STAX Service User's Guide

STAf eXecution engine
Version 1.5.10

April 17, 2005

Document Owners: Sharon Lucas, David Bender, Charles Rankin
eMail Addresses: lucass@us.ibm.com, bdavid@us.ibm.com, rankinc@us.ibm.com

---

Table of Contents

Overview

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

STAX accepts job definitions, in the form of XML documents. Fundamentally, these job definitions allow you to specify the processes and STAF commands necessary to perform the job. STAX provides a wealth of expressive functionality on top of this, making it easy to implement, manage, track, and monitor your jobs.

STAX uses the Python scripting language for variable and expression evaluation. The Python code is executed by Jython, a version of Python written entirely in Java. This allows STAX to take advantage of the powerful and easy-to-use features of Python.

The sections that follow describe the basic concepts behind STAX, explain the STAX XML language used to define your jobs, and detail the commands externalized by the STAX service. Read on to find out more about the exciting new world of STAX.

Concepts

STAX Elements

A STAX Element is a node in a STAX XML document. 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, exceptions and signals, and wrappers such as functions and blocks that encompass other STAX Elements.

Processes and Commands

A STAX job definition describes the execution flow for processes and STAF commands. A 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. For example, the command could specify to run a Java application or a Rexx file which is one of your testcases. A stafcmd element defines execution information for other STAF commands. A stafcmd element specifies the STAF service and request to be executed (e.g. RESPOOL REQUEST POOL, FS COPY FILE) and the machine where it should be run. Processes and stafcmds may be put into sequential and/or parallel wrappers which can be nested.

Expression Evaluation via Python

STAX allows you to avoid hard-coding information in your job definition by using Python to assign values to variables and then using Python scripting language to evaluate expressions and to execute Python code. STAX also sets some variables in Python that provide runtime information about the job definition's execution.

For example, instead of hardcoding the name of the machines where processes and commands are executed during the job, a Python variable can be assigned the name of the machine and specified for the location element. Python variables also be provided at the time the job is requested to be executed. A Python variable may also be assigned a list of machine names.

After STAX processes some elements (e.g. process and stafcmd), the return code and result (if applicable) are accessible via STAX variables. These variables can be referenced by other STAX elements (e.g. via the if element's expression attribute) to determine logic flow within the job.

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. Additionally, there are Iterate Elements which allow a STAX Element to be executed repeatedly while stepping through a list of data for each iteration (this could be used, for example, to execute a sequence of commands for each machine in a list).

STAX-Threads

When the STAX Service executes elements in parallel, rather than using real system threads (and thereby potentially creating an overabundance of system threads), the STAX Service will simulate the threading capabilities via a thread pool which will utilize a manageable number of real system threads. These simulated threads are called STAX-Threads.

Whenever a new STAX-Thread is created, existing variables are cloned from the parent STAX-Thread. To create a global variable that can be accessed across STAX-Threads, use the STAXGlobal class described in "Function" section. STAX elements that can create STAX-Threads include the following: <parallel>, <paralleliterate>, <process-action>, and <function> elements with a local scope.

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 (which provide a unique name that can be called from other STAX Elements in the Job Definition.

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 us to group and parametize chunks of XML to be used arbitrarily many times later. Functions also provide a tool for splitting jobs into pieces that have a well-defined role. STAX allows functions to be imported from other XML files so that you can build up libraries of STAX functions that can be reused by many different STAX jobs.

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 (for example, a return code) to determine logic flow within the STAX Job Definition, thus allowing job flow to branch dynamically.

Exceptions and Signals

The STAX Service provides exception and signal handling capabilities. STAX exception handlers alter the execution flow of the job. STAX signal handlers provide asynchronous error handling of raised signals. The STAX execution engine may also raise signals or throw exceptions for errors which occur during job execution.

Monitoring STAX Jobs

A STAX Monitor application is available for the STAX Service. This application displays a graphical representation of the currently running elements of a given Job. The STAX Monitor makes it easy to see which processes and STAF commands are currently running as well as the blocks which contain them. You may select a process or STAF command to get more detailed information about it. You may also select a block and then control the execution of the job by choosing to hold, release, or terminate the block. The STAX Monitor also displays a list of testcases that have been run and the number of passes and fails for each. Also, a messages panel displays any messages that are sent by the job. This can help make debugging a job definition easy.

Logging

The STAX Service maintains a service log which records high level information about all jobs that have been submitted.

The STAX Service also maintains an individual job log for each submitted job. These job-specific logs record information such as testcase status and job execution tracing. If the "log" element is used within the STAX Job Definition, then a user log is also created for the submitted job.

Using XML to Define STAX Jobs

STAX uses XML (Extensible Markup Language) to describe STAX job definitions. XML is a language defined by the World Wide Web Consortium (W3C), the body that sets the standards for the Web. It is called extensible because it is not a fixed format like HTML (a single, predefined markup language). Instead, XML is actually a 'metalanguage' -- a language for describing other languages -- which lets you design your own customized markup languages for limitless different types of documents. This section reviews some XML fundamentals. Refer to the "References" section for where to get more information about XML.

Both markup and text in an XML document are case-sensitive. All XML processing instructions start with <? and end with ?>.

XML comments start with <!-- and end with -->. In XML, tags always start with < and end with >. The names that can be used for a tag are defined by the DTD (Document Type Definition).

XML documents are made up of XML *elements*. Much like in HTML, 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>. It's necessary to enclose the entire document, except for processing instructions, in one element, called the *root element* -- that's the <stax> element here:

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

Empty Elements

Empty elements have only one tag, not a start and end tag. In XML, you close an empty element with />. For example, nop is an empty element:

```
<nop/>
```

Attributes

Attributes in XML are name-value pairs that let you specify additional data in start and empty tags. To assign a value to an attribute, you use an equal sign. Because markup is always text, attributes are also text. Even if you're assigning a number to an attribute, you treat that number as a text string and enclose it in quotes. In XML, you must enclose attribute values in quotation marks. Usually, you use double quotes, but if the attribute value itself contains double quotes, you can use single quotes to surround the text.

If the attribute value contains both single and double quotes, you can use the XML-defined entity &apos; for a single quote and &quot; for double quotes.

An example of a defaultcall element with a function attribute is:

```
<defaultcall function="MainFunction"/>
```

Creating a STAX XML document

The root element can contain other elements, of course. Here, I added elements for three functions to the document and added one element that defines the function to call first by default. Note that the function element has an attribute called name and the defaultcall element is empty and has an attribute called function.

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

  <function name="FunctionA">
      ...
  </function>
  <function name="FunctionB">
      ...
  </function>

  <function name="FunctionC">
      ...
  </function>
</stax>
```

The function element can contain a single task element as defined by the STAX DTD. Here, I added a process element to FunctionA, a stafcmd element to FunctionB, and a log element to functionC. A process element can contain other elements. In this case I added location, command, and parms. A stafcmd element can contain other elements. In this case I added location, service, and request.

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

  <function name="FunctionA">
      <process>
        <location>'local'</location>
        <command>'java'</command>
        <parms>'com.ibm.staf.service.stax.TestProcess 2 4 0'
      </process>
  </function>

  <function name="FunctionB">
      <stafcmd>
        <location>'local'</location>
        <service>'misc'</service>
        <request>'version'</request>
      </stafcmd>
  </function>

  <function name="FunctionC">
      <log>'This function logs this message'</log>
  </function>
</stax>
```

Document Type Definition (DTD)

The STAX DTD specifies the correct syntax of the document. A STAX XML document is valid if it complies with the STAX DTD. A DTD is a formal description in XML Declaration Syntax of a particular type of document. It defines what names are to be used for the different types of elements, where they may occur, and how they all fit together. Refer to the "STAX Document Type Definition (DTD)" section to see the entire STAX DTD.

For example, the STAX DTD allows you to describe a STAF Command which has a name attribute and contains location, service, and request elements. The relevant part of the STAX DTD contains:

```
            <!ELEMENT stafcmd   (location, service, request)>
            <!ATTLIST stafcmd
                      name      CDATA    #IMPLIED
```

```
>
<!ELEMENT location  (#PCDATA)>
<!ELEMENT service   (#PCDATA)>
<!ELEMENT request   (#PCDATA)>
```

This defines a stafcmd as an element type containing location, service, and request elements; and it defines location, service, and request as element types containing just plain text (Parsed Character Data or PCDATA) and defines name as an attribute type containing just plain text (Character Data or CDATA). Validating parsers read the DTD before they read your document so that they can identify where every element type ought to come and how each relates to the other, so that applications which need to know this in advance (such as the STAX service) can set themselves up correctly. The example above lets you create STAF commands like:

```
<stafcmd name="'Delay'">
  <location>'local'</location>
  <service>'delay'</service>
  <request>'delay 5000'</request>
</stafcmd>
```

A DTD provides applications with advance notice of what names and structures can be used in a particular document type. Using a DTD when editing files means you can be certain that all documents which belong to a particular type will be constructed and named in a consistent and conformant manner. The STAX service parses an XML document to break it down into its component parts and then handles the resulting data. STAX uses the XML Parser for Java which is a validating XML parser.

Refer to the "References" section for where to get more information about the XML Parser for Java.

Using Python for Expression Evaluation

STAX uses the 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 Java Virtual Machine (JVM). Using Jython, you can write Python code that interacts with any Java code.

STAX variable names must follow the Python variable naming conventions. In Python, variable names come into existence when you assign values to them, but there are a few rules to follow when picking names for variables.

- *Syntax: (underscore or letter) + (any number of letters, digits or underscores)*
  Variable names must start with an underscore or letter, and be followed by any number of letters, digits, or underscores. _machine, machine, and Mach_1 are legal names, but 1_Mach, mach$, and @#! are not.

- *Case matters: MACHNAME is not the same as machname*
  Python always pays attention to case, both in names you create and in reserved words. For instance, X and x refer to two different variable names.

- *Python reserved words are off limits*
  You cannot define variable names to be the same as words that mean special things in the Python language. Following are reserved words in Python: and, assert, break, class, continue, def, del, elif, else, except, exec, finally, for, from global, if, import, in, is, lambda, not, or, pass, print, raise, return, try, while.

  Note: Python lets you use the names of Python built-in functions as variable names. However, we recommend that you don't use the name of a Python built-in function as a variable name because you may want to use the Python built-in function at some point in your STAX job. Following are names of Python built-in functions: abs, basestring, bool, callable, chr, classmethod, cmp, compile, complex, delattr, dict, dir, divmod, enumerate, eval, execfile, file, filter, float, getattr, globals, hasattr, hash, help, hex, id, input, int, isinstance, issubclass, iter, len, locals, long, map, max, min, object, oct, open, ord, pow, property, range, raw_input, reduce, reload, repr, round, setattr, slice, staticmethod, str, sum, super, tuple, type, unichr, unicode, vars, xrange, zip.

- *STAX reserved words are off limits*
  You should not define variable names to be the same as words that mean special things to the STAX service. Following are reserved words in STAX:
  - RC
  - STAFResult
  - any word beginning with STAX (e.g. STAXJobName, STAXThreadID)
  - any word beginning with STAF (e.g. STAFResult, STAFRC)

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 XML document. They assign a string constant (literal) "CoolTest" 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, a STAXPythonEvaluationError signal is raised.

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

For elements and attributes whose values are evaluated via Python, we need to distinguish between literals and variables.

For example, the following request element's value contains a string constant which is concatenated with the value of a variable named machName. So, if the value of variable machName is 'testA.austin.ibm.com', after being evaluated by Python, the request element's value would be: 'RELEASE POOL ClientMachPool ENTRY testA.austin.ibm.com'.

```
<request>'RELEASE POOL ClientMachPool ENTRY ' + machName</request>
```

Another way to do this is:

```
<request>'RELEASE POOL ClientMachPool ENTRY %s' % (machName)</request>
```

where the %s indicates a String format (and can also be used for decimal format, etc.), and where the value of the machName variable would replace the %s marker.

Also, note that the following two lines do exactly the same thing in a STAX XML document. They assign a string constant (literal) "VerifyRC" to the function attribute's value.

```
<call function="'VerifyRC'"/>
<call function='"VerifyRC"'/>
```

However, the following line is not the same. It assigns the value of a variable named VerifyRC to the function attribute's value. If a variable named VerifyRC does not exist, a STAXPythonEvaluationError signal is raised.

```
<call function="VerifyRC"/>
```

Also, note that XML processors assume that < always starts a tag and that & always starts an entity reference, so you should avoid using those characters for anything else. You must use the entity reference &lt; instead of < and entity reference &amp; instead of & or else you'll get an XML parsing error. This can be difficult sometimes as the < character is used as the less-than operator in Python, as in this example, where RC < 0 is being assigned to the expression attribute.

```
<if expr="RC &lt; 0">
```

Here's another example that shows a <script> element that contains Python code using the regular expression (re) module to look for pattern <**pass**> anywhere in a string.

```
<script>
  import re

  # Look for <pass> anywhere in the string.
  # Note have to use &lt; to represent a < in the pattern.
  matchstr = r'.*?&lt;pass>.*?'

  matchFlag = re.match(matchstr, STAFResult)
</script>
```

Refer to the "References" section for where to get more information about Jython and Python.

If you are already a CPython programmer, or are hoping to use CPython code under Jython, refer to the "Jython and CPython Differences" section for information about differences in the two implementations of Python.

Element Syntax and Usage

A job to be executed by the STAX Service is described by an XML document. The XML document must comply with the STAX document type definition (DTD) shown in "STAX Document Type Definition (DTD)". The function of the STAX XML document is to describe STAF command and process execution.

The first line in an XML document should start with an XML declaration. This indicates the document is written in XML and specifies the XML version, the language encoding for the document, and indicates that the document refers to an external DTD (standalone="no").

The second line in an XML document should be the document type declaration. This is used to indicate the DTD used for the document. It defines the name of the root element (stax), and the DTD to be used. STAX checks the syntax of XML documents using a validating XML parser to verify that the document complies with the DTD. Note that DTDs are all about specifying the structure and syntax of XML documents (not their content).

So, the first two lines in a STAX XML document should look like:

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

This section describes the elements that can be used in a STAX XML document.

To ease the description of the elements, some elements will be grouped as follows so that they can be referenced as a group and will be shown in bold italics:

```
Reference         Elements

task              process | stafcmd | nop |
                  sequence | parallel | paralleliterate |
                  call | call-with-list | call-with-map | return | import |
                  if | loop | iterate | break | continue |
                  try | throw | rethrow |
                  signalhandler | raise |
                  hold | release | terminate |
                  testcase | tcstatus | script |
                  block | timer | log | message
```

Notes:

- Elements and attribute names in an XML document are case sensitive. That is, element <sequence> is different from <Sequence> and <SEQUENCE>.
- Only **sequence**, **parallel** and **stax** elements can contain multiple ***task*** elements.
- Names for elements (e.g. function names, block names, etc.) should not begin with STAX as these names are reserved for future extensions and enhancements.

Also, some examples of the usage of elements use "..." for brevity to represent that additional XML would be included in place of the "...".

Root: stax

An XML document must contain a root element which contains all other elements in the document. The root element of a STAX XML document is **stax**.

stax

The **stax** element consists of any number of **function**, **script**, and/or **signalhandler** elements and an optional **defaultcall** element.

Usage:

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

<stax>

  <defaultcall function="FunctionA"/>
  <script>machName = "test1.austin.ibm.com"</script>

  <function name="FunctionA">
    ...
  </function>
  <function name="FunctionB">
    ...
  </function>
  ...

</stax>
```

Python Code Execution: script

STAX uses Python, which is an object-oriented scripting language. To specify Python code to be executed, you can use the **script** element. The Python code can include any Python statements such as variable assignments, importing and running Python modules, accessing Java libraries, and running built-in Python tools.

All **script** elements contained in the root **stax** element are initialized at the beginning of the job, as each is encountered in sequential order, regardless of their placement within the **stax** element, and are accessible throughout the job (like global variables). All **script** elements contained in elements other than the **stax** element (e.g. such as the **sequence** element) are assigned as each is encountered and are accessible within their scope and are inherited from parent STAX-Threads.

Whenever a new STAX-Thread is created, existing variables are cloned from the parent STAX-Thread. To create a global variable that can be accessed across STAX-Threads, use the STAXGlobal class described in "Function" section. STAX elements that can create STAX-Threads include the following: **parallel**, **paralleliterate**, **process-action**, and **function** elements with a local scope.

script

The **script** element is used to specify Python code to be executed. For example, the **script** element can be used to define STAX variables which are utilized during the parsing of the XML document and the subsequent execution of the STAX job. Note that a STAX variable is used solely by the STAX job and is not associated with STAF variables. You can also use the **script** element to import and run Python modules and built-in Python tools.

Usage:

Goal: Create a variable named testName and assign it value "CoolTest1"

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

Goal: Create a variable named machName and assign it the value of the STAFResult variable.

```
<script>machName = STAFResult</script>
```

Goal: Create a list called machList containing 10 machine names by running the STAF Resource Pool Request command in a loop ten times, each time adding the STAF Result value from the RESPOOL REQUEST POOL command (which contains a machine name) to the list. Note that this example first creates an empty list and then adds a machine name to the list 10 times.

```
<script>machList = []</script>
<script>clientPool = 'ClientMachinePool'</script>
<loop var="i" from="1" to="10">
  <sequence>
    <stafcmd>
      <location>'server1.austin.ibm.com'</location>
      <service>'RESPOOL'</service>
      <request>'REQUEST POOL %s' % (clientPool)</request>
    </stafcmd>
    <script>machList.append(STAFResult)</script>
  </sequence>
</loop>
```

Goal: Create a list called allMachList by combining lists named unallocMachList and allocMachList. New list allMachList contains ['MachA','MachB','MachC','MachD','AllocMachA','AllocMachB'].

```
<script>unallocMachList = ['MachA','MachB','MachC','MachD']</script>
<script>allocMachList = ['AllocMachA','AllocMachB']</script>
<script>allMachList = unallocMachList + allocMachList</script>
```

Goal: Generate a random number (which could be used to randomly select which function to call) using the random module provided by Python. Note that in Python, you can use a semicolon to separate multiple statements on the same line.

```
<script>from random import random; r=random()*100</script>
```

Goal: Use a Java class, com.ibm.staf.STAFUtil (provided in JSTAF.jar), and it's wrapData() method to turn strings containing spaces into the colon-length-colon form needed for submitting a STAF Notify request. Note that the Java class STAFUtil needs to be imported in order to use its wrapData() method. This example also shows that for statements that are too long to fit on one line, Python lets you continue typing the statement on the next line, if you're coding something enclosed in (), {}, or [] pairs. Continuation lines can start at any indentation level.

```
<script>
  NotifyProfile = 'Jane Smith'
  Message = 'STAX Job ID %s failed.' % (STAXJobID)

  from com.ibm.staf import STAFUtil

  Request = ('NOTIFY PROFILE %s LEVEL NORMAL MESSAGE %s' %
            (STAFUtil.wrapData(NotifyProfile), STAFUtil.wrapData(Message)))
</script>

<message>NotifyRequest</message>
```

The message element would display something like:

```
NOTIFY PROFILE :10:Jane Smith LEVEL NORMAL MESSAGE :21:STAX Job ID 6 failed.
```

Goal: Create a Python class object, Server, and generate three instance objects from the class and create a list of these Server objects. Then iterate through the server list, logging information about each server object in the server list.

```
<script>
  # Define Server class
  class Server:
    def __init__(self, hostname, dir):
      self.hostname = hostname
      self.dir      = dir
    def __repr__(self):
      return "<Server: hostname=%s, directory=%s>" % (self.hostname, self.dir)
    def getHostname(self):
      return self.hostname
    def getDir(self):
      return self.dir

  # Create an array of 3 Server objects
  serverList = [
                 Server('myServer.austin.ibm.com', 'C:/install'),
                 Server('serverA.portland.ibm.com', 'D:/install'),
                 Server('linuxServer.austin.ibm.com', '/usr/local/install')
               ]
</script>

<iterate var="server" in="serverList" indexvar="i">
  <log>
    'Server #%s: hostname=%s, directory=%s' % (i+1, server.getHostname(),
server.getDir())
  </log>
</iterate>
```

Actions: process, stafcmd, job, nop

The process, stafcmd, job, and nop elements perform actions.

process

The **process** element represents a STAF process which will be executed on a specified machine.

After a process has completed (or if it could not be started) the following variables are set and can be referenced by the job:

- **RC -** the return code from the process. It is numeric. If it's the actual return code from the process, it is a Python Long numeric type (e.g. 0L, 25L). If an error occurred starting the process, it is an Python Integer numeric type.

- **STAFResult -** the STAF result from starting the process. If the process failed to start successfully, it contains any error messages from starting the process. It is a string.

- **STAXResult -** contains any files specified by **returnstdout**, **returnstderr**, and/or **returnfile(s)**. If no files are returned, STAXResult is set to special Python object None. If one or more files are returned, the format of STAXResult is a list as follows:

  ```
  [ [<File 1 rc>, <File 1 data>], [<File 2 rc>, <File 2 data> ], ... ]
  ```

  Each entry in the list represents a returned file and consists of a 2-element list as follows:
  1. <File n rc> is a number containing the STAF return code indicating the success or failure of retrieving the file's contents
  2. <File n data> is a string containing the file's contents

  Files will be returned in the order of standard output, then standard error, then any files specified with the **returnfile(s)** element(s).

  For example, assume that the standard output of a process was simply "This is stdout data", and that the standard error of a process was "This is stderr data", and that you asked for both of these to be returned. STAXResult would look like the following.

```
[ [0, 'This is stdout data'], [0, 'This is stderr data'] ]
```

The process element has one attribute:

- **name -** This element is optional. It is the name that the STAX Monitor uses to refer to the <process> element. It defaults to Process<number>, where <number> is a unique number for each process executed in a job.

The process element contains the following elements in the order listed (with some variations). Refer to the "STAX Document Type Definition (DTD)" section to see the DTD for the process element.

Note that these elements are equivalent to the options allowed for the STAF Process Service (except where noted), so see the STAF User's Guide for more information.

The process element has the following required elements:

- **location -** Location specifies the machine where the process should be run. This element is required.

- **command -** This element is required.

  This element has the following optional attributes:

  - **mode -** specifies whether the command should be executed as though you were at a shell prompt. The value must evaluate via Python to one of the following:
    - 'default' - specifies that the command should not be started via a separate shell. This is the default.
    - 'shell' - specifies that the command should be started via a separate shell. This allows complex commands involving pipelines to be readily executed.
  - **shell -** specifies the shell to use when starting the process via a separate shell. This attribute is used only if the command's mode is set to 'shell'. The value overrides any shell defaults specified in the STAF configuration file. The value is evaluated via Python to a string.

The process element has the following optional elements. Each of these optional elements may specify an **if** attribute. The **if** attribute must evaluate via Python to a true or false value. If it does not evaluate to a true value, the element is ignored. The default value for the **if** attribute is 1, a true value. Note that in Python, true means any nonzero number or nonempty object; false means not true, such as a zero number, an empty object, or None. Comparisons and equality tests return 1 or 0 (true or false).

- **parms -** specifies any parameters that you wish to pass to the command. The value is evaluated via Python to a string. This element is optional.

- **workdir -** specifies the directory from which the command should be executed. If you do not specify this element, the command will be started from whatever directory STAFProc is currently in. The value is evaluated via Python to a string. This element is optional.

- **title -** specifies the program title of the process. Unless overridden by the process, the title will be the text that is displayed on the title bar of the application. The value is evaluated via Python to a string. This element is optional.

- **workload -** specifies the name of the workload for which this process is a member. This may be useful in conjunction with other process elements. The value is evaluated via Python to a string. This element is optional.

- **vars** (or **var**) **-** specifies STAF variables that go into the process specific STAF variable pool. The value must evaluate via

Python to a string or a list of strings. Multiple vars elements may be specified for a process. The format for each variable is: 'varname=value' So, a list containing 3 variables could look like: ['var1=value1', 'var2=value2', 'var3=value3']. Specifying only one variable could look like either: ['var1=value1'] or 'var1=value1'. There can be 0 or more occurrences of this element.

- **envs** (or **env**) - specifies environment variables that will be set for the process. Environment variables may be mixed case, however most programs assume environment variable names will be uppercase, so, in most cases, ensure that your environment variable names are uppercase. The value must evaluate via Python to a string or a list of strings. Multiple envs elements may be specified for a process. The format for each variable is: 'varname=value' So, a list containing 3 variables could look like: ['ENV_VAR_1=value1', 'ENV_VAR_2=value2', 'ENV_VAR_3=value3']. Specifying only one variable could look like either: ['ENV_VAR_1=value1'] or 'ENV_VAR_1=value1'. There can be 0 or more occurrences of this element.

- **useprocessvars** - specifies that STAF variable references should try to be resolved from the STAF variable pool associated with the process being started first. If the STAF variable is not found in this pool, the STAF global variable pool should then be searched. This element is optional and is an empty element.

- **username** - specifies the username under which the process should be started. The value is evaluated via Python to a string. This element is optional.

- **password** - specifies the password with which to authenticate the user specified with the username element. The value is evaluated via Python to a string. This element is optional.

- **disabledauth** - allows you to specify the action to take if a username/password is specified but authentication has been disabled. This element is optional and is an empty element.

  This element has the following required attribute (in addition to the **if** attribute):

  - **action** - Must evaluate via Python to a string containing either:
    - 'error' - specifies that an error should be returned.
    - 'ignore' - specifies that any username/password specified is ignored if authentication is desabled.

    This action overrides any default specified in the STAF configuration file.

- **stdin** - specifies the name of the file from which standard input will be read. The value is evaluated via Python to a string. This element is optional.

- **stdout** - specifies the name of the file to which standard output will be redirected. The value is evaluated via Python to a string. This element is optional.

  This element has the following required attribute (in addition to the **if** attribute):

  - **mode** - specifies what to do if the file already exists. The value must evaluate via Python to one of the following:
    - 'replace' - specifies that the file will be replaced. This is the default.
    - 'append' - specifies that the process' standard output will be appended to the file.

- **stderr** - specifies the file to which standard error will be redirected. The value is evaluated via Python to a string. This element is optional.

  This element has the following attribute (in addition to the **if** attribute):

  - **mode** - specifies what to do if the file already exists or to redirect standard error to the same file as standard output. The value must evaluate via Python to one of the following:

        - 'replace' - specifies that the file will be replaced. This is the default.
        - 'append' - specifies that the process' standard error will be appended to the file.
        - 'stdout' - specifies that standard error will be redirected to the same file to which standard output is being redirected. If a file name is specified, it is ignored.

- **returnstdout** - specifies to return in STAXResult the contents of the file where standard output was redirected when the process completes. This element is optional and is an empty element.

- **returnstderr** - specifies to return in STAXResult the contents of the file where standard error was redirected when the process completes. This element is optional and is an empty element.

- **returnfiles** (or **returnfile**) - specifies to return in STAXResult the contents of the specified file(s) when the process completes. The value must evaluate via Python to a string (containing a file name) or a list of strings (each containing a file name). There can be 0 or more occurrences of this element.

- **stopusing** - allows you to specify the method by which this process will be STOPed, if not overridden on the STOP command. The value is evaluated via Python to a string. This element is optional.

- **console** - allows you to specify if the process should get a new console window or share the STAFProc console. This element is optional and is an empty element.

  This element has the following required attribute (in addition to the **if** attribute):

    - **use** - Must evaluate via Python to a string containing either:
        - 'new' - specifies that the process should get a new console window. This option only has effect on Win32. This is the default for Win32 and OS/2 systems.
        - 'same' - specifies that the process should share the STAFProc console. This option only has effect on Win32. This is the default for Unix systems.

- **statichandlename** - specifies that a static handle should be created for this process. The value is evaluated via Python to a string. It will be the registered name of the static handle. Using this option will also cause the environment variable STAF_STATIC_HANDLE to be set appropriately for the process. This element is optional.

- **other** - specifies any other STAF parameters that may arise in the future. It is used to pass additional data to the STAF PROCESS START request. The value is evaluated via Python to a string. This element is optional.

- **process-action** - specifies a task which will be executed after the process has started. This task will be executed in parallel with the process via a new STAX-Thread. The task will be able to use the variable STAXProcessHandle to obtain the process' handle in order to interact with the process. If the process completes before the task completes, the process will remain in a non-complete state until the task completes. If the process cannot be started, the process-action task is not executed. This element is optional, but if specified must be the last element in the <process> element.

  Note: To create a global variable that can be accessed across STAX-Threads, use the STAXGlobal class described in "Function" section.

Options allowed for the STAF Process Service which are not allowed for the STAX process element are as follows.

- WAIT
- ASYNC
- NOTIFY ONEND

These options should not be put in the **other** element.

Usage:

In the following example of a **process** element, a Java program is executed. When the process completes, the **if** element is run which checks the return code from the process.

```
<sequence>

  <process name="'TestProcess'">
    <location>'local'</location>
    <command>'java'</command>
    <parms>'com.ibm.staf.service.stax.TestProcess 5 1 0'</parms>
    <title>'Test Process'</title>
  </process>

  <if expr="RC != 0">
    <raise signal="'NonZeroRCError'"/>
  </if>

</sequence>
```

In the following example of a **process** element, a ping command is executed as though you were at a shell prompt. The ping is executed within an infinite loop contained within a timer. If the ping command does not complete successfully (indicated by RC 0) within 30 seconds, a failure message is sent.

```
<sequence>

  <timer duration="'30s'">
    <loop until="RC == 0">
      <process name="'Ping'">
        <location>'local'</location>
        <command mode="'shell'">'ping -n 1 -w 1 %s' % machName</command>
      </process>
    </loop>
  </timer>

  <if expr="RC != 0">
    <message>'Ping of machine %s failed' % machName</message>
  </if>

</sequence>
```

The following example of a **process** element shows many of the optional elements that a process can contain and shows the use of the **if** element to specify whether an optional element should be used based on an expression evaluated while the job is running.

```
<script>
  machName = 'local'
  opSys = 'Win32'
  className = 'com.ibm.staf.service.stax.TestProcess'
  commonEnvVarList = ['COMMON_ENV_VAR_1=value1','COMMON_ENV_VAR_2=value2']
</script>

<process name="'aProcess'">
  <location>machName</location>
  <command>'java'</command>

  <parms if="opSys != 'Linux'">
    '%s 2 15 100' % className
  </parms>

  <title>'Title example for process with many elements'</title>

  <vars if="opSys == 'Win32'">
    ['tempPath=C:/temp', 'winRunPath=C:/temp/processa']
  </vars>

  <vars if="opSys == 'Linux'">
    ['tempPath=/test/temp']
  </vars>

  <var>
    'commonMachName=%s' % (machName)
  </var>

  <envs if="opSys == 'Win32'">
    ['TEMP_DIR=C:/temp']
  </envs>

  <envs>commonEnvVarList</envs>

  <useprocessvars if="opSys == 'Win32'"/>

  <disabledauth if="opSys == 'Win32'" action="'ignore'"/>

  <stdout mode="'replace'">
    'c:/temp/aProcess.out'
  </stdout>

  <stderr mode="'append'">
    'c:/temp/aProcess.err'
  </sdterr>

  <console if="opSys == 'Win32'" use="'same'"/>

</process>
```

In the following example of a **process** element, a command which writes to stdout and stderr and produces a couple of files (C:\process1.inf and C:\process2.inf) is run. The contents of the stdout file and the two additional files are returned in STAXResult when the process completes. Note that the stdout file also contains stderr output because <stderr> specified mode 'stdout' instead of specifying a different file name. Then the contents all returned files are written to one central place, the STAX Job User Log.

```
<sequence>

  <process>
    <location>machName</location>
    <command>cmd</command>
    <stdout>'C:/temp.out'</stdout>
    <stderr mode="'stdout'"/>
    <returnstdout/>
    <returnfiles>['C:/process1.inf', 'C:/process2.inf']</returnfiles>
  </process>

  <if expr="RC != 0">
    <log level="'error'">
      'Process failed with RC=%s, Result=%s' % (RC, STAFResult)
    </log>

    <elseif expr="STAXResult != None">
      <iterate var="fileInfo" in="STAXResult" indexvar="i">
        <if expr="fileInfo[0] == 0">
          <sequence>
            <log level="'info'">fileInfo[1]</log>
          </sequence>
          <else>
            <log level="'error'">
              'Retrieval of file %s contents failed with RC=%s' % (i, fileInfo[0])
            </log>
          </else>
        </if>
      </iterate>
    </elseif>

    <else>
      <log level="'info'">'STAXResult is None'</log>
    </else>

  </if>

</sequence>
```

In the following example of a **process** element, the following shell-style command is executed, "grep 'Node Count = ' /tests/cli.out | awk '{print $8}'" redirecting its standard output and standard error to /tests/awk.out and returning the output. Note the use of the caret (^) as an escape character for "{" so that it doesn't try to resolve a variable named "print $8".

```
<process name="'Grep_and_awk_numClustNodes'">
  <location>machName</location>
  <command mode="'shell'">
    "/bin/grep 'Node Count = '  /tests/cli.out | awk '{print $8}'"
  </command>
  <stdout mode="'replace'" >'tests/awk.out'</stdout>
  <stderr mode="'stdout'"/>
  <returnstdout/>
</process>
```

In the following example of a **process** element, the command is started in a separate Cygwin shell on Windows, redirecting its standard output and standard error to D:/temp/copy.out and returning the output (if any).

```
<process name="'copyFiles'">
  <location>machName</location>
  <command mode="'shell' shell="'D:/Cygwin/bin/bash -c %C'">
    'cp -fr D:/tests/test1/*.java D:/output/test1'
  </command>
  <stdout mode="'replace'" >'D:/temp/copy.out'</stdout>
  <stderr mode="'stdout'"/>
  <returnstdout/>
</process>
```

In the following example of a **process** element, a **process-action** element is specified. The **process-action** task will be executed after the process starts.

```
<script>
  machName = 'local'
  className = 'com.ibm.staf.service.stax.TestProcess'
  msg = 'Data being sent to the Process Queue after it starts executing'
</script>

<process name="'aProcess'">
  <location>machName</location>
  <command>'java'</command>
  <parms>'className'</command
  <process-action>
    <sequence>
      <stafcmd>
        <location>machName</location>
        <service>'queue'</service>
        <request>'queue handle %s %s' % (STAXProcessHandle, msg)</request>
      </stafcmd>
    </sequence>
  </process-action>
  </process>
```

stafcmd

The **stafcmd** element represents a STAF command which will be executed on a specified machine.

After the STAF command has completed, the following variables are set and can be referenced by the job:

- **RC -** the return code from the STAF command. It is an integer.
- **STAFResult -** the result from the STAF command. It is a string.

The **stafcmd** element has one attribute:

- **name -** This element is optional. It is the name that the STAX Monitor uses to refer to the <stafcmd> element. It defaults to STAFCommand<number>, where <number> is a unique number for each STAF command executed in a job.

The **stafcmd** element contains the following elements in the order listed:

- **location -** It specifies the machine where the STAF command should be run. This element is required.
- **service -** It specifies the name of the service to which you are submitting a request. This element is required.
- **request -** It specifies the actual request string that you wish to submit to the service. This element is required.

Usage:

In the following example of a **stafcmd** element, the STAF respool request is executed on the machine specified by a variable named resPoolServer. The STAF respool request is requesting a machine name from a pool specified by a variable named clientPool. When the STAF command completes, the **if** element checks the return code variable set by the STAF command. If the return code is 0 (aka STAFRC.STAFOk), the value of the STAFResult variable is stored to another variable named machName.

```
<sequence>

  <script>resPoolServer = "server1.austin.ibm.com"</script>
  <script>clientPool = "clientMachinePool"</script>

  <stafcmd name="'Respool Request Pool'">
    <location>resPoolServer</location>
    <service>'respool'</service>
    <request>'request pool %s' % (clientPool)</request>
  </stafcmd>

  <if expr="RC == STAFRC.STAFOk">
    <script>machName = STAFResult</script>
    <else>
      <raise signal="'NonZeroRCError'"/>
    </else>
  </if>

</sequence>
```

In the following example of a **stafcmd** element, a STAF notify request sends notification message "STAX Job ID 6 failed" to Jane Smith. STAFUtil's wrapData() method is used to turn strings containing spaces into the colon-length-colon form needed for submitting a STAF Notify request. Note that the java class STAFUtil needs to be imported in order to use its wrapData() method.

```
<sequence>
  <script>
    NotifyProfile = 'Jane Smith'
    NotifyServer = 'server1'
    Message = 'STAX Job ID %s failed.' % (STAXJobID)

    from com.ibm.staf import STAFUtil

    Request = ('NOTIFY PROFILE %s LEVEL NORMAL MESSAGE %s' % \
              (STAFUtil.wrapData(NotifyProfile), STAFUtil.wrapData(Message)))
  </script>

  <stafcmd>
    <location>NotifyServer</location>
    <service>'notify'</service>
    <request>Request</request>
  </stafcmd>

</sequence>
```

job

The **job** element represents a sub-job which will be executed within the parent job.

Notes:

- A sub-job will appear as a separate job.
- Terminating a parent job will terminate any sub-jobs as well.
- Holding/releasing a parent job will not hold/release its sub-jobs.

After a sub-job has completed (or if it could not be started) the following variables are set and can be referenced by the job:

- **RC** - the return code from submitting the request to execute the sub-job. It is an integer.

- **STAFResult -** the STAF result from submitting the request to execute the sub-job. If the sub-job failed to start successfully, it contains any error messages from trying the execute the sub-job. It is a string.

- **STAXSubJobID -** the job ID of the sub-job. If the sub-job was not successfully submitted for execution (e.g. due to a command parsing error, an XML parsing error, etc.), it is set to 0. It is an integer.

- **STAXResult -** contains the result returned by the starting function of the sub-job. If nothing is returned (due to a problem submitting the request to execute the sub-job, or an error running the sub-job, or if an empty <return> element or if no <return> element is specified in the starting function), STAXResult is set to special Python object None.

The job element has the following optional attributes:

- **name -** Specifies the name of the sub-job that is used to identify the sub-job. This attribute is equivalent to the JOBNAME option for a STAX EXECUTE command.

- **clearlogs -** specifies whether to delete the STAX Job and Job User logs before the sub-job is executed to ensure that only this sub-job's contents are in the log. This attribute is equivalent to the CLEARLOGS option for a STAX EXECUTE command. The value must evaluate via Python to one of the following (not case-sensitive):
  - 'parent' - specifies to use the same value that was specified for the parent job. This is the default.
  - 'default' - specifies to use the default STAX service setting for "Clear Logs".
  - 'enabled' - specifies to clear its job logs
  - 'disabled' - specifies to not clear its job logs
  - 'otherwise, if it evaluates to a true value, the job logs are cleared; if it evaluates to a false value, the job logs are not cleared.

- **monitor -** If it evaluates to true, it specifies that the job should automatically be monitored by the STAX Monitor (when the current job is already being monitored by the STAX Monitor). Note that "Automatically monitor recommended sub-jobs" must be selected in the STAX Job Monitor properties in order for it to be used. The default value is 0, a false value.

- **logtcelapsedtime -** specifies whether to log the elapsed time for testcases in the summary records at the end of a STAX Job log and in the LIST TESTCASES output for the sub-job. This attribute is equivalent to the LOGTCELAPSEDTIME option for a STAX EXECUTE command. The value must evaluate via Python to one of the following (not case-sensitive):
  - 'parent' - specifies to use the same value that was specified for the parent job. This is the default.
  - 'default' - specifies to use the default STAX service setting for "Log TC Elapsed Time".
  - 'enabled' - specifies to log the elapsed time for testcases
  - 'disabled' - specifies to not log the elapsed time for testcases
  - 'otherwise, if it evaluates to a true value, the elapsed time for testcases is logged; if it evaluates to a false value, the elapsed time for testcases is not logged.

- **logtcnumstarts -** specifies whether to log the number of starts for testcases in the summary records at the end of a STAX Job log and in the LIST TESTCASES output for the sub-job. This attribute is equivalent to the LOGTCNUMSTARTS option for a STAX EXECUTE command. The value must evaluate via Python to one of the following (not case-sensitive):
  - 'parent' - specifies to use the same value that was specified for the parent job. This is the default.
  - 'default' - specifies to use the default STAX service setting for "Log TC Num Starts".

  - 'enabled' - specifies to log the number of starts for testcases
  - 'disabled' - specifies to not log the number of starts for testcases
  - 'otherwise, if it evaluates to a true value, the number of starts for testcases is logged; if it evaluates to a false value, the number of starts for testcases is not logged.

- **logtcstartstop** - specifies whether to log 'start' and 'stop' level records for testcases in the STAX Job log. This attribute is equivalent to the LOGTCSTARTSTOP option for a STAX EXECUTE command. The value must evaluate via Python to one of the following (not case-sensitive):
  - 'parent' - specifies to use the same value that was specified for the parent job. This is the default.
  - 'default' - specifies to use the default STAX service setting for "Log TC Start/Stop".
  - 'enabled' - specifies to log the 'start' and 'stop' records for testcases
  - 'disabled' - specifies to not log the 'start' and 'stop' records for testcases
  - 'otherwise, if it evaluates to a true value, the 'start' and 'stop' records for testcases are logged; if it evaluates to a false value, the 'start' and 'stop' records for testcases are not logged

The job element contains the following elements in the order listed (with some variations). Refer to the "STAX Document Type Definition (DTD)" section to see the DTD for the job element.

Note that these elements are equivalent to the options allowed for the EXECUTE request (except where noted), so refer to the "EXECUTE" section for more information.

The job element must contain either a job-file or job-data element as follows:

- **job-file** - Specifies the fully qualified name of a file containing the XML document for the sub-job to be executed. The job-file element is equivalent to the FILE option for an EXECUTE request. This element has the following optional attribute:
  - **machine** - specifies the name of the machine where the XML document is located. If not specified, it defaults to Python variable STAXJobXMLMachine. The machine attribute is equivalent to the MACHINE option for an EXECUTE request.

- **job-data** - Specifies a string containing the XML document contents for the job to be executed. This element is equivalent to the DATA option for an EXECUTE request. This element has the following optional attribute:
  - **eval** - specifies whether the data is to be evaluated by Python in the parent job. This value must evaluate via Python to a true or false value. For example, if the job-data value is dynamically generated and assigned to a Python variable, rather then just containing the literal XML value, you would need to set the **eval** attribute to true (e.g. eval="1"). The default is false (eval="0").

The job element has the following optional elements. Each of these optional elements may specify an **if** attribute. The **if** attribute must evaluate via Python to a true or false value. If it does not evaluate to a true value, the element is ignored. The default value for the **if** attribute is 1, a true value. Note that in Python, true means any nonzero number or nonempty object; false means not true, such as a zero number, an empty object, or None. Comparisons and equality tests return 1 or 0 (true or false).

- **job-function** - specifies the name of the function element to call to start the job, overriding the defaultcall element, if specified, in the XML document. It must specify the name of a function element specified in the XML document. This element is equivalent to the FUNCTION option for an EXECUTE request. The value is evaluated via Python to a string. This element is optional.

- **job-function-args** - specifies arguments to pass to the function element called to start the job, overriding any arguments for the defaultcall element, if specified, in the XML document. This element is equivalent to the ARGS option for an EXECUTE request. This element is optional and has the following optional attribute:
  - **eval** - specifies whether the value is to be evaluated by Python in the parent job. The default is false (eval="0").

- **job-script** - specifies Python code to be executed. This element is equivalent to the SCRIPT option for an EXECUTE

request. Multiple job-script elements may be specified for a job. This element has the following optional attribute:
    - **eval** - specifies whether the value is to be evaluated by Python in the parent job. The default is false (eval="0").

- **job-scriptfile** (or **job-scriptfiles**) - specifies the fully qualified name of a file containing Python code to be executed, or a list of file names containing Python code to be executed. The value must evaluate via Python to a string or a list of strings. This element is equivalent to the SCRIPTFILE option for a STAX EXECUTE command.

  Specifying only one script file could look like either:

  ```
  'C:/stax/scriptfiles/scriptfile1.py'   or    ['C:/stax/scriptfiles/scriptfile1.py']
  ```

  Specifying a list containing three script files could look like:

  ```
  ['C:/stax/scriptfiles/scriptfile1.py', 'C:/stax/scriptfiles/scriptfile2.py',
  'C:/stax/scriptfiles/scriptfile3.py']
  ```

  The job-scriptfile element has the following optional attribute:
    - **machine** - specifies the name of the machine where the script file(s) are located. If not specified, it defaults to Python variable STAXJobScriptFileMachine. This attribute is equivalent to the SCRIPTFILEMACHINE option for an EXECUTE request.

- **job-action** - specifies a task which will be executed after the sub-job has started execution. This task will be executed in parallel with the sub-job via a new STAX-Thread. The task will be able to use the variable STAXSubJobID to obtain the sub-job's job ID in order to interact with the sub-job, if desired. If the sub-job completes before the task completes, the sub-job will remain in a non-complete state until the task completes. If the sub-job cannot start execution, the job-action task is not executed. This element is optional, but if specified, it must be the last element in the <job> element.

  Note: To create a global variable that can be accessed across STAX-Threads, use the STAXGlobal class described in "Function" section.

Options allowed for the EXECUTE command which are not allowed for the job element are as follows.

- WAIT
- HOLD
- TEST

Usage:

In the following example of a **job** element, a sub-job defined by an XML file named C:/stax/xml/myJob2.xml (located on the machine specified by STAXJobXMLMachine) is executed and given a job name of "MyJob". Since the monitor attribute is set to a true value, if the current job is being monitored by the STAX Monitor, and the "Automatically monitor recommended sub-jobs" option has been selected in the STAX Job Monitor Properties, a STAX Monitor window will be opened automatically for the sub-job.

```
<job name="'Job 2'" monitor="1">
  <job-file>'C:/stax/xml/myJob2.xml'</job-file>
</job>
```

In the following example of a **job** element, a sub-job defined by an XML file named tests/testB/xml located on machine myMachine is executed and given a job name of 'Test B'. The job is started by calling function 'Main' and passing this function an argument list of [1, 'server']. In addition, two scriptfiles are specified as well as a couple of script elements. This sub-job is similar to the following STAX EXECUTE request:

```
  EXECUTE FILE /tests/testB.xml MACHINE myMachine JOBNAME "Test B" CLEARLOGS
          FUNCTION Main ARGS "[1, 'server1']" SCRIPTFILEMACHINE myMachine
          SCRIPTFILE /tests/testB1.py SCRIPTFILE /tests/testB2.py
          SCRIPT "MachineList = ['machA', 'machB'] SCRIPT "maxTime = '1h'"
```

In addition, a **job-action** element is run in parallel with the sub-job after the sub-job has been started. In this example, it simply logs a message containing the job ID for the sub-job being executed.
When the sub-job completes, the return code (RC) set when starting the sub-job is checked.

```
<job name="'Test B'" clearlogs="'Enabled'">
  <job-file machine="'myMachine'">'/tests/testB.xml'</job-file>
  <job-function>'Main'</job-function>
  <job-function-args>[1, 'server1']</job-function-args>
  <job-scriptfiles machine="'myMachine'">['/tests/testB1.py',
'/tests/testB2.py']</job-scriptfiles>
  <job-script>machineList = ['machA', 'machB']<job-script>
  <job-script>maxTime = '1h'</job-script>
  <job-action>
    <log>'Started sub-job %s' % (STAXSubJobID)</log>
  </job-action>
</job>

<if expr="RC == 0">
  <message>'Sub-job %s completed.  Result: %s' % (STAXSubJobID, STAXResult)</message>
  <else>
    <message>'Sub-job could not be started. RC: %s  Result: %s' % (RC,
STAFResult)</message>
  </else>
</if>
```

In the following example of a **job** element, a sub-job defined by an XML file named C:/tests/Scenario01.xml located on machine myMachine is executed and given a job name of 'Scenario 01'. The option to clear the job logs is enabled, and all of the testcase logging options are enabled as well.

```
  EXECUTE FILE C:/tests/Scenario01.xml MACHINE myMachine JOBNAME "Scenario 01"
          CLEARLOGS Enabled LOGTCELAPSEDTIME Enabled LOGTCNUMSTARTS Enabled
          LOGTCSTARTSTOP Enabled
```

```
<job name="'Scenario 01'" clearlogs="'Enabled'">
     logtcelapsedtime="'Enabled'" logtcnumstarts="'Enabled'"
     logtcstartstop="'Enabled'">
  <job-file machine="'myMachine'">'C:/tests/Scenario01.xml'</job-file>
</job>
```

nop

The nop element indicates that no operation should be performed. This element is typically used in conjunction with the if, elseif, and else elements.

Usage:

In the following example of a **nop** element, if an expression is true, then use the **nop** element to do nothing; else perform function "ErrorRoutine".

```
<if expr="RC == 0">
  <nop/>
  <else>
    <call function="'ErrorRoutine'"/>
  </else>
</if>
```

Sequential Execution: sequence

The sequence element may contain any number of STAX elements and executes them serially.

sequence

The **sequence** element represents a container of STAX elements which will be executed serially, in the order in which the contained elements are listed in the sequence. A **sequence** element may contain any number of ***task*** elements. You may nest sequence elements.

Usage:

In the following example of a **sequence** element, a variable is created. Then the **stafcmd** element is executed. When the **stafcmd** element completes, the **process** element is executed. When the **process** element completes, the **call** element is executed. When the **call** element completes, the **sequence** element is complete.

```
<sequence>

  <script>server1 = "machine1.test.austin.ibm.com"</script>

  <stafcmd>
    ...
  </stafcmd>

  <process>
    ...
  </process>

  <call function="'VerifyRC'"/>

</sequence>
```

Parallel Execution: parallel, paralleliterate

The parallel and paralleliterate elements execute tasks in parallel.

parallel

The **parallel** element represents a container of ***task*** elements which will be executed in parallel. Each ***task*** element it contains will be executed on a separate STAX-Thread and existing variables are cloned for each thread. The **parallel** element is considered to be complete when each of its contained ***task*** elements has completed. A **parallel** element may contain any number of ***task*** elements. You may nest **parallel** elements.

Note: To create a global variable that can be accessed across STAX-Threads, use the STAXGlobal class described in "Function" section.

Usage:

In the following example of a **parallel** element, the **stafcmd**, **process**, and **call** elements are executed at the same time. When all three tasks are complete, the **parallel** element is complete and processing will continue on to the next element defined after the </**parallel**> element.

```
<parallel>

  <stafcmd>
    ...
  </stafcmd>

  <process>
    ...
  </process>

  <call function="'VerifyRC'"/>

</parallel>
```

paralleliterate

The **paralleliterate** element contains a single ***task*** element. The **paralleliterate** element performs the task for each value in a list. The iterations of the contained ***task*** element are executed in parallel (unlike the **iterate** element whose tasks are performed serially). Each iteration will be executed on a separate STAX-Thread and existing variables are cloned for each thread. The **paralleliterate** element is considered to be complete when all its iterations of the ***task*** element have completed. ZZZ:

Note: To create a global variable that can be accessed across STAX-Threads, use the STAXGlobal class described in "Function" section.

The **paralleliterate** element has the following attributes:

- **var** - is the name of the variable which will contain the current item in the list/tuple being iterated. It is a literal and is required.
- **in** - is the list to be iterated. It is evaluated via Python and must evaluate to be a list or tuple. It is required. Note that the list may be assigned in many ways. Here are some examples of **in** attribute assignments:
  - the name of the variable that is a list or tuple: in="machList"
  - a literal list: in="['machA','machB','machC']"
  - a literal tuple: in="('testA','testB')"
  - a slice of a list/tuple: in="machList[1:]"
  - a concatenation of lists/tuples: in="machList1 + ['machD','machE']"

- **indexvar** - is the name of a variable which will contain the index of the current item in the list/tuple being iterated. It is a literal and is optional. Note that the index for the first element in the list/tuple is 0.

Usage:

The following example of a **paralleliterate** element runs ProcessA simultaneously on a group of machines whose names are contained in a list. The **paralleliterate** element is not complete until "ProcessA" has completed on all of the machines whose names are contained in the list.

```
<script>machList = ['machA','machB','machC','machD']</script>
<paralleliterate var="machName" in="machList">
  <process>
    <location>machName</location>
    <command>'ProcessA'</command>
  </process>
</paralleliterate>
```

The following example of a **paralleliterate** element submits a STAF request to the PING service to ping each machine in a list. If the ping fails, the name of the machine which could not be pinged is added to a list. The STAXGlobal class was used to store this list so that it can be accessed across STAX-Threads that are running in parallel.

```
<script>
  machineList = ['machA', 'machB', 'machC' ]
  gPingFailList = STAXGlobal([])
</script>

<paralleliterate var="machName" in="machineList">
  <sequence>

    <stafcmd>
      <location>machName</location>
      <service>'PING'</service>
      <request>'PING'</request>
    </stafcmd>

    <if expr="RC != 0">
      <script>gPingFailList.append(machName)</script>
    </if>

  </sequence>
</paralleliterate>

<if expr="len(gPingFailList) != 0">
  <message>
    'Could not ping the following machines: %s' % (gPingFailList.get())
  </message>
</if>
```

Functions: function, call, call-with-list, call-with-map, defaultcall, return, import

The function, call, call-with-list, call-with-map, defaultcall, return, and import elements deal with functions and how they are invoked.

function

To understand how to use the **function** element, you need to understand the following concepts:

- *<function> creates a function object and assigns it to a name*
  The **function** element creates a function object and assigns it to a name. The function name becomes a reference to the function object. There are really two sides to the function picture: a *definition* (the function element that creates a function) and a *call* (a call element that tells the STAX Job to run the function).

- *<return> sends a result object back to the caller*
  When a function is called, the caller stops until the function finishes its work and returns control to the caller. Functions that compute a value send it back to the caller with a **return** element. The **return** element can send back any sort of object, including multiple values.

- *Scope determines how variables are to be assigned*
  By default, all variables assigned in a function are global to the job's current STAX-Thread. However, you can assign a **local** scope to a function such that all names assigned in a function are local to that function and exist only while the function runs. Functions defined with a local scope provide a nested namespace, which contains a copy of variables in the caller's scope and which localizes any new variables created or changes to existing variables.

- *STAXGlobal Class provides for truly global variables*
  An exception to the above scoping rules are variables that are instances of the STAXGlobal class. A STAXGlobal class is a Python class which STAX provides as a wrapper to provide for the creation of truly global variables, even across STAX-Threads and when used in functions declared with a local scope. It provides get() and set(value) methods.

  A STAXGlobal variable allows you to modify it's value if it is passed into a local-scoped function or if it is shared across multiple threads. However, **creating** a STAXGlobal **inside** a local-scoped function will not cause it to appear once that function has returned. Any variable created within a local-scoped function will disappear when that function ends. Normally, any updates to variables disappear as well, but STAXGlobals allow updates to persist, so long as the STAXGlobal variable was created before the local-scoped function was called.

  Note that you will generally want to use a list to define a STAXGlobal variable even if its value is just an integer or a string. For example, when function Main (see below) is called, the message it displays is (2, 3, 3, 3), not (3, 3, 3, 3) as you may have thought would be displayed.

```
<function name="Main" scope="local">
  <sequence>

    <script>
      ctr  = STAXGlobal(2)     # Be careful if you don't use a list
      gCtr = STAXGlobal([2])   # Instead, use a list to define
      gCtr2 = STAXGlobal(2)    # No list, but set method is used.
      gCtr3 = STAXGlobal(2)    # Be careful if you don't use a list
    </script>

    <call function="'TestSTAXGlobal'"/>

    <message>ctr, gCtr[0], gCtr2, gCtr3</message>
  </sequence>
</function>

<function name="TestSTAXGlobal" scope="local">
  <script>
    ctr = ctr + 1           # Now ctr is bound to an integer
                           #   object, not a STAXGlobal
    gCtr[0] = gCtr[0] + 1  # Still a STAXGlobal
    gCtr2.set(gCtr2 + 1)   # Still a STAXGlobal
    gCtr3 += 1             # Still a STAXGlobal
  </script>
</function>
```

The **function** element defines a named task and contains a single ***task*** element. A **function** element may only be defined within the root **stax** element.

The first function called when a job is started is determined by the **defaultcall** element or by the FUNCTION parameter of an EXECUTE request. Functions are called within a job definition file using the **call**, **call-with-list**, or **call-with-map** elements.

The **function** element has the following attributes:

- **name** - is the name of the function. It is required. No two function elements may have the same name. It is a literal. It cannot contain a variable as it is not evaluated via Python. It must be a proper XML name which means it must start with a letter, an underscore, or a colon. The next characters may be letters, digits, underscores, hyphens, periods, and colons (but no whitespace).

- **requires -** is used with the <import> element and specifies other functions, separated by a space, in the same job that must be imported along with the specified function. This allows the function's users to only import the function itself, without having to worry about any other functions it may call. This attribute is optional.

- **scope** - specifies the scope of the function. It is a literal and, if specified, must be set to either global or local. It is optional and the default is global.

    - **global** - the function uses the same namespace so that the effects of variables defined/changed inside the function are global (within the same STAX-Thread).

    - **local -** the function provides a nested namespace which localizes the variables it uses. All variables assigned in a function are local to that function and exist only while the function runs. Variables previously defined in other functions with a global scope are still visible, however, any changes to these variables or new variables created are not visible to the caller after the function completes, unless the variable is an instance of the STAXGlobal class. Changes to STAXGlobal variables persist after a function completes.

The **function** element can also optionally contain the following elements, in the order listed, before the ***task*** element:

- **function-prolog** element - Contains a textual description of the function that can be used when transforming a STAX XML document to a Function Description document in HTML. This information will be placed before the function argument table in the Function Description document in HTML. This element replaces the deprecated **function-description** element.

  Note: This information is not used by STAX in any way and is completely ignored. In particular, this value is never passed to the Python interpreter, and thus, it should be a literal, not a quoted string. If you want to use standard HTML markup such as <p>, <b>, and <ol> in the description, then enclose the text in a CDATA section.

- **function-epilog** element - Contains a textual description of the function that can be used when transforming a STAX XML document to a Function Description document in HTML. This information will be placed after the function argument table in the Function Description document in HTML.

  Note: This information is not used by STAX in any way and is completely ignored. In particular, this value is never passed to the Python interpreter, and thus, it should be a literal, not a quoted string. If you want to use standard HTML markup such as <p>, <b>, and <ol> in the description, then enclose the text in a CDATA section.

- One of the following elements can be specified to define formal function arguments. Defining arguments can help prevent unintended namespace collisions and allows STAX to detect argument validation errors when calling functions at runtime.

  - **function-no-args** - is an empty element that specifies that the function does not allow any arguments to be passed to it.

  - **function-single-arg** - contains either a <function-required-arg> or a <function-optional-arg> element.

  - **function-list-args** - specifies a list of arguments. This element can contain zero or more <function-required-arg> elements, followed by zero or more <function-optional-arg elements (but must contain at least one of these arguments), followed by an optional <function-other-args> element. Note, the order of the arguments is important. All <function-required-args> must precede all <function-optional-args>, and the <function-other-args> element must be last (if present).

  - **function-map-args** - specifies a map of arguments (recorded as name/value pairs). This element can contain any combination of one or more <function-required-arg> and <function-optional-arg> elements, followed by an optional <function-other-args> element. Note that the order of the <function-required-arg> and <function-optional-arg> elements is not relevant, unless an argument's default value specifies another argument's value. However, the <function-other-args> element must be specified last (if present).

  If more arguments are passed to the function when called than are defined (assuming a <function-other-args> element is not specified) or if not all required arguments are passed to the function when called, a STAXFunctionArgValidate signal is raised to indicate the failure and the function is not executed.

The function argument elements are defined as follows:

- **function-required-arg** - defines an argument that must be passed when calling a function. It may optionally contain a description for the argument. This element has the following attribute:
  - **name** - is the name of the argument as it will appear inside the function. It is a literal and is required.

- **function-optional-arg** - defines an argument that may optionally be passed when calling a function. It may optionally contain a description for the argument. This element has the following attributes:
  - **name** - is the name of the argument as it will appear inside the function. It is a literal and is required.
  - **default** - is the default value for the argument. Its value is evaluated via Python. It is optional, and defaults to the special Python object None. If an optional argument is not provided when calling a function, its default value is used.

- **function-other-args** - defines the name of a list (if defined within a <function-list-args> element) or a map (if defined within a <function-map-args> element) which will contain any additional arguments passed when calling a function. If additional arguments are not provided when calling a function, its value is either an empty list or an empty map, depending on whether it was defined within a <function-list-args> element or a <function-map-args> element, respectively. It may optionally contain a description for the argument. The <function-other-args> element has the following attribute:
  - **name -** is the name of the argument list or map as it will appear inside the function. It is a literal and is required.

A function that does not define its arguments is implicitly defined as:

```
<function-single-arg>
  <function-optional-arg name="STAXArg" default="None"/>
<function-single-arg>
```

Note that the function argument elements (**function-required-arg**, **function-optional-arg**, and **function-other-args**) can contain a description of the argument. This information, along with the values of the **function-prolog** element (or the deprecated **function-description** element) and the **function-epilog** element, can be used in conjunction with an XSLT stylesheet to generate a nicely formatted HTML file documentating functions and their associated arguments specified in a STAX job. Refer to the "Generating STAX Function Documentation using a XSL Stylesheet" section for more information.

Usage:

Goal: Define a simple function containing a sequence element (which can then contain any number of other elements).

```
<function name="FunctionA">
  <sequence>
    ...
  </sequence>
</function>
```

Goal: Define a function which you intend to import into other STAX XML job files. The requires attribute defines the two additional functions it requires so that they will be automatically imported as well when FunctionB is imported.

```
<function name="FunctionB" requires="FunctionC FunctionD">
  <sequence>
    ...
    <call function="'FunctionC'"/>
    ...
    <call function="'FunctionD'"/>
    ...
  </sequence>
</function>
```

Goal: Illustrate the use of local function scope and the STAXGlobal class. Note that only changes to globalVar (which is an instance of the STAXGlobal class) are visible after function Bar completes. Also, all existing variables are visible inside functions with "local" scope. Thus, variables localVar and globalVar are visible inside function Bar, even though function Bar has "local" scope and had not defined them. The following messages are displayed in the STAX Monitor when this example is run:

```
Before Bar: localVar=[1, 2], globalVar=[1, 2]
After  Bar: localVar=[1, 2], globalVar=[1, 2, 3]
```

```
<stax>

  <script>
    localVar = [1, 2]
    globalVar = STAXGlobal([1, 2])
  </script>

  <defaultcall function="Main"/>

  <function name="Main" scope="local">
```

```
    <sequence>

      <message>
        'Before Bar: localVar=%s, globalVar=%s" % (localVar, globalVar)
      </message>

      <call function="'Bar'"/>

      <message>
        'After  Bar: localVar=%s, globalVar=%s' % (localVar, globalVar)
      </message>

    </sequence>

  </function>

  <function name="Bar" scope="local">

    <script>
      localVar.append(3)
      globalVar.append(3)
    </script>

  </function>

</stax>
```

Goal: Illustrate the specification of a function which does not allow any arguments to be passed to it. If any arguments are passed to it when called, a STAXFunctionArgValidate signal is raised and the function is not run.

```
  <function name="NoArgsFunction">
    <function-no-args/>

    <sequence>
      ...
    </sequence>

  </function>
```

Goal: Illustrate the specification of a function which requires one argument, duration, to be passed to it. If zero or more than one argument is passed to it when called, a STAXFunctionArgValidate signal is raised and the function is not run.

```
  <function name="OneRequiredArgFunction" scope="local">

    <function-single-arg>
      <function-required-arg name="duration"/>
    </function-single-arg>

    <timer duration="duration">
      <loop>
        ...
      </loop>
    </timer>

  </function>
```

```
This function could be called in any of the following ways with the same result:

<call function="'OneRequiredArgFunction'">'24h'</call>

<call-with-list function="'OneRequiredArgFunction'">
  <call-list-arg>'24h'</call-list-arg>
</call-with-list>
```

Goal: Illustrate the specification of a function which requires two map arguments (returnCode and result) and has one optional argument (msg). If the two required arguments are not passed to it when called, a STAXFunctionArgValidate signal is raised and the function is not run. A function prolog element is provided to describe what this function does and descriptions of the arguments passed to the function are also provided.

```
<function name="Check-STAFCmd-RC" scope="local">

  <function-prolog>
    Checks if a STAFCmd was successful and updates testcase status
  </function-prolog>

  <function-map-args>

    <function-required-arg name="returnCode">
      Return Code from a STAF Command
    </function-required-arg>

    <function-required-arg name="result">
      Result from a STAF Command
    </function-required-arg>

    <function-optional-arg name="msg" default="''">
      Message to display if an error occurs
    </function-optional-arg>

  </function-map-args>

  <if expr="RC == 0">
    <tcstatus result="'pass'"/>
    <else>
      <tcstatus result="'fail'">
        '%s; RC=%s, Result=%s' % (msg, returnCode, result)
      </tcstatus>
    </else>
  </if>

</function>

This function could be called in any of the following ways with the same result:

<call function="'Check-STAFCmd-RC'">
  { 'returnCode': RC, 'result': STAFResult, 'msg': 'This is the error message' }
</call>

<call-with-map function="'Check-STAFCmd-RC'">
  <call-map-arg name="'result'">STAFResult</call-map-arg>
  <call-map-arg name="'returnCode'">RC</call-map-arg>
  <call-map-arg name="'msg'">'This is the error message'<call-map-arg>
  </call-with-map>
```

Goal: Illustrate the specification of a function which requires a list argument (machName) and may have any number of additional arguments which will be stored in a list called testList. This example also shows the use of a STAXGlobal variable which is updated across STAX-Threads.

```
<function name="RunTests" scope="local">

    <function-list-args>
      <function-required-arg name="machName"/>
      <function-other-args name="testList"/>
    </function-list-args>

    <sequence>

      <script>
        testsRun = STAXGlobal([0])   # Number of tests run
      </script>

      <paralleliterate var="testName" in="testList">

        <sequence>

          <process>
            <location>machName</location>
            <command mode="'shell'">testName</command>
          </process>

          <script>testsRun[0] += 1</script>

        </sequence>

      </paralleliterate>

      <message>'Ran %s tests' % testsRun[0]</message>

    </sequence>

  </function>

  This function could be called in any of the following ways with the same result:

  <call function="'RunTests'">
    'local', 'ping machineA', 'dir C:\ > C:\out'
  </call>

  <call function="'RunTests'">
    [ 'local', 'ping machineA', 'dir C:\ > C:\out' ]
  </call>

  <call-with-list function="'RunTests'">
    <call-list-arg>'local'</call-list-arg>
    <call-list-arg>'ping machineA'</call-list-arg>
    <call-list-arg>'dir C:\ > C:\out'<call-list-arg>
  </call-with-list>
```

Goal: Illustrate the specification of a function that includes a complete desription of the function using the **function-prolog** and **function-epilog** elements. These elements utilize a CDATA section so that the text can include standard HTML markup so that when transformed via an XSLT processor, the text is easily readable. Note this function is actually provided in the sample STAXUtil.xml file provides in the STAX zip/tar file.

```
<function name="STAXUtilLogAndMsg" scope="local">

  <function-prolog>
    <![CDATA[
    <p>
      Logs a message and sends the message to the STAX Monitor.
      It's a shortcut for specifying the <message> and <log> elements
      for the same message.
    </p>
    ]]>
  </function-prolog>

  <function-epilog>
    <![CDATA[
    <h4>Returns:</h4>
    <p>Nothing.  That is, STAXResult = None.</p>
    <h4>Example:</h4>
    <pre>
<call function="'STAXUtilLogAndMsg'">'Here is my message'</call></pre>
    ]]>
  </function-epilog>

  <function-list-args>

    <function-required-arg name="message">
      The message you want to log in the STAX Job User log and to send to
      the STAX Monitor.
    </function-required-arg>

    <function-optional-arg name="level" default="'info'">
      The level of the message to be logged in the STAX Job User log.
    </function-optional-arg>

  </function-list-args>

  <sequence>

    <message>message</message>

    <log level="level">message</log>

  </sequence>

</function>
```

call

The **call** element specifies the name of a **function** element to be executed. When a **call** element is executed during the job execution, the function referred to is executed. The **call** element has the following required attribute:

- **function** - is the name of the function to call. Its value must evaluate via Python to a string. There must be a function element that exists whose name exactly matches, including case, the evaluated string.

Optionally, arguments may be passed when calling a function. The arguments are evaluated via Python in the caller's namespace. If no argument data is specified, then special Python object None is passed to the function. Any kind of argument data can be passed to functions using the <call> element and all of the types of function arguments (<function-no-args>, <function-single-arg>, <function-list-args>, or <function-map-args>) may be specified via this mechanism.

Usage:

Goal: Call a function named 'FunctionA', passing no arguments.

```
  <call function="'FunctionA'"/>
```

Goal: Serially call each function (passing no arguments) whose name is in a list.

```
  <iterate var="funcName" in="['FuncA','FuncB','FuncC','FuncD']">
    <call function="funcName"/>
  </iterate>
```

Goal: Call a function which expects one argument.

```
  <call function="'FunctionWithOneArg'">'Hi'</call>
```

Goal: Call a function which expects a list of three arguments.

```
  <call function="'FunctionWithThreeArgs'">
    5, 'This is a message', ['test1', 'test2']
  </call>
or
  <call function="'FunctionWithThreeArgs'">
    [ 5, 'This is a message', ['test1', 'test2'] ]
  </call>
```

Goal: Call a function which expects a map of two required values named "testList" and "machineList":

```
  <call function="'Foo'">
    {
      'testList' : ['test1', 'test2'],
      'machineList' : ['machine1', 'machine2']
    }
  </call>
```

call-with-list

The **call-with-list** element specifies the name of a **function** element to be executed. When a **call-with-list** element is executed during the job execution, the function referred to is executed.The **call-with-list** element has the following required attribute:

- **function** - is the name of the function to call. Its value must evaluate via Python to a string. There must be a function element that exists whose name exactly matches, including case, the evaluated string.

The **call-with-list** element can contain any number of **call-list-arg** elements. Each **call-list-arg** element contains a value for an argument which is evaluated via Python in the caller's namespace and will be passed to the function in the form of a list.

Usage:

Goal: Call a function named 'FunctionWithArgs' passing it three arguments in the form of a list.

```
<call-with-list function="'FunctionWithArgs'">
  <call-list-arg>5</call-list-arg>
  <call-list-arg>'This is a message'</call-list-arg>
  <call-list-arg>['test1', 'test2']</call-list-arg>
</call-with-list>
```

Note that this is equivalent to the following examples which use the **call** element instead:

```
<call function="'FunctionWithArgs'">
  5, 'This is a message', ['test1', 'test2']
</call>

<call function="'FunctionWithArgs'">
  [ 5, 'This is a message', ['test1', 'test2'] ]
</call>
```

call-with-map

The **call-with-map** element specifies the name of a **function** element to be executed. When a **call-with-map** element is executed during the job execution, the function referred to is executed. The **call-with-map** element has the following required attribute:

- **function** - is the name of the function to call. Its value must evaluate via Python to a string. There must be a function element that exists whose name exactly matches, including case, the evaluated string.

The **call-with-map** element can contain any number of **call-map-arg** elements. Each **call-map-arg** element has a required name attribute and contains an argument value. Both the name attribute and the argument value are evaluated via Python in the caller's namespace. The arguments are passed to the function in the form of a map of name/value pairs (also known as a dictionary in Python).

Usage:

Goal: Call a function named 'FunctionWithArgs' passing it three arguments in the form of a map (Python dictionary).

```
<call-with-map function="'FunctionWithArgs'">
  <call-map-arg name="'size'">5</call-map-arg>
  <call-map-arg name="'msg'">'This is a message'</call-map-arg>
  <call-map-arg name="'testList'">['test1', 'test2']</call-map-arg>
</call-with-map>
```

Note that this is equivalent to the following example which uses the **call** element instead:

```
<call function="'FunctionWithArgs'">
  {'size' : 5, 'msg' : 'This is a message', 'testList' : ['test1', 'test2'] }
</call>
```

defaultcall

The **defaultcall** element specifies the name of the function to call to start the job if no FUNCTION parameter is specified when the job is started using an EXECUTE request. The **defaultcall** element has the following required attribute:

- **function -** is the name of the function to call. It is a literal and so it cannot contain a variable as it is not evaluated via Python. There must be a function element that exists whose name exactly matches (including case).

Optionally, arguments may be passed via the **defaultcall** element. The arguments are evaluated via Python. If no argument data is specified, then special Python object None is passed to the function. Any kind of argument data can be passed to functions using the **defaultcall** element and all of the types of function arguments (**function-no-args**, **function-single-arg**, **function-list-args**, or **function-map-args**) may be specified via this mechanism.

A **defaultcall** element may only be defined within the root **stax** element, but it is not required. If a **defaultcall** element is not specified, a FUNCTION parameter on the STAX EXECUTE request must be specified.

Usage:

Goal: Call FunctionA by default to start the STAX job. No arguments are passed to FunctionA.

```
<stax>

  <defaultcall function="FunctionA"/>

  <function name="FunctionA">
    ...
  </function>
  ...

</stax>
```

Goal: Call FunctionA by default to start the STAX job. Pass a list of 2 arguments (duration and testList) to FunctionA.

```
<stax>

  <defaultcall function="FunctionA">[ '24h', ['machA', 'machB'] ]</defaultcall>

  <function name="FunctionA">
    <function-list-args>
      <function-required-arg name="duration"/>
      <function-optional-arg name="testList" default="['local']"/>
    </function-list-args>
    ...
  </function>
  ...

</stax>
```

return

The **return** element ends the function call and sends a result back to the caller. The **return** element is optional; if it's not present, a function exits when control flow falls off the end of the function body.

After the call of a function has completed, the **STAXResult** variable contains the result sent back from the call. It can be set to any type of object. For example, an integer, a list, a string, etc. This can be especially useful when the function called is defined with a **local** scope.

If no **return** element is specified within a function, or if no value is specified for the result object, STAXResult is set to the special Python None object. If an error occurred calling the function (e.g. invalid arguments, Python Evaluation error), STAXResult is set to a result object called STAXFunctionError.

Note that because the return sends back any sort of object, it can return multiple values, by packaging them as a tuple. Thus, call by reference can be simulated by returning tuples and assigning back to the original argument names in the caller. See the last example in the Usage section.

Usage:

Goal: Return control to the caller with STAXResult set to RC (e.g. an integer value set by a process or STAF command).

```
  <return>RC</return>
```

Goal: Return control to the caller with STAXResult set to None.

```
  <return/>
```

Goal: Return control to the caller with STAXResult set to a list. The caller can access the RC by specifying STAXResult[0] and the message by specifying STAXResult[1].

```
  <return>[RC, 'A descriptive message']</return>
```

Goal: Simulate call by reference by returning new values in a tuple and assigning the results to the caller's names. After the call, A = 3 and B = ['test1', 'test2', 'test3']

```
<function name="FunctionPassByReference" scope="local">

  <function-list-args>
    <function-required-arg name="x"/>
    <function-required-arg name="y"/>
  </function-list-args>

  <sequence>
    <script>
      x = x + 2
      y.append('test3')
    </script>
    <return>x, y</return>
  </sequence>

</function>

The above function is called from another function as follows:

<script>
  A = 1
  B = ['test1', 'test2']
</script>

<call function="'FunctionPassByReference'">A, B</call>

<script>
  A, B = STAXResult
</script>
```

import

The **import** element specifies a set of functions to be imported from another STAX XML job file. The **import** element has the following attributes:

- **machine** - is the name of the machine where the XML file is located, and is required. Its value is evaluated via Python.
- **file** - is the fully-qualified path to the XML job file from which the function(s) will be imported, and is required. Its value is evaluated via Python.
- **mode** - specifies what happens when an error occurs during a function import. It is optional. Its value must evaluate via Python to one of the following:
  - 'error' - specifies that a signal will be raised if an error occurs. This is the default.
  - 'ignore' - specifies that no signal will be raised if an error occurs. However, STAXResult[0] will contain the error.

The **import** element contains the following optional elements:

- **import-include** - It specifies a list of the the functions to import from the job file.
- **import-exclude** - It specifies a list of functions which will not be imported from the job file.

If **<import-include>** is not present, then all functions will be imported (bound by any exclude list). If **<import-exclude>** is not present, then no functions will be excluded.

The **<import-include>** and **<import-exclude>** elements support grep matching.

After executing an **import** element, STAXResult will be set to a list containing:

- STAXResult[0]: Either None or a list containing a STAXImportError object and a text string with details about the error.
- STAXResult[1]: A list of the successfully imported functions that were requested to be imported.
- STAXResult[2]: A list of the successfully imported functions that were required by other functions.
- STAXResult[3]: A list of the functions that were requested to be imported but already existed (so they were not imported).
- STAXResult[4]: A list of the functions that were required by other functions but already existed (so they were not imported).
- STAXResult[5]: A list of the functions that were not requested to be imported and were not required by other functions.
- STAXResult[6]: A list of functions requested to be imported that were not found.

If an error occurs while executing an **import** element, a STAXImportError signal will be raised if its mode is 'error'. When a STAXImportError signal is raised, the variable STAXSignalData will be set to a list containing an error type and an error description. The possible error types for STAXImportError are:

- STAXNoResponseFromMachine - the machine specified could not be contacted
- STAXFileCopyError - an error occurred while copying the xml file
- STAXXMLParseError - an XML parse error occurred while parsing the imported XML file

If you override the default Signal Handler for STAXImportError, you can access the error type in this manner:

```
  <if expr="STAXSignalData[0] is STAXNoResponseFromMachine">
```

Usage:

The **import** element may be specified anywhere except in the root <stax> element. This allows it to be executed at runtime, allowing Python expressions to be used in the element and enabling dynamic importing of functions. The element acts like any other element and is not executed until runtime.

Note that after an **import** element is executed, any other function can then call the imported function. So, for example, if functionA calls functionB and then functionC, and functionB imports functionX, functionC can call functionX without doing another import. If you have many functions to import, you can also create a function which does all of the imports and is the first function which is called in your job.

The following example of an **import** element imports all functions from file c:\util\library.xml, which is located on machine Server1A.

```
  <import machine="'Server1A'" file="'c:/util/library.xml'"/>
```

The following example of an **import** element only imports functions FunctionA and FunctionB.

```
  <import machine="'Server1A'" file="'c:/util/library.xml'">
      <import-include>'FunctionA', 'FunctionB'</import-include>
  </import>
```

The following example of an **import** element imports all functions except those that start with "FunctionA".

```
  <import machine="'Server1A'" file="'c:/util/library.xml'">
      <import-exclude>'FunctionA.*'</import-exclude>
  </import>
```

The following example of an **import** element imports all functions that start with "MyFuncs" but do not start with "MyFuncsWin32".

```
  <import machine="'Server1A'" file="'/usr/local/util/library.xml'">
      <import-include>'MyFuncs.*'</import-include>
      <import-exclude>'MyFuncsWin32.*'</import-exclude>
  </import>
```

Here's is a more complete snippet of a STAX job that shows an **import** element that is called by the job's starting function so that the imported functions can then be called throughout the job, from any function. This **import** element imports all of the functions provided in STAXUtil.xml. Refer to the "STAX Utility Functions" section for more information about common functions like STAXUtilLogAndMsg that are provided in the STAXUtil.xml file.

```
<stax>

  <defaultcall function="main"/>

  <script>
    # ImportMachine should be set to the machine where STAXUtil.xml resides
    # (e.g. 'local' if the file resides on the STAX service machine).
    # ImportDirectory should be set to the directory which contains STAXUtil.xml.
    ImportMachine = 'local'
    ImportDirectory = '{STAF/Config/STAFRoot}/services/libraries'
    ImportFile1 = '%s/STAXUtil.xml' % (ImportDirectory)
  </script>

  <function name="main">
    <sequence>

      <import machine="ImportMachine" file="ImportFile1"/>

      <call function="'STAXUtilLogAndMsg'">
        'This is the beginning of the job'
      </call>

      <call function="'FunctionA'"/>
      <call function="'FunctionB'"/>

    </sequence>
  </function>

  <function name="FunctionA">
    <sequence>

      <call function="'STAXUtilLogAndMsg'">
        'This is the beginning of FunctionA'
      </call>

      <!-- Add elements as needed -->

    </sequence>
  </function>

  <function name="FunctionB">
    <sequence>

      <call function="'STAXUtilLogAndMsg'">
        'This is the beginning of FunctionB'
      </call>

      <!-- Add elements as needed -->

    </sequence>
  </function>

</stax>
```

Loops: loop, iterate, break, continue

The loop, iterate, break, and continue elements deal with repeatedly executing a task.

loop

The **loop** element contains a single ***task*** element which may be executed a specified number of times, allowing specification of an upper and lower bound with an increment value and where the index counter is available to the contained ***task*** element. In addition, specification of a while and/or until expression is allowed. If no constraint attributes (e.g. to, until, or while) are specified for the **loop** element, then it loops "forever".

The **loop** element has the following attributes:

- **var** - is the name of the variable which will contain the loop index variable. It is optional.
- **from** - is the starting value of the loop index variable. It defaults to 1 if not specified.
- **to** - is the maximum value of the loop index variable. It is optional.
- **by** - is the increment value for the loop index variable. It defaults to 1 if not specified.
- **while** - is an expression that must evaluate to a boolean value and is performed at the top of each loop. If it evaluates to false, it breaks out of the loop. It is optional.
- **until** - is an expression that must evaluate to a boolean value and is performed at the bottom of each loop. If it evaluates to true, it breaks out of the loop. It is optional.

Usage:

The following example of a **loop** element executes a process five times.

```
<loop from="1" to="5">
  <process>
    <location>'machA.austin.ibm.com'</location>
    <command>'P3.exe'</command>
  </process>
</loop>
```

The following example of a **loop** element serially calls each function in a list named funcList until the return code set in a called function is not 0.

```
<script>funcList = ['Func1','Func2','Func3','Func4']</script>
<loop var="funcIndex" from="0" to="3" until="RC != 0">
  <call function="funcList[funcIndex]"/>
</loop>
```

The following example of a **loop** element loops "forever". The job will not end until the block containing the continuous loop is terminated. Function 'LongFunction' runs in parallel with a block that runs function 'ShortFunction' in a forever loop. When function 'LongFunction' completes, the block containing the "forever" loop is terminated so that the job may complete.

```
<parallel>

  <sequence>
    <call function="'LongFunction'"/>
    <terminate block name="'main.LoopForever'"/>
  </sequence>

  <block name="'LoopForever'">
    <loop>
      <call function="'ShortFunction'"/>
    </loop>
  </block>

</parallel>
```

iterate

The **iterate** element contains a single ***task*** element. The **iterate** element performs the task for each value in a list. The iterations of the contained ***task*** element are executed serially (unlike the **paralleliterate** element whose tasks are performed in parallel). The **iterate** element has the following attributes:

- **var -** is the name of the variable which will contain the current item in the list/tuple being iterated. It is a literal and is required.
- **in -** is the list to be iterated. It is evaluated via Python and must evaluate to be a list or tuple. It is required. Here are some examples of **in** attribute assignments:
    - the name of the variable that is a list or tuple: in="machList"
    - a literal list: in="['machA','machB','machC']"
    - a literal tuple: in="('testA','testB')"
    - a slice of a list/tuple: in="machList[1:]"
    - a concatenation of lists/tuples: in="machList1 + ['machD','machE']"
- **indexvar** - is the name of the variable which will contain the index of the current item in the list/tuple being iterated. It is a literal and is optional. Note that the index for the first element in the list/tuple is 0.

Usage:

The following example of an **iterate** element runs a STAF RESPOOL RELEASE request to release each machine name in a list.

```
<script>allocMachList = ['machA','machB','machC']</script>
<iterate var="machName" in="allocMachList">
  <stafcmd>
    <location>'machine1.austin.ibm.com'</location>
    <service>'RESPOOL'</service>
    <request>'RELEASE POOL ClientMachPool ENTRY %s' % machName</request>
  </stafcmd>
</iterate>
```

The following example of an **iterate** element runs each process whose name is in a list on a machine. In addition, the **iterate** element is nested within a **paralleliterate** element such that this occurs simultaneously on all machines in the machine list.

```
<paralleliterate var="machName" in="['machA','machB','machC']">
  <iterate var="procName" in="['proc1','proc2','proc3','proc4']">
    <process>
      <location>machName</location>
      <command>procName</command>
    </process>
  </iterate>
</paralleliterate>
```

break

The **break** element may be used to break out of a **loop** or **iterate** element.

Usage:

The following example of a **break** element breaks out of an **iterate** element when a non-zero return code is encountered. So if the list contains "ProcessA", "ProcessB", and "ProcessC" and ProcessA returns 0 but then ProcessB returns a non-zero RC, ProcessC will not be executed.

```
<iterate var="processName" in="processList">
  <sequence>
    <process>
      <location>'machineA'</location>
      <command>processName</command>
    </process>
    <if expr="RC != 0">
      <break/>
    </if>
  </sequence>
</iterate>
```

The following example of a **break** element breaks out of a **loop** element when a non-zero return code is encountered after running ShortTestProcess. However, since the loop contains a **parallel** element, when the break occurs, the other task running in parallel, LongTestProcess, will be killed in order to exit the loop.

```
<script>className = 'com.ibm.staf.service.stax.TestProcess'</script>

<function name="LoopParallelBreakTest">

  <loop var="i" from="0" by="1" to="3">

    <sequence>

      <message>'Beginning loop #%s' % i</message>

      <parallel>

        <sequence>

          <process name="'ShortTestProcess'">
            <location>machName</location>
            <command>'java'</command>
            <parms>'%s 2 2 %s' % (className, i-1)</parms>
          </process>

          <if expr="RC != 0">
            <sequence>
              <message>'  ShortTestProcess failed with RC=%s' % RC</message>
              <message>'  Breaking out of loop #%s' % i</message>
              <break/>
            </sequence>
            <else>
              <message>'  ShortTestProcess completed'</message>
            </else>
          </if>

        </sequence>

        <sequence>
          <process name="'LongTestProcess'">
            <location>machName</location>
            <command>'java'</command>
            <parms>'%s 4 2 0' % className</parms>
          </process>
          <message>'  TestProcess2 completed'</message>
        </sequence>

      </parallel>

      <message>'Completed loop #%s' % i</message>

    </sequence>

  </loop>

</function>
```

Note that the TestProcess class is provided as part of STAX. Its last input parameter is the return code that the process will return. The "Messages" output that you would see if you were monitoring a job running this function using the STAX Job Monitor would be:

```
Beginning loop #0
  ShortTestProcess completed
  LongTestProcess completed
Completed loop #0
Beginning loop #1
  ShortTestProcess completed
  LongTestProcess completed
Completed loop #1
Beginning loop #2
  ShortTestProcess failed with RC=1
  Breaking out of loop #2
```

continue

The **continue** element may be used to continue to the top of a **loop** or **iterate** element.

Usage:

The following example of a **continue** element continues to the top of the loop when a non-zero return code is encountered so that ProcessB is not executed if ProcessA fails on a loop iteration.

```
<loop var="i" from="1" to="10">
  <sequence>
    <process>
      <location>'machA.austin.ibm.com'</location>
      <command>'cmd.exe'</command>
      <parms>'ProcessA.exe'</parms>
    </process>
    <if expr="RC != 0">
      <continue/>
    </if>
    <process>
      <location>'machA.austin.ibm.com'</location>
      <command>'cmd.exe'</command>
      <parms>'ProcessB.exe'</parms>
    </process>
  </sequence>
</loop>
```

Conditional: if / elseif / else

The if element is a conditional that can control the logic and flow of the job.

if / elseif / else

The **if** element specifies a conditional. It allows you to specify a task to execute if an expression is evaluated to be true. It allows optional **elseif** elements and an optional **else** element to be performed if the expression is evaluated to be false. The **if**, **else**, and **elseif** elements may contain a single ***task*** element.

STAX uses Python as the expression evaluator engine.

Usage:

The following example of an **if** and **else** element checks the value of a variable named index to see if it has a zero remainder (even number), and if so, calls function Ogre1 and, otherwise, calls function Ogre2.

```
<if expr="(index % 2) == 0">
  <call function="'Ogre1'"/>
  <else>
    <call function="'Ogre2'"/>
  </else>
</if>
```

The following example uses a Python random number generator to determine which of four functions to randomly call:

```
<sequence>

  <script>from random import random</script>
  <script>r=random()*100</script>

  <if expr="r > 75">
    <call function="'Function1'"/>
    <elseif expr="r > 50">
      <call function="'Function2'"/>
    </elseif>
    <elseif expr="r > 25">
      <call function="'Function3'"/>
    </elseif>
    <else>
      <call function="'Function4'"/>
    </else>
  </if>
```

```
</sequence>
```

Wrappers: block, testcase / tcstatus, timer

The block, testcase, and timer elements act as a wrapper around a single ***task*** element and provide additional functionality to the ***task*** element.

The tcstatus element records the status of a testcase and must be contained within a testcase wrapper.

block

The **block** element is a wrapper which defines a task for which execution control is provided. The **block** element contains a single ***task*** element. It may be used in conjunction with the **hold**, **release**, and **terminate** elements and the STAX Service HOLD, RELEASE, and TERMINATE requests to define a task block that may be held, released, or terminated. Only blocks that are currently in use (e.g. running or held) can have actions (e.g. hold, release, or terminate) performed on them.

Blocks may be nested. A block named 'main' exists that wraps everything in the job. Block names must be unique within a given block scope. For nested blocks, the block name will be recorded in the hierarchical form of ParentBlockName.ChildBlockName in the STAX logs and queries (so don't use periods, ".", in your block names). The hierarchical block name must be unique with the parent block's scope.

Usage:

This example shows a block which contains a process. The block's name is the value of a variable named machName. The **block** element provides execution control (the ability to hold, release, terminate) for the **process** element contained in the **block** element.

```
<block name="machName">
  <process>
    <location>machName</location>
    <command>'P4.exe'</command>
  </process>
</block>
```

Using the following example of the **block** element, here are some comments about holding and terminating the blocks:

- You can hold Block2, then hold Block1, and then release Block2, and then release Block1.
- If you hold Block1, you cannot hold Block2 since it's already blocked from holding Block1.
- If you terminate Block2, then process P4 would be terminated and process P5 would start running.
- If you terminate Block1, then all the processes currently running in it would be terminated and process P3 would start running.

```
<sequence>

  <process>
    <location>machName</location>
    <command>'P1.exe'</command>
  </process>
  <block name="'Block1'">

    <parallel>

      <process>
        <location>machName</location>
        <command>'P2.exe'</command>
      </process>

      <sequence>

        <block name="'Block2'">

          <process>
            <location>machName</location>
            <command>'P4.exe'</command>
          </process>

        </block>

        <process>
          <location>machName</location>
          <command>'P5.exe'</command>
        </process>

      </sequence>

    </parallel>

  </block>

  <process>
    <location>machName</location>
    <command>'P3.exe'</command>
  </process>

</sequence>
```

testcase and tcstatus

The testcase element is a wrapper which defines a testcase and contains a single ***task*** element for which testcase status is recorded. The **testcase** element is used in conjunction with the **tcstatus** element to increment testcase pass/fail counters.

A **testcase** element has the following attributes:

- **name** - the name of the testcase. It is required and its value must evaluate via Python to a string.
- **mode** - the mode of the testcase. This attribute defines whether information about the testcase should be reported even if no tcstatus elements have been encountered. It is optional and its value must evaluate via Python to a string. The default value is "default", and indicates that information about the testcase should only be reported if a tcstatus element has been encountered. If the attribute's value is "strict", then information about the testcase will be reported even if no tcstatus elements have been encountered (in this case, pass and fail for the testcase will be 0).

A **tcstatus** element has the following attribute:

- **result** - indicates whether the current test passed or failed. It is required and its value must evaluate via Python to either 'pass' or 'fail'.

You may also specify additional information about the testcase status in the **tcstatus** element. It is optional and its value must evaluate via Python to a string.

You may nest **testcase** elements. For nested testcases, the testcase name will be recorded in the form of ParentTestcase.ChildTestcase in the STAX logs and queries (so don't use periods, ".", in your testcase names).

Testcase status information can be displayed in the "Testcase Information" section of the STAX Monitor GUI. A summary of the number of passes and fails for each testcase as well as any additional information specified about the status of a testcase is logged in the STAX Job Log.

Usage:

In the following example of a **testcase** element, the testcase consists of a process that is run 10 times. Each time the process runs successfully, the test status pass counter is incremented, otherwise the test status fail counter is incremented and additional information about the error is recorded (and automatically sent to the STAX Monitor and logged in the STAX job log).

```
<testcase name="'TestA'">
  <loop var="i" from="1" to="10">
    <sequence>
      <process>
        ...
      </process>
      <if expr="RC == 0">
        <tcstatus result="'pass'"/>
        <else>
          <tcstatus result="'fail'">'RC=%s on loop %s' % (RC, i)</tcstatus>
        </else>
      </if>
    <sequence>
  </loop>
</testcase>
```

In the following example of a **testcase** element, mode 'strict' is used so that an entry for each testcase will be logged, even if no <tcstatus> elements were executed. For example, an entry for testcase 'Test1' is logged in the STAX job log and sent to the STAX Monitor with 0 passes and 0 fails even if no <tcstatus> elements were executed within it.

```
  <function name="Main" scope="local">
    <testcase name="'Test1'" mode="'strict'">
      <paralleliterate var="machine" in="machList">
        <testcase name="machine" mode="'strict'">
          <sequence>
            <process>
              <location>machine</location>
              <command>'test1.exe'</command>
            </process>
            <if expr="RC == 0">
              <tcstatus result="'pass'"/>
              <else>
                <tcstatus result="'fail'">'Failed with RC=%s' % RC</tcstatus>
              </else>
            </if>
           </sequence>
        </testcase>
      </paralleliterate>
    </testcase>
  </function>
```

timer

The **timer** element is a wrapper which defines a task for which time control is provided. The **timer** element contains a single ***task*** element and runs the task for a specified duration, stopping the task at the end of the specified duration (if the task is still running).

The **timer** element has the following attribute:

- **duration -** specifies the maximum length of time to run the task. It is required. The time may be expressed in milliseconds, seconds, minutes, hours, days, weeks, or years. For example:
  - duration="50" specifies 50 milliseconds.
  - duration="90s" specifies 90 seconds.
  - duration="5m" specifies 5 minutes.
  - duration="36h" specifies 36 hours.
  - duration="3d" specifies 3 days.
  - duration="1w" specifies 1 week.
  - duration="1y" specifies 1 year.

Also, the following variable is set by the STAX execution engine upon completion of a **timer** element:

- **RC -** specifies the timer's return code
  - Set to 1 if the task is still running at the end of the specified duration.
  - Set to 0 if the task ended before the specified duration.
  - Set to -1 if the task could not begin, possibly due to an invalid value for the duration attribute.

Usage:

The following example of a **timer** element simultaneously calls function P3 in a continuous loop on a list of machines. A loop is not complete until function P3 has been run on all of the machines. After 24 hours, the test is stopped. The test is successful if it did not end before 24 hours.

```
<testcase name="'TestP3'">

  <sequence>

    <script>timerDuration = '24h'</script>

    <timer duration="timerDuration">
      <loop>
        <paralleliterate var="machName" in="MachList">
          <call function="'P3'"/>
        </paralleliterate>
      </loop>
    </timer>

    <if expr="RC == 1">
```

```
      <tcstatus result="'pass'">
        'Timer ran for %s' % timerDuration
      </tcstatus>
      <else>
        <tcstatus result="'fail'">
          'Timer did not run for %s. RC=%s' % (timerDuration, RC)
        </tcstatus>
      </else>
    </if>

  </sequence>

</testcase>
```

The following example of a **timer** element is like the previous example, but it also uses Python to calculate the elapsed time that the timer element ran so that it can record the elapsed time in the testcase status message.

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

  <sequence>

    <script>
      timerDuration = '45m'
      import time
      starttime = time.time(); # record starting time
    </script>

    <timer duration="timerDuration">
      <loop>
        <paralleliterate var="machName" in="MachList">
          <call function="'aProcess'"/>
        </paralleliterate>
      </loop>
    </timer>

    <script>
      stoptime = time.time()             # record ending time
      elapsedSecs = stoptime - starttime # yields time elapsed in seconds
    </script>

    <if expr="RC == 1">
      <tcstatus result="'pass'">
        'Timer ran for %s seconds' % elapsedSecs
      </tcstatus>
      <else>
        <tcstatus result="'fail'">
          'Timer only ran for %s seconds. RC=%s' % (elapsedSecs, RC)
        </tcstatus>
      </else>
    </if>

  </sequence>

</testcase>
```

Directives: hold, release, terminate

The hold, release, and terminate elements specify execution control during the execution of a job.

A signal is raised if you try to hold, release, or terminate a block that doesn't exist. If you try to hold a block that isn't RUNNING or release a block that isn't HELD, the request is silently ignored.

hold

The **hold** element specifies to hold a block in the STAX job. To hold the entire job, specify the "main" block. If no block is specified, the default is the current block.

release

The **release** element specifies to release a block in the STAX job. If no block is specified, the default is the current block.

terminate

The **terminate** element specifies to terminate a block in the STAX job. To terminate the entire job, specify the "main" block. If no block is specified, the default is the current block.

Usage:

In the following example of the **hold** and **terminate** elements:

- If process P1 ended with a non-zero return code, then the job would be terminated.
- If process P2 ended with a non-zero return code, then Block1 would be held.
- If process P4 ended with a non-zero return code, then Block1 would be terminated and processing would continue at process P5.

```
<sequence>

  <process>
    <location>machName</location>
    <command>'P1.exe'</command>
  </process>

  <if expr="RC != 0">
    <terminate block="'main'"/>
  </if>

  <block name="'Block1'">

    <parallel>

      <sequence>

        <process>
```

```
          <location>machName</location>
          <command>'P2.exe'</command>
        </process>

        <if expr="RC != 0">
          <hold/>
        </if>

      </sequence>

      <sequence>

        <process>
          <location>machName</location>
          <command>'P4.exe'</command>
        </process>

        <if expr="RC != 0">
          <terminate block="'main.Block1'"/>
        </if>

      </sequence>

    </parallel>

  </block>

  <process>
    <location>machName</location>
    <command>'P5.exe'</command>
  </process>

</sequence>
```

Exceptions: try / catch, throw, rethrow

The try / catch, throw, and rethrow elements deal with the processing of STAX exceptions. STAX exceptions alter the flow control of the job (unlike signals).

try / catch

The **try** element allows you to perform a task and to catch exceptions that are thrown by the task. A **try** element contains a single ***task*** element and one or more **catch** elements. You may nest **try** elements.

A **catch** element performs a task when the specified exception is caught. A **catch** element has the following attributes:

- **exception** - is the name of the exception that the **catch** element handles. All exceptions that are a sub-type of this exception are caught as well. For example, a sub-type of an exception named 'STAXException' is 'STAXException.SubType1'. You may specify '...' to indicate that all exceptions are to be caught by this catch block. This attribute is required and its value must evaluate via Python to a string.
- **var -** is the name of a variable containing any additional information provided when the exception was thrown. It is a literal.

This attribute is optional.

- **typevar** - is the name of a variable containing the specific type of exception thrown. It is a literal. This attribute is optional.

The first catch block that can handle the exception will be performed. For example, if a 'STAXException.SubType1' exception is thrown and there is a <catch exception="'STAXException'"> element and a <catch exception="'STAXException.SubType1'"> element, the catch element which is listed first will handle the exception. However, note that if the <catch exception="'STAXException'"> element is listed first, the <catch exception="'STAXException.SubType1'"> block will never handle any exceptions since it is a sub-type of 'STAXException'.

If an exception is thrown but there's no catch block for it, then the job logs an error and terminates.

Usage:

The following example of the **try/catch** elements shows a **try** block containing a sequence of tasks to perform. If an exception is thrown, the **catch** block for the exception thrown is run and then processing will continue by executing the element following the end of the **try** block. If a 'Timeout.ServerStart' or 'Timeout.ClientStart' exception is thrown, the catch block for exception 'Timeout' can handle these exceptions because they are sub-types of exception 'Timeout'.

```
<try>

  <sequence>
    <call function="'CheckServerAvailability'"/>
    <if expr="RC != 0">
      <throw exception="'ServerNotAvailable'"/>
    </if>
    <call function="'StartServers'"/>
    <if expr="RC != 0">
      <throw exception="'Timeout.ServerStart'">'Server %s' % machine</throw>
    </if>
    <call function="'StartClients'"/>
    <if expr="RC != 0">
      <throw exception="'Timeout.ClientStart'"/>'Client %s' % machine</throw>
    </if>
  </sequence>

  <catch exception="'ServerNotAvailable'">
    <log>'Handler: ServerNotAvailable'</log>
  </catch>

  <catch exception="'Timeout'" typevar="exceptionType" var="eInfo">
    <log>'Handler: Timeout, eType: %s, eInfo: %s' % (exceptionType, eInfo)</log>
  </catch>

</try>
```

throw

The **throw** element specifies an exception to throw. You may also optional specify additional information when the exception is thrown.

A **throw** element has the following attribute:

- **exception** - is the name of the exception being thrown. It must evaluate via Python to a string value. This attribute is required.

Usage:

The following example of the **throw** element shows a "NotAvailableException" exception being thrown.

```
<throw exception="'NotAvailableException'"/>
```

The following example shows a "TimerFailedException" exception being thrown with additional information provided about how long the timer ran.

```
<throw exception="'TimerFailedException'">
  'Only ran for %s seconds' % elapsedTime
</throw>
```

rethrow

The **rethrow** element specifies to rethrow the exception up the chain to a higher level try/catch block. The **rethrow** element should only be used in a **catch** block.

A **rethrow** element has no attributes.

Usage:

The following is an example of the **rethrow** element.

```
<rethrow/>
```

The following example of the **try/catch**, **throw**, and **rethrow** elements shows three nested try blocks.

If the return code after function 'CheckServerAvailability' is called is not zero, exception 'STAXException.ServerNotAvailable' is thrown. There is no catch block for this exception in this try block so the exception is thrown up the chain to its parent try block which has a catch block for this exception since it is a sub-type of exception 'STAXException'. The catch block throws exception 'OtherException' which is handled by its parent try block. The following messages are sent to the Job Monitor:

```
  Handler: STAXException eType: STAXException.ServerNotAvailable, eInfo: Server %s
  Handler: ..., eType: OtherException, eInfo: None
```

If the return code after function 'StartServers' is called is not zero, exception 'STAXException.Timeout.StartingServer' is thrown. There is a catch block that can handle this exception since it is a sub-type of exception 'STAXException.Timeout'. The catch block rethrows the exception up the chain to its parent try block. It has a catch block that can handle this exception since it is a sub-type of exception 'STAXException'. The catch block throw exception 'OtherException' which is handled by its parent try block. The following messages are sent to the Job Monitor:

```
  Servers are available
  Handler: STAXException.Timeout, eType: STAXException.Timeout.StartingServer, eInfo: 
Server %s
  Handler: STAXException, eType: STAXException.Timeout.StartingServer, eInfo: Server 
%s
```

```
  Handler: ..., eType: OtherException, eInfo: None
```

If the return code after function 'StartClients' is called is not zero, exception 'STAXException.Timeout.StartingClient' is thrown. There is a catch block that can handle this exception. The following messages are sent to the Job Monitor:

```
  Servers are available
  Servers are started
  Handler: STAXException.Timeout.StartingClient, eType:
STAXException.Timeout.StartingClient, eInfo: Client %s
```

If no exceptions were thrown (all the return codes were 0), the following messages are sent to the Job Monitor:

```
  Servers are available
  Servers are started
  Clients are started
  Ran the Test
```

```
<try>

  <try>

    <try>

      <sequence>

        <call function="'CheckServerAvailability'"/>
        <if expr="RC != 0">
          <throw exception="'STAXException.ServerNotAvailable'">
            'Server %s' % machine
          </throw>
        </if>

        <message>'Servers are available'</message>

        <call function="'StartServers'"/>
        <if expr="RC != 0">
          <throw exception="'STAXException.Timeout.StartingServer'">
            'Server %s' % machine
          </throw>
        </if>

        <message>'Servers are started'</message>

        <call function="'StartClients'"/>
        <if expr="RC != 0">
          <throw exception="'STAXException.Timeout.StartingClient'">
            'Client %s' % machine
          </throw>
        </if>

        <message>'Clients are started'</message>

        <call function="'RunTest'"/>
        <message>'Ran the test'</message>
      </sequence>

      <catch exception="'STAXException.Timeout.StartingClient'"
             typevar="eType" var="eInfo">
        <sequence>
          <message>
            'Handler: STAXException.Timeout.StartingClient, ' + \
            'eType: %s, eInfo: %s' % (eType, eInfo)
          </message>
        </sequence>
      </catch>

      <catch exception="'STAXException.Timeout'" typevar="eType" var="eInfo">
         <sequence>
           <message>
             'Handler: STAXException.Timeout, ' + \
             'eType: %s, eInfo: %s' % (eType, eInfo)
           </message>
           <rethrow/>
         </sequence>
      </catch>

    </try>

    <catch exception="'STAXException'" typevar="eType" var="eInfo">
      <sequence>
        <message>
          "Handler: STAXException, eType: %s, eInfo: %s" % (eType, eInfo)
        </message>
        <throw exception="'OtherException'"/>
      </sequence>
    </catch>

  </try>

  <catch exception="'...'" typevar="eType" var="eInfo">
    <message>
      "Handler: ..., eType: %s, eInfo: %s" % (eType, eInfo)
    </message>
  </catch>

</try>
```

Signals: raise, signalhandler

The raise and signalhandler elements deal with the raising and handling of STAX signals. Unlike STAX exceptions, STAX signals, by themselves, do not alter execution flow. Signal handlers