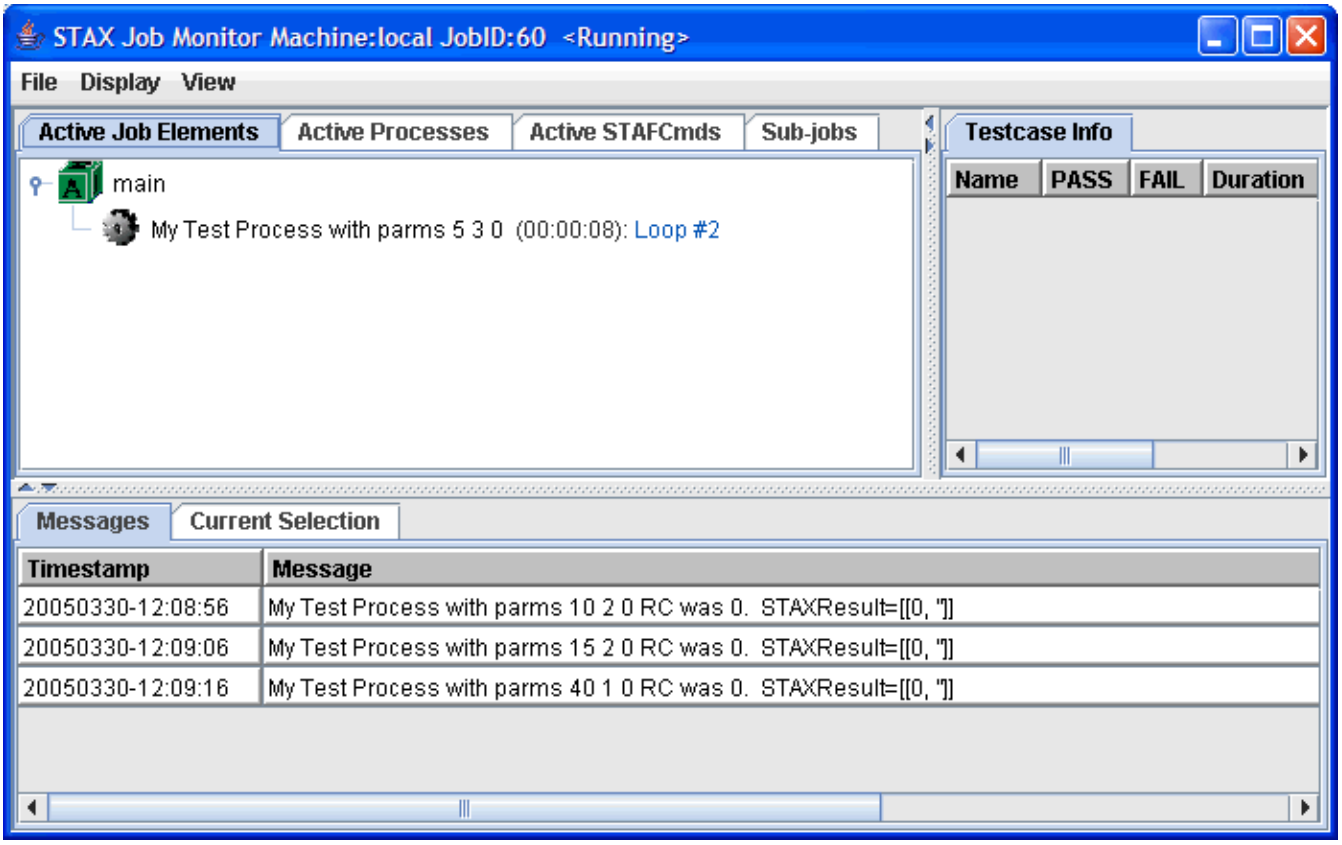
The "15 2 0" process will complete next:

Figure 54.



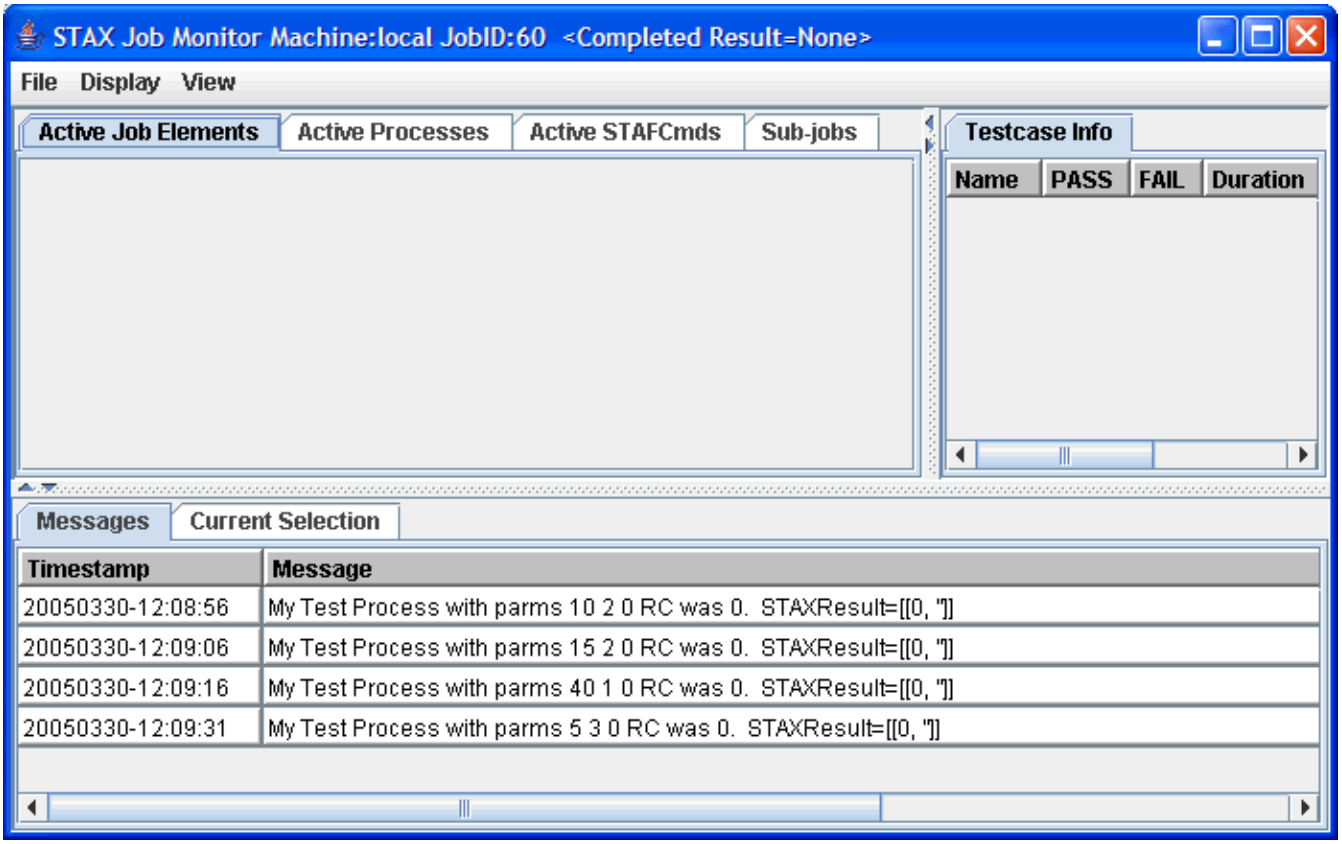
After the "40 1 0" process completes, the **parallel** will be complete, and then the third item in the sequence is executed:

Figure 55.



After the "5 3 0" process completes, the job is complete:

Figure 56.



7.13. Looping in a STAX job

Now let's look at a STAX job that demonstrates how you can use loops within STAX jobs:

```
1:   <?xml version="1.0" encoding="UTF-8" standalone="no"?>
2:   <!DOCTYPE stax SYSTEM "stax.dtd">
3:
4:   <stax>
5:
6:     <defaultcall function="begin_tests"/>
7:
8:     <script>
9:       ImportMachine = 'local'
10:      ImportDirectory = '{STAF/Config/STAFRoot}/services/stax'
11:    </script>
12:
13:    <function name="begin_tests">
14:
15:      <sequence>
16:
17:        <import machine="ImportMachine"
18:          file="'%s/FunctionParametersLogging.xml' % ImportDirectory"/>
19:
20:        <loop from="1" to="3" var="index">
21:
22:          <block name="'Block #%s' % index">
23:
24:            <call function="'main'">{ 'parms' : '10 %s 0' % index }</call>
25:
```

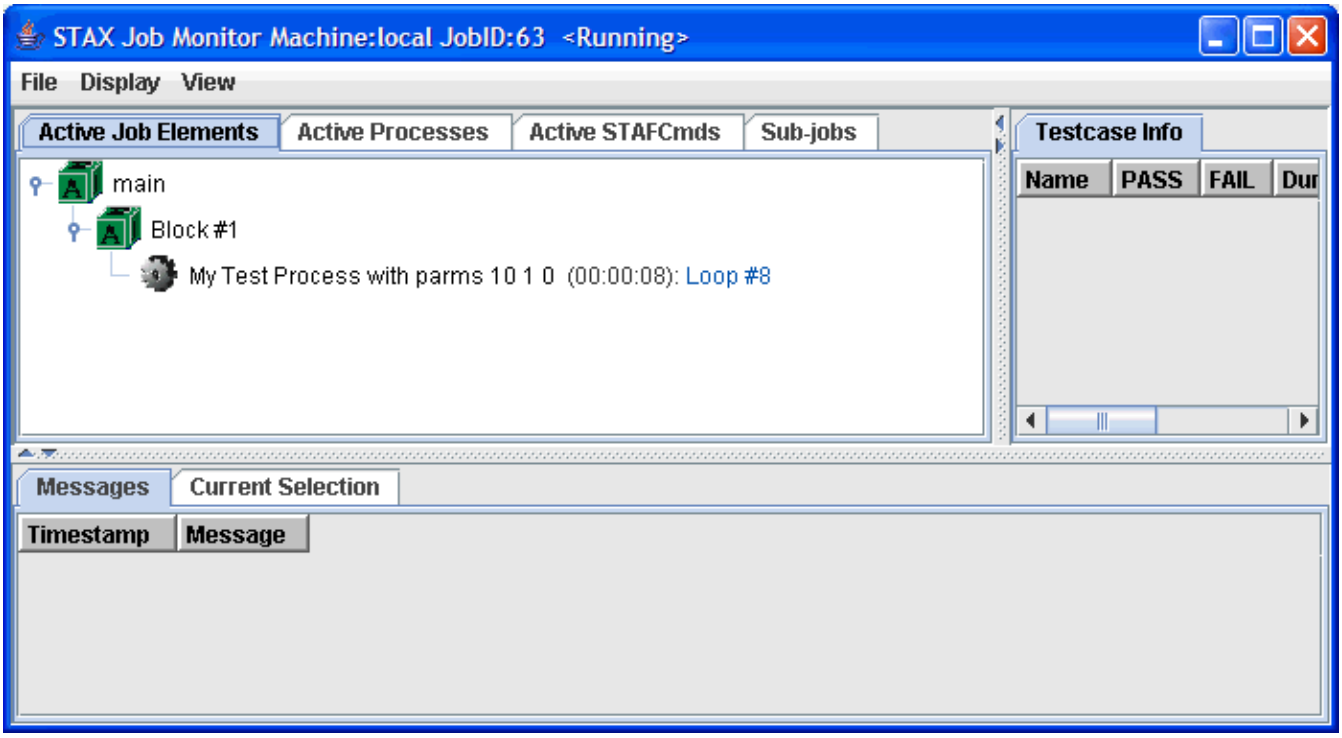
```
26:          </block>
27:
28:        </loop>
29:
30:      </sequence>
31:
32:    </function>
33:
34:  </stax>
```

Notice on line 20 that we have a **loop** element. It is used when you want to repeatedly execute a STAX element. We have specified the "from" attribute as 1 and to the "to" attribute as 3. Notice that we do not use single quotes here, because these are Python integers (not literal strings). This means that the STAX element contained with the **loop** will be executed 3 times. We have also specified the "var" attribute, so that we can use the variable "index" to refer to the loop index. On line 22 we use this variable to create the name of the **block**, and on line 24 it is used as the second parameter to the TestProcess.

Open your favorite text editor, or use an XML editor such as Cooktop, copy/paste lines 1-34 (delete the first five characters on each line, to get rid of the line numbers and whitespace), and save the file as Loop.xml. Go to the main STAX Monitor panel, and click on the "Submit New Job..." button. For "XML Job File" "Filename:", enter the full path to the Loop.xml file.

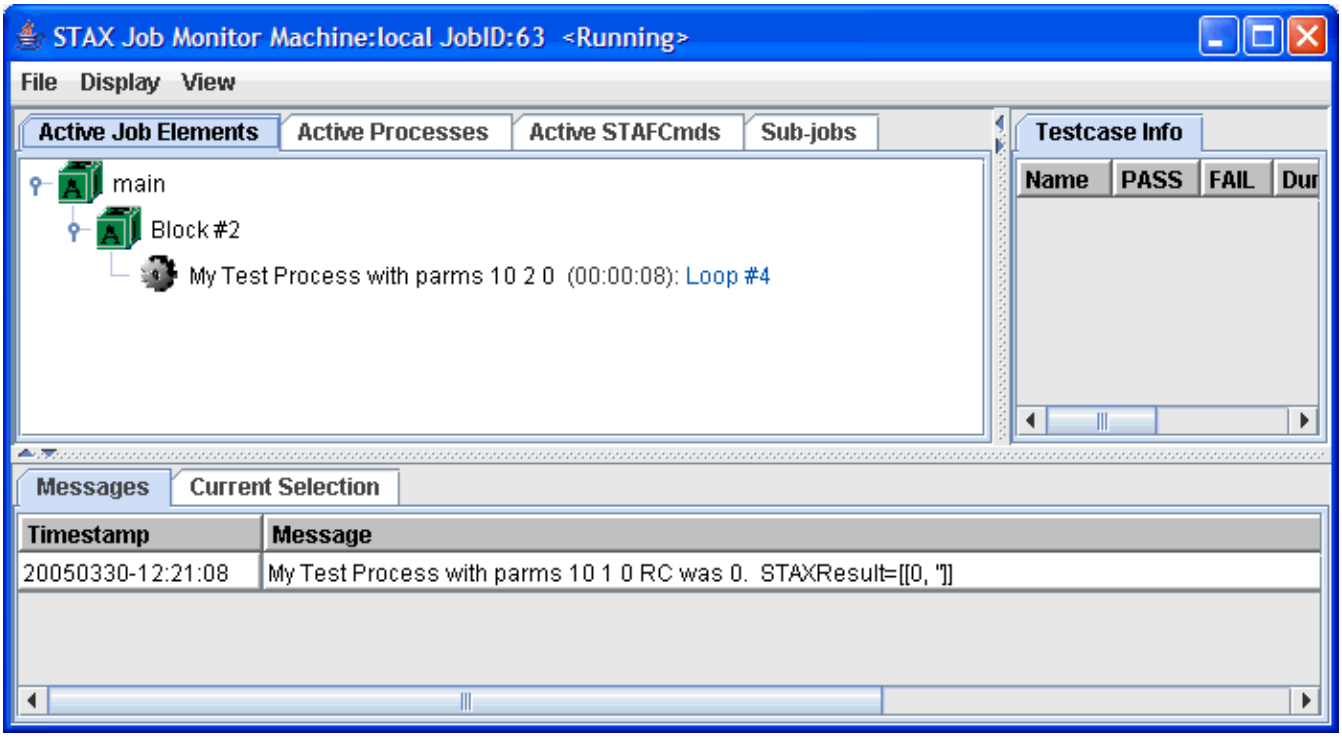
Next, click on the "Test" button to check the STAX job for XML and Python errors. You should see a "Validation Successful" popup. Next, click on the "Submit New Job" button. You should see a new STAX Monitor window for this STAX job. Your STAX Monitor window should look like the following

Figure 57.



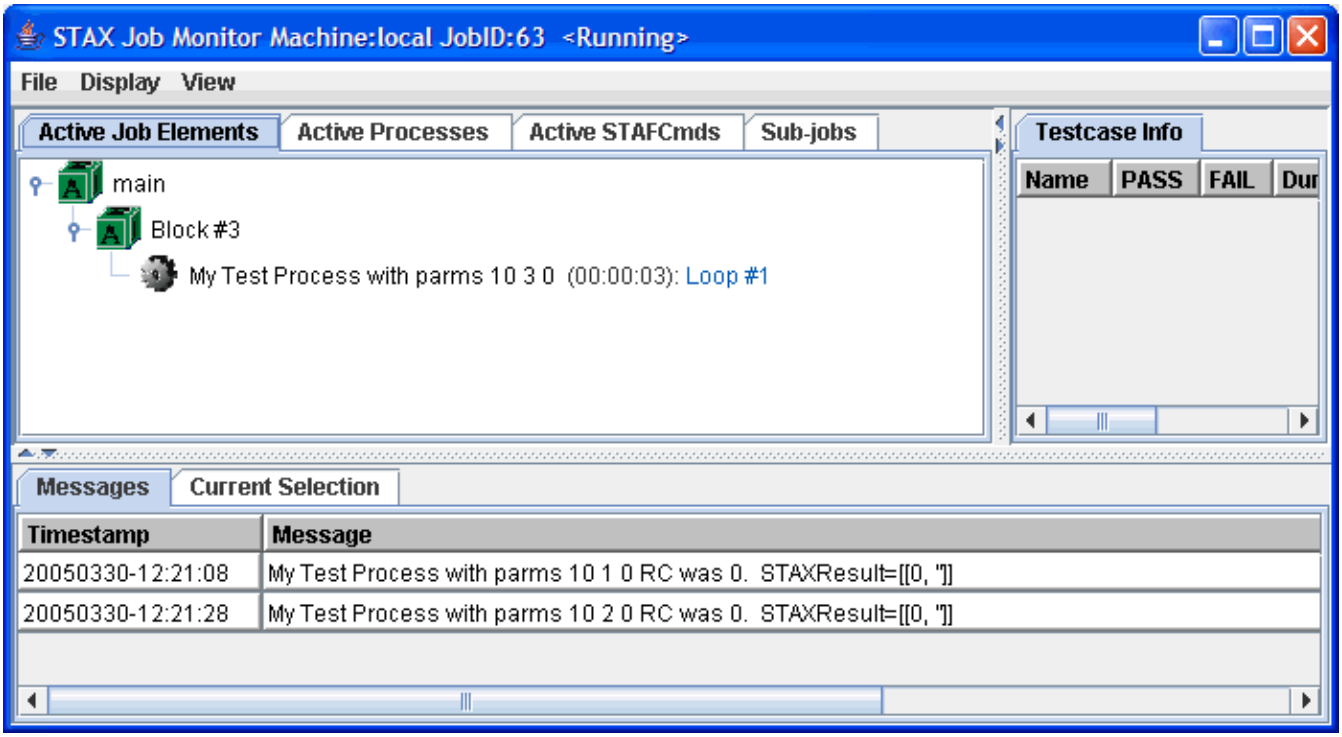
Notice that the block name is "Block #1" and the second parameter for the TestProcess is 1, the current loop index. This process will execute for 10 seconds, and then the loop index variable will be incremented by one (the default):

Figure 58.



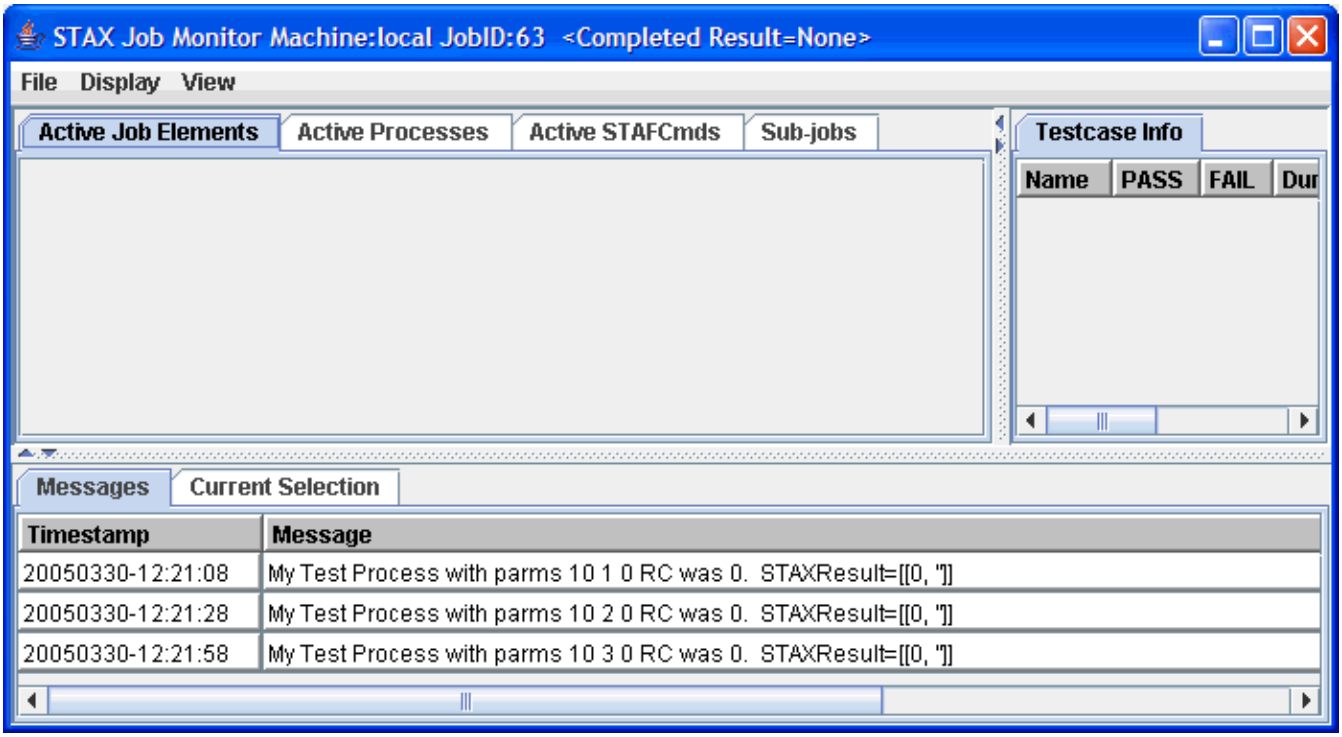
Next the block name is "Block #2" and the second parameter for the TestProcess is 2, the current loop index. This process will execute for 20 seconds, and then the loop index variable will be incremented by one:

Figure 59.



Next the block name is "Block #3" and the second parameter for the TestProcess is 3, the current loop index. This process will execute for 30 seconds, and then the loop index variable will be incremented by one. Since the loop index variable now equals 4, which is greater than the value specified for the "to" attribute, the **loop** is complete, and so the job completes:

Figure 60.



7.14. Adding testcases into a STAX job

Now let's look at a STAX job that demonstrates how you can define testcases within your STAX jobs:

```
1:   <?xml version="1.0" encoding="UTF-8" standalone="no"?>
2:   <!DOCTYPE stax SYSTEM "stax.dtd">
3:
4:   <stax>
5:
6:     <defaultcall function="begin_tests"/>
7:
8:     <script>
9:       ImportMachine = 'local'
10:      ImportDirectory = '{STAF/Config/STAFRoot}/services/stax'
11:      from random import randint
12:    </script>
13:
14:    <function name="begin_tests">
15:
16:      <sequence>
17:
18:        <import machine="ImportMachine"
19:          file="'%s/FunctionParametersLogging.xml' % ImportDirectory"/>
20:
21:        <loop from="1" to="10">
22:
23:          <testcase name="'Test Process'">
24:
25:            <sequence>
26:
27:              <script>r = randint(1, 100)</script>
28:
29:              <call function="'main'">{ 'parms' : '1 1 %s' % r }</call>
30:
31:              <if expr="STAXResult &lt;= 50">
32:                <tcstatus result="'pass'"/>
```

```
33:                <else>
34:                  <tcstatus result="'fail'"/>
35:                </else>
36:              </if>
37:
38:            </sequence>
39:
40:          </testcase>
41:
42:        </loop>
43:
44:      </sequence>
45:
46:    </function>
47:
48:  </stax>
```

This example also demonstrates how you can call Python libraries within your STAX job. In this case we will be using the Python "random" library to generate a random number between 1 and 100, which will be passed to the TestProcess as the return code that it should return. On line 8 we have a **script** element in which we import the "randint" function from the "random" library.

On line 23 we have a **testcase** element named "Test Process". Within this **testcase** element, we can have as many **tcstatus** elements that we need. Each **tcstatus** will increase the testcase's pass/fail count by 1.

On line 27 we have another **script** that creates a random number and stores the value in variable "r". On line 29, we pass this variable as the third parameter to the TestProcess. When the function returns the return code, it will be available as the "STAXResult" variable.

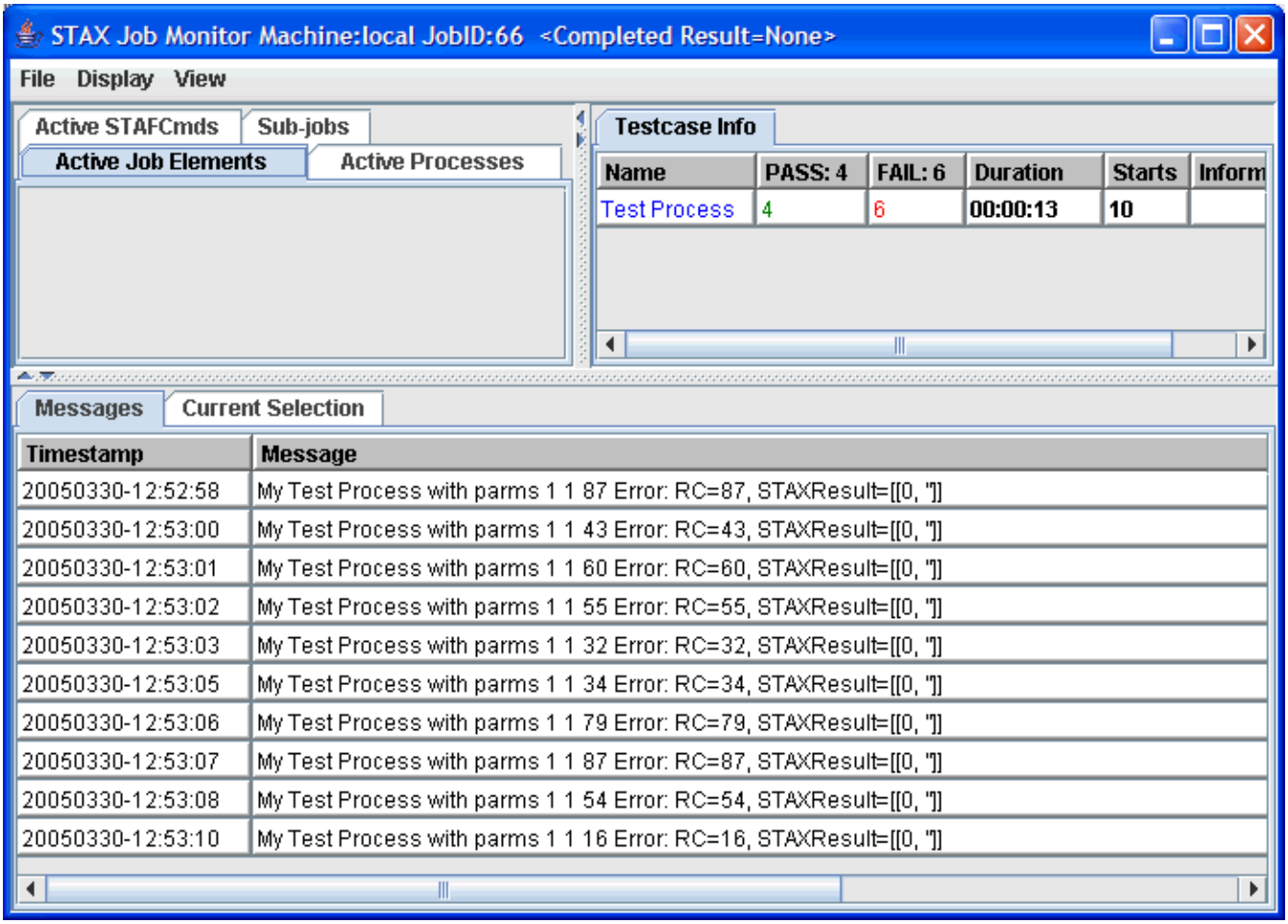
On line 31 we have an **if** element to check if the return code is less than or equal to 50. If it is, then the **tcstatus** element on line 32 will be executed to increment the testcase's pass count. If the return code is greater than 50, then the **tcstatus** element on line 34 will be executed to increment the testcase's fail count.

Notice on line 31, for the "expr" attribute we did not specify "STAXResult <= 50". Since the < character denotes an opening tag for XML, that would be invalid XML syntax. Instead we need to use the special string "<" in place of the <: "STAXResult <= 50".

Open your favorite text editor, or use an XML editor such as Cooktop, copy/paste lines 1-48 (delete the first five characters on each line, to get rid of the line numbers and whitespace), and save the file as Testcase.xml. Go to the main STAX Monitor panel, and click on the "Submit New Job..." button. For "XML Job File" "Filename:", enter the full path to the Testcase.xml file.

Next, click on the "Test" button to check the STAX job for XML and Python errors. You should see a "Validation Successful" popup. Next, click on the "Submit New Job" button. You should see a new STAX Monitor window for this STAX job. After the STAX job completes, your STAX Monitor window should look like the following:

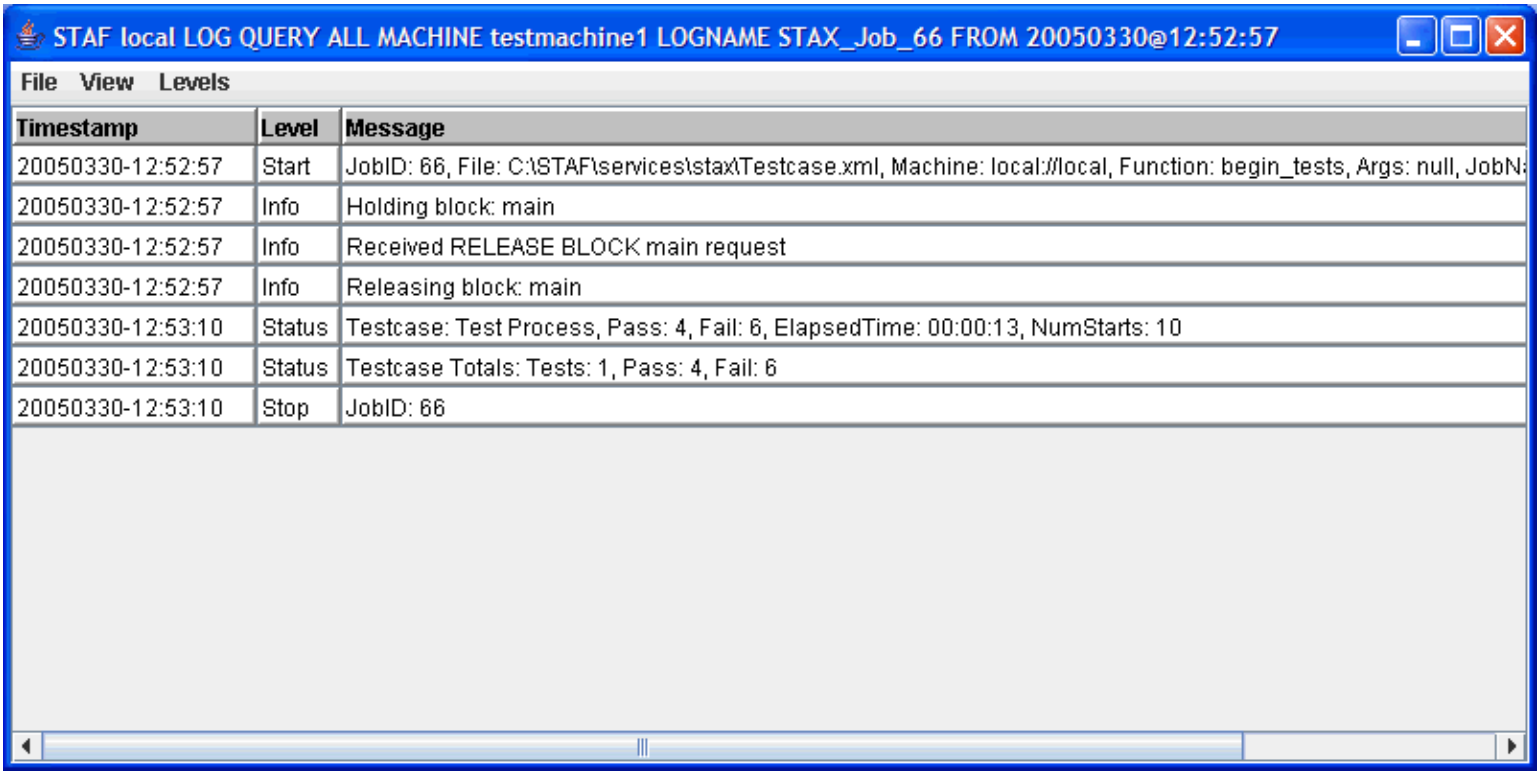
Figure 61.



In the "Testcase Info" tab we see that the testcase named "Test Process", and we see how many passes and fails were recorded. Note that since the return codes were randomly generated, you will not always have 5 passes and 5 fails.

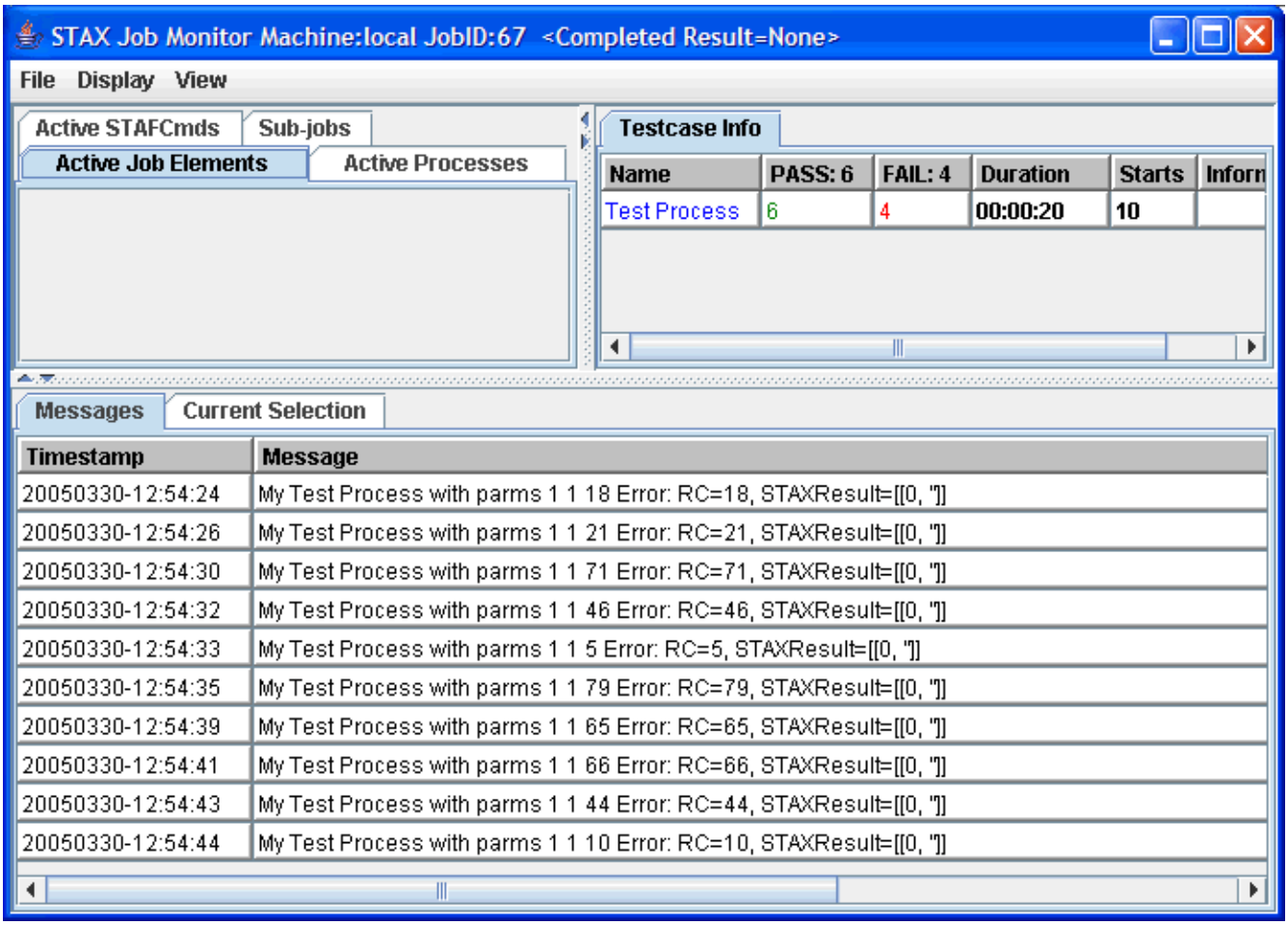
Click on the "Display" in the menu bar, and then select "Display Job Log". You should see something similar to:

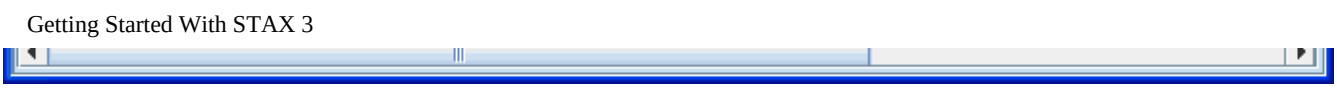
Figure 62.



Note that the testcase information is stored in the Job Log (so that it will still be available after the STAX job completes and the STAX Monitor is closed). Close the Job Log, and click on "File" in the menu bar, and then select "Exit and Resubmit Job". This will close the current STAX Monitor window, and submit a new job with the same options. Run this job a few times and note the testcase passes/fails based on the random return codes:

Figure 63.





7.15. Adding time constraints into a STAX job

Now let's look at a STAX job that demonstrates how you can add time constraints to your STAX jobs:

```
1:   <?xml version="1.0" encoding="UTF-8" standalone="no"?>
2:   <!DOCTYPE stax SYSTEM "stax.dtd">
3:
4:   <stax>
5:
6:     <defaultcall function="begin_tests"/>
7:
8:     <script>
9:       ImportMachine = 'local'
10:      ImportDirectory = '{STAF/Config/STAFRoot}/services/stax'
11:      test_process_times = ['30', '10', '25']
12:    </script>
13:
14:    <function name="begin_tests">
15:
16:      <sequence>
17:
18:        <import machine="ImportMachine"
19:          file="'%s/FunctionParametersLogging.xml' % ImportDirectory"/>
20:
21:        <iterate var="parm1" in="test_process_times" indexvar="index">
22:
23:          <sequence>
24:
25:            <timer duration="'20s'">
26:
27:              <call function="'main'">{ 'parms' : '%s 1 0' % parm1 }</call>
28:
29:            </timer>
30:
31:            <if expr="RC == 1">
32:              <message log="1">
33:                'Test # %s still running after timer the expired' % index
34:              </message>
35:              <elseif expr="RC == 0">
36:                <message log="1">
37:                  'Test # %s ended before the timer expired' % index
38:                </message>
39:              </elseif>
40:            </if>
41:
42:          </sequence>
43:
44:        </iterate>
45:
46:      </sequence>
47:
48:    </function>
49:
50:  </stax>
```

Notice on line 25 that we have a **timer** element, and we have specified the "duration" attribute as '20s' to indicate that we want the contents of the **timer** to be executed up to 20 seconds. If the contents of the **timer** were still executing after 20 seconds, then they will be terminated and the RC variable will be set to 1. If the contents finished before 20 seconds, then the RC variable will be set to 0. This gives you control over how to gauge

whether the contents of the **timer** were successful or not.

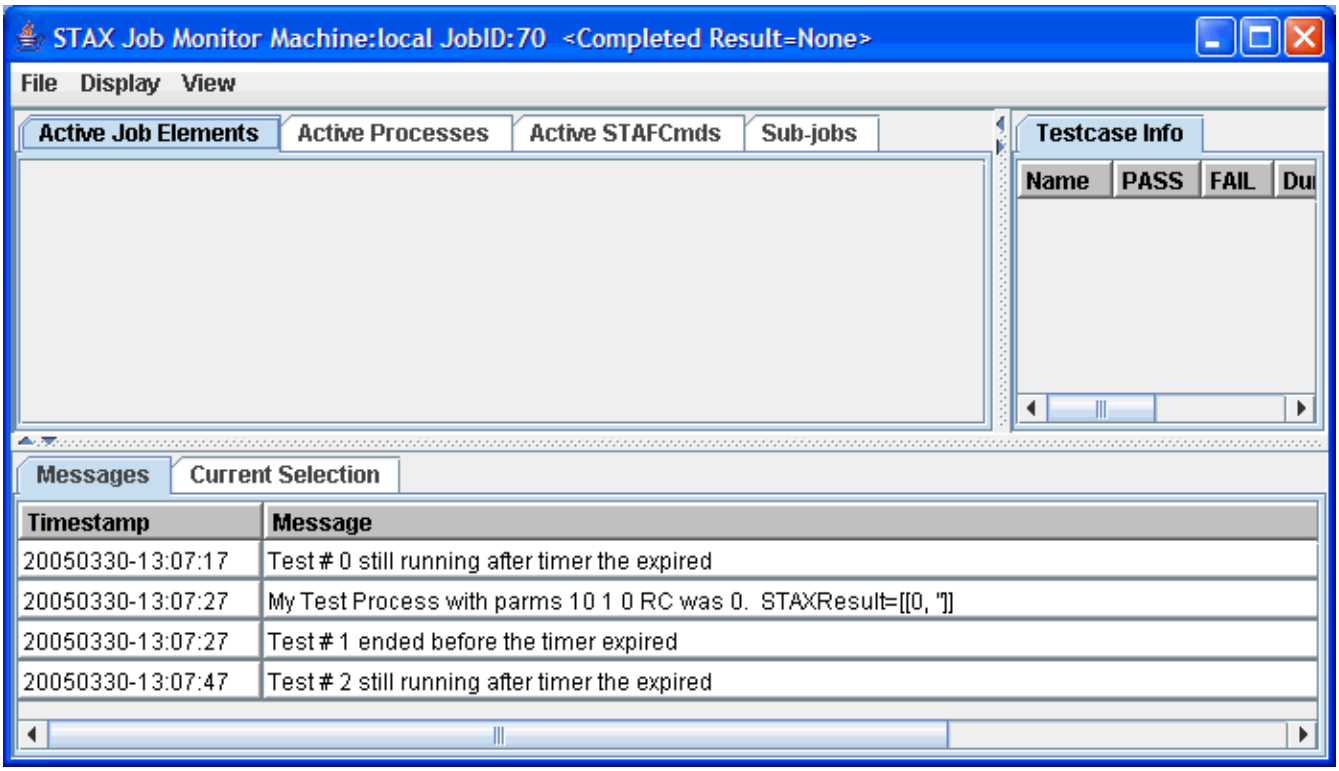
Also notice that on line 21 we have an **iterate** element. It is used to execute its contents while iterating over a list of data. On line 8 we have a **script** element that defines the variable test_process_times, which is set to a list of 3 strings ('30', '10', and '25'). These values represent the first parameter that will be passed to the TestProcess (the number of loops it should execute). In the **iterate** element on line 21, we are specifying the test_process_times variable as the value for the "in" attribute. This means that we will execute the contents of the **iterate** 3 times (because the test_process_times list contains 3 items). We have also specified the "indexvar" attribute to be "index", so that we can refer to the current index number (0-based). We have also specified the "var" attribute as "parm1". This allows us to refer to the current item in the list over which we are iterating. On line 27, when we call the function, we pass it parm1 as the first parameter.

So, first we will call the function with parameters "30 1 0". Since the **timer** duration is 20 seconds, the timer will expire and the process will be termianted, and RC will be set to 1. Next, we will then call the function with parameters "10 1 0", and when it is complete (even though the timer's duration of 20 seconds has not been reached, the **timer** will complete and RC will be set to 0. Lastly, we will call the function with parameters "25 1 0", and since the **timer** duration is 20 seconds the timer will expire and the process will be termianted, and RC will be set to 1.

Open your favorite text editor, or use an XML editor such as Cooktop, copy/paste lines 1-50 (delete the first five characters on each line, to get rid of the line numbers and whitespace), and save the file as Timer.xml. Go to the main STAX Monitor panel, and click on the "Submit New Job..." button. For "XML Job File" "Filename:", enter the full path to the Timer.xml file.

Next, click on the "Test" button to check the STAX job for XML and Python errors. You should see a "Validation Successful" popup. Next, click on the "Submit New Job" button. You should see a new STAX Monitor window for this STAX job. After the STAX job completes, your STAX Monitor window should look like the following:

Figure 64.



7.16. Sending email in a STAX job

Now let's look at how you can send email in STAX jobs:

```
1:   <?xml version="1.0" encoding="UTF-8" standalone="no"?>
2:   <!DOCTYPE stax SYSTEM "stax.dtd">
3:
4:   <stax>
5:
6:     <script>
7:       from com.ibm.staf import STAFUtil
8:       emailTo = 'user@company.com'
9:       emailSubject = 'This is a test of STAX and the email service'
10:      emailMessage = ('Hello\n\nSTAX Job ID %s Email test ' % STAXJobID +
11:                      'successful!\n\nCheers!')
12:    </script>
13:
14:    <defaultcall function="email_example"/>
15:
16:    <function name="email_example">
17:
18:      <sequence>
19:
20:        <stafcmd name="'Sending email'">
21:          <location>'local'</location>
22:          <service>'email'</service>
23:          <request>
24:            'send to %s subject %s message %s' % (emailTo, \
25:              STAFUtil.wrapData(emailSubject), STAFUtil.wrapData(emailMessage))
26:          </request>
27:        </stafcmd>
28:
29:        <message log="1">'Email RC=%s, Result=%s' % (RC, STAFResult)</message>
30:
31:      </sequence>
32:
33:    </function>
34:
35:  </stax>
```

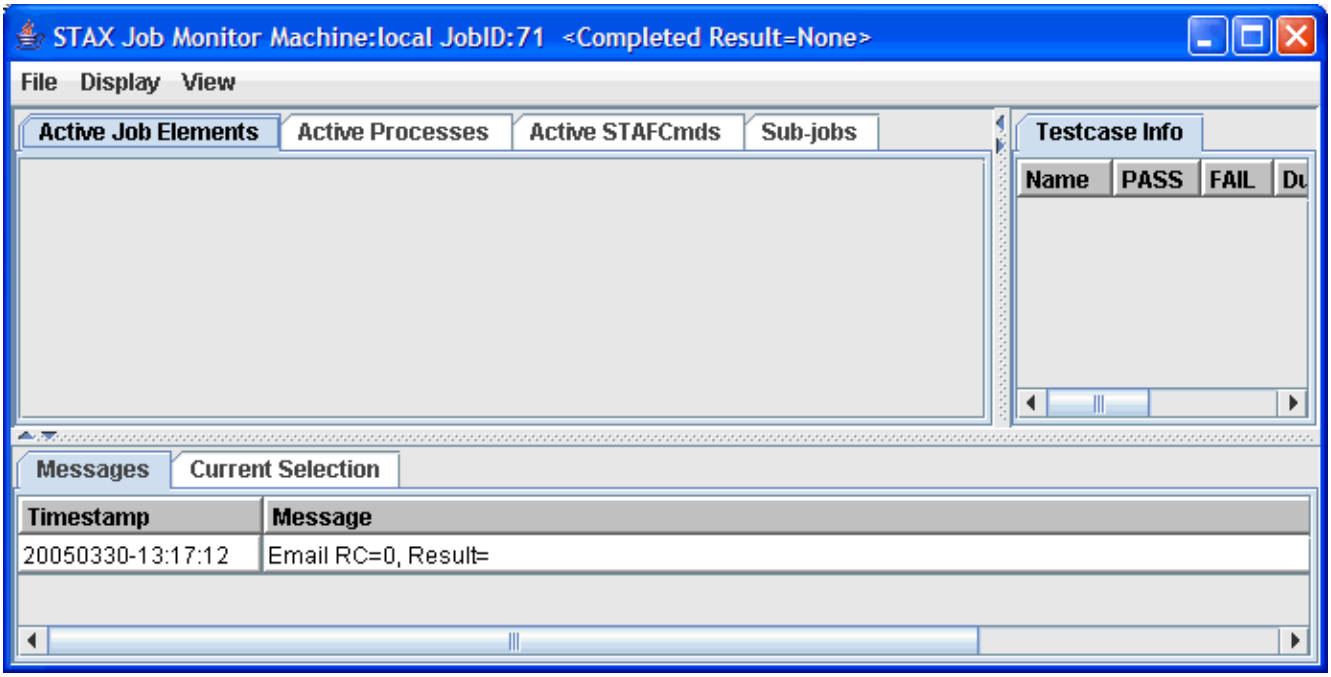
In this example, on line 20 we have a **stafcmd** that calls the Email service with a SEND request. On line 6 we have a **script** that defines some variables for the email. In this example, we are actually going to call some Java classes. On line 7, we are importing the STAFUtil class (so that we can later call one of its methods). Line 10 demonstrates one way to continue a Python statement on multiple lines. You can enclose the statement with paranthesis.

Line 24 demonstrates another way to continue a Python statement on multiple lines. You can use place the "\" character at the end of line you wish to continue on the next line. On line 25 we are calling the STAFUtil.wrapData method twice, and passing the emailSubject and emailMessage variables. This method converts the string into a known format for STAF, so that you don't need to worry about embedded spaces or special characters. It is quite useful when sending requests to STAF services.

Open your favorite text editor, or use an XML editor such as Cooktop, copy/paste lines 1-35 (delete the first five characters on each line, to get rid of the line numbers and whitespace), change line 8 to be your email address instead of "user@company.com", and save the file as Email.xml. Go to the main STAX Monitor panel, and click on the "Submit New Job..." button. For "XML Job File" "Filename:", enter the full path to the Email.xml file.

Next, click on the "Test" button to check the STAX job for XML and Python errors. You should see a "Validation Successful" popup. Next, click on the "Submit New Job" button. You should see a new STAX Monitor window for this STAX job. After the STAX job completes, your STAX Monitor window should look like the following:

Figure 65.



Now check your email inbox and you should see an email similar to:

```
Subject: This is a test of STAX and the email service

*****************************************************************
* DO NOT RESPOND TO THE SERVICE MACHINE THAT GENERATED THIS NOTE *
*****************************************************************

Hello

STAX Job ID 71 Email test successful!

Cheers!
```

7.17. Starting STAX jobs via EventManager

Now let's look at how you automatically start STAX jobs via the EventManager service. Here is the command line request to execute the email STAX job from the previous section:

```
staf local stax execute file c:/STAF/services/stax/Email.xml
```

Execute this from the command line, and verify that you receive the email.

Now let's register the same STAX request with the EventManager service, and then generate the event:

```
staf local eventmanager register machine local service stax request "execute file
c:/STAF/services/stax/Email.xml" type abc subtype xyz
staf local event generate type abc subtype xyz
```

The first command will return the EventManager ID associated with that request (most likely 1 since this is the first EventManager registration you have submitted). The second command will generate an event with type "abc" and subtype "xyz", and will trigger the EventManager service to execute the STAX job. Verify that you receive the email. This is how you can automatically start your STAX jobs based on events (such as a product build event).

Also note that the EventManager registrations are persistent (they will still be registered even after your restart STAF or reboot the machine). So, let's run the following command to unregister this request:

```
staf local eventmanager unregister id 1
```

7.18. Starting STAX jobs via Cron

You can use the Cron service start your STAX jobs at certain times (12:00AM, for example). In this sample we will execute the email STAX job every minute.

Run the following command to register with the Cron service:

```
staf local cron register minute ANY machine local service stax request "execute file
c:/STAF/services/stax/Email.xml"
```

Since we have specified "minute ANY", the Cron service will execute the STAX job every minute. In addition to MINUTE, the other options you can specify are HOUR, DAY, MONTH, and WEEKDAY.

Verify that you are now receiving the email every minute.

Also note that the Cron registrations are persistent (they will still be registered even after your restart STAF or reboot the machine). So, let's run the following command to unregister this request:

```
staf local cron unregister id 1
```

8. Appendix: Sample STAX jobs

8.1. DoesNothing.xml
8.2. RunNotepadProcess.xml
8.3. RunDelayRequest.xml
8.4. CheckSTAFCmdRC.xml
8.5. RunTestProcess.xml
8.6. UsingScripts.xml
8.7. FunctionParameters.xml
8.8. FunctionParametersLogging.xml
8.9. ImportFunction.xml
8.10. Block.xml
8.11. Parallel.xml
8.12. Loop.xml
8.13. Testcase.xml
8.14. Timer.xml
8.15. Email.xml

8.1. DoesNothing.xml

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<!DOCTYPE stax SYSTEM "stax.dtd">

<stax>

  <defaultcall function="main"/>

  <function name="main">
    <nop/>
  </function>

</stax>
```

8.2. RunNotepadProcess.xml

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<!DOCTYPE stax SYSTEM "stax.dtd">

<stax>

  <defaultcall function="main"/>

  <function name="main">

    <process>
      <location>'local'</location>
      <command>'notepad'</command>
    </process>

  </function>

</stax>
```

8.3. RunDelayRequest.xml

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<!DOCTYPE stax SYSTEM "stax.dtd">

<stax>

  <defaultcall function="main"/>

  <function name="main">

    <stafcmd>
      <location>'local'</location>
      <service>'delay'</service>
      <request>'delay 30000'</request>
    </stafcmd>

  </function>

</stax>
```

8.4. CheckSTAFCmdRC.xml

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<!DOCTYPE stax SYSTEM "stax.dtd">

<stax>

  <defaultcall function="main"/>

  <function name="main">

    <sequence>

      <stafcmd>
        <location>'local'</location>
        <service>'var'</service>
        <request>'resolve string {STAF/Config/OS/Name}'</request>
      </stafcmd>

      <if expr="RC != 0">
```

```
        <message>'Oops, RC = %s, Result = %s' % (RC, STAFResult)</message>
        <else>
          <message>'Great!  STAF/Config/OS/Name = %s' % (STAFResult)</message>
        </else>
      </if>

    </sequence>

  </function>

</stax>
```

8.5. RunTestProcess.xml

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<!DOCTYPE stax SYSTEM "stax.dtd">

<stax>

  <defaultcall function="main"/>

  <function name="main">

    <sequence>

      <process name="'My Test Process'">
        <location>'local'</location>
        <command>
          'java com.ibm.staf.service.stax.TestProcess 10 3 99'
        </command>
        <env>
          'CLASSPATH=C:/STAF/bin/JSTAF.jar;C:/STAF/services/stax/STAXMon.jar'
        </env>
        <stderr mode="'stdout'"/>
        <returnstdout/>
      </process>

      <if expr="RC != 0">
        <message>'Error: RC=%s, STAXResult=%s' % (RC, STAXResult)</message>
        <else>
          <message>'Process RC was 0.  STAXResult=%s' % STAXResult</message>
        </else>
      </if>

    </sequence>

  </function>

</stax>
```

8.6. UsingScripts.xml

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<!DOCTYPE stax SYSTEM "stax.dtd">

<stax>

  <script>
    jstafJar = '{STAF/Config/STAFRoot}/bin/JSTAF.jar'
    staxmonJar = '{STAF/Config/STAFRoot}/services/stax/STAXMon.jar'
    machine = 'local'
    java_command = 'java'
    java_class = 'com.ibm.staf.service.stax.TestProcess'
    loopCount = 10
    incSeconds = 3
    returnCode = 50
    parms = '%s %s %s' % (loopCount, incSeconds, returnCode)
    cp = 'CLASSPATH=%s;%s' % (jstafJar, staxmonJar)
  </script>

  <defaultcall function="main"/>

  <function name="main">

    <sequence>

      <process name="'My Test Process'">
        <location>machine</location>
        <command>'%s %s %s' % (java_command, java_class, parms)</command>
        <env>cp</env>
        <stderr mode="'stdout'"/>
        <returnstdout/>
      </process>

      <if expr="RC != 0">
        <message>'Error: RC=%s, STAXResult=%s' % (RC, STAXResult)</message>
        <else>
          <message>'Process RC was 0.  STAXResult=%s' % STAXResult</message>
        </else>
      </if>

    </sequence>

  </function>

</stax>
```

8.7. FunctionParameters.xml

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<!DOCTYPE stax SYSTEM "stax.dtd">

<stax>

  <defaultcall function="main"/>

  <function name="main">

    <function-prolog>
      This function is used as an example in the "Getting Started with STAX"
      document.  It starts the TestProcess, and allows the parms, machine,
      java_command, java_class, processName, and classpath to be passed as
      arguments to the function.
    </function-prolog>
```

```
    <function-map-args>
      <function-required-arg name="parms">
        The three parameters to pass to the process.
      </function-required-arg>
      <function-optional-arg name="machine" default="'local'">
        The name of machine where the test process should run.
      </function-optional-arg>
      <function-optional-arg name="java_command" default="'java'">
        The name of java executable that should be used to execute the test
        process.
      </function-optional-arg>
      <function-optional-arg name="java_class"
          default="'com.ibm.staf.service.stax.TestProcess'">
        The name of java class for the test process.
      </function-optional-arg>
      <function-optional-arg name="processName" default="'My Test Process'">
        The name of the process.
      </function-optional-arg>
      <function-optional-arg name="classpath"

default="'{STAF/Config/STAFRoot}/bin/JSTAF.jar;{STAF/Config/STAFRoot}/services/stax/STAXMon.jar'">
        The CLASSPATH that should be used when the test process is started..
      </function-optional-arg>
    </function-map-args>

    <sequence>

      <process name="processName">
        <location>machine</location>
        <command>'%s %s %s' % (java_command, java_class, parms)</command>
        <env>'CLASSPATH=%s' % classpath</env>
        <stderr mode="'stdout'"/>
        <returnstdout/>
      </process>

      <if expr="RC != 0">
        <message>'Error: RC=%s, STAXResult=%s' % (RC, STAXResult)</message>
        <else>
          <message>'Process RC was 0.  STAXResult=%s' % STAXResult</message>
        </else>
      </if>

    </sequence>

  </function>

</stax>
```

8.8. FunctionParametersLogging.xml

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<!DOCTYPE stax SYSTEM "stax.dtd">

<stax>

  <defaultcall function="main"/>

  <function name="main">

    <function-prolog>
      This function is used as an example in the "Getting Started with STAX"
      document.  It starts the TestProcess, and allows the parms, machine,
      java_command, java_class, processName, and classpath to be passed as
      arguments to the function.
    </function-prolog>

    <function-map-args>
      <function-required-arg name="parms">
        The three parameters to pass to the process.
      </function-required-arg>
      <function-optional-arg name="machine" default="'local'">
        The name of machine where the test process should run.
      </function-optional-arg>
      <function-optional-arg name="java_command" default="'java'">
        The name of java executable that should be used to execute the test
        process.
      </function-optional-arg>
      <function-optional-arg name="java_class"
          default="'com.ibm.staf.service.stax.TestProcess'">
        The name of java class for the test process.
      </function-optional-arg>
      <function-optional-arg name="processName" default="'My Test Process'">
        The name of the process.
      </function-optional-arg>
      <function-optional-arg name="classpath"

default="'{STAF/Config/STAFRoot}/bin/JSTAF.jar;{STAF/Config/STAFRoot}/services/stax/STAXMon.jar'">
        The CLASSPATH that should be used when the test process is started..
      </function-optional-arg>
    </function-map-args>

    <sequence>

      <process name="'%s with parms %s' % (processName, parms)">
        <location>machine</location>
        <command>'%s %s %s' % (java_command, java_class, parms)</command>
        <env>'CLASSPATH=%s' % classpath</env>
        <stderr mode="'stdout'"/>
        <returnstdout/>
      </process>

      <if expr="RC != 0">
        <message log="1" level="'Error'">
          '%s with parms %s Error: RC=%s, STAXResult=%s' % \
            (processName, parms, RC, STAXResult)
        </message>
        <else>
          <message log="1">
            'SUCCESS %s with parms %s\nSTAXResult=%s' % \
              (processName, parms, STAXResult)
          </message>
        </else>
```

```
      </if>

      <return>RC</return>

    </sequence>

  </function>

</stax>
```

8.9. ImportFunction.xml

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<!DOCTYPE stax SYSTEM "stax.dtd">

<stax>

  <defaultcall function="begin_tests"/>

  <script>
    ImportMachine = 'local'
    ImportDirectory = '{STAF/Config/STAFRoot}/services/stax'
  </script>

  <function name="begin_tests">

    <sequence>

      <import machine="ImportMachine"
        file="'%s/FunctionParametersLogging.xml' % ImportDirectory"/>

      <call function="'main'">{ 'parms' : '9 2 7' }</call>

      <call function="'main'">{ 'parms' : '2 9 15' }</call>

    </sequence>

  </function>

</stax>
```

8.10. Block.xml

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<!DOCTYPE stax SYSTEM "stax.dtd">

<stax>

  <defaultcall function="begin_tests"/>

  <script>
    ImportMachine = 'local'
    ImportDirectory = '{STAF/Config/STAFRoot}/services/stax'
  </script>

  <function name="begin_tests">

    <block name="'SVT_Regression'">

      <sequence>

        <import machine="ImportMachine"
          file="'%s/FunctionParametersLogging.xml' % ImportDirectory"/>
```

```
        <call function="'main'">{ 'parms' : '30 1 0' }</call>

        <call function="'main'">{ 'parms' : '15 2 0' }</call>

      </sequence>

    </block>

  </function>

</stax>
```

8.11. Parallel.xml

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<!DOCTYPE stax SYSTEM "stax.dtd">

<stax>

  <defaultcall function="begin_tests"/>

  <script>
    ImportMachine = 'local'
    ImportDirectory = '{STAF/Config/STAFRoot}/services/stax'
  </script>

  <function name="begin_tests">

    <sequence>

      <import machine="ImportMachine"
        file="'%s/FunctionParametersLogging.xml' % ImportDirectory"/>

      <block name="'Run Processes in Parallel'">

        <parallel>

          <call function="'main'">{ 'parms' : '40 1 0' }</call>
          <call function="'main'">{ 'parms' : '15 2 0' }</call>
          <call function="'main'">{ 'parms' : '10 2 0' }</call>

        </parallel>

      </block>

      <call function="'main'">{ 'parms' : '5 3 0' }</call>

    </sequence>

  </function>

</stax>
```

8.12. Loop.xml

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<!DOCTYPE stax SYSTEM "stax.dtd">

<stax>

  <defaultcall function="begin_tests"/>

  <script>
    ImportMachine = 'local'
    ImportDirectory = '{STAF/Config/STAFRoot}/services/stax'
  </script>

  <function name="begin_tests">

    <sequence>

      <import machine="ImportMachine"
        file="'%s/FunctionParametersLogging.xml' % ImportDirectory"/>

      <loop from="1" to="3" var="index">

        <block name="'Block #%s' % index">

          <call function="'main'">{ 'parms' : '10 %s 0' % index }</call>

        </block>

      </loop>

    </sequence>

  </function>

</stax>
```

8.13. Testcase.xml

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<!DOCTYPE stax SYSTEM "stax.dtd">

<stax>

  <defaultcall function="begin_tests"/>

  <script>
    ImportMachine = 'local'
    ImportDirectory = '{STAF/Config/STAFRoot}/services/stax'
    from random import randint
  </script>

  <function name="begin_tests">

    <sequence>

      <import machine="ImportMachine"
        file="'%s/FunctionParametersLogging.xml' % ImportDirectory"/>

      <loop from="1" to="10">

        <testcase name="'Test Process'">

          <sequence>

            <script>r = randint(1, 100)</script>
```

```
          <call function="'main'">{ 'parms' : '1 1 %s' % r }</call>

          <if expr="STAXResult &lt;= 50">
            <tcstatus result="'pass'"/>
            <else>
              <tcstatus result="'fail'"/>
            </else>
          </if>

        </sequence>

      </testcase>

    </loop>

  </sequence>

</function>

</stax>
```

8.14. Timer.xml

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<!DOCTYPE stax SYSTEM "stax.dtd">

<stax>

  <defaultcall function="begin_tests"/>

  <script>
    ImportMachine = 'local'
    ImportDirectory = '{STAF/Config/STAFRoot}/services/stax'
    test_process_times = ['30', '10', '25']
  </script>

  <function name="begin_tests">

    <sequence>

      <import machine="ImportMachine"
        file="'%s/FunctionParametersLogging.xml' % ImportDirectory"/>

      <iterate var="parm1" in="test_process_times" indexvar="index">

        <sequence>

          <timer duration="'20s'">

            <call function="'main'">{ 'parms' : '%s 1 0' % parm1 }</call>

          </timer>

          <if expr="RC == 1">
            <message log="1">
              'Test # %s still running after timer the expired' % index
            </message>
            <elseif expr="RC == 0">
              <message log="1">
                'Test # %s ended before the timer expired' % index
              </message>
            </elseif>
```

```
        </if>

      </sequence>

    </iterate>

  </sequence>

</function>

</stax>
```

8.15. Email.xml

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<!DOCTYPE stax SYSTEM "stax.dtd">

<stax>

  <script>
    from com.ibm.staf import STAFUtil
    emailTo = 'user@company.com'
    emailSubject = 'This is a test of STAX and the email service'
    emailMessage = ('Hello\n\nSTAX Job ID %s Email test ' % STAXJobID +
                    'successful!\n\nCheers!')
  </script>

  <defaultcall function="email_example"/>

  <function name="email_example">

    <sequence>

      <stafcmd name="'Sending email'">
        <location>'local'</location>
        <service>'email'</service>
        <request>
          'send to %s subject %s message %s' % (emailTo, \
           STAFUtil.wrapData(emailSubject), STAFUtil.wrapData(emailMessage))
        </request>
      </stafcmd>

      <message log="1">'Email RC=%s, Result=%s' % (RC, STAFResult)</message>

    </sequence>

  </function>

</stax>
```

9. End of Document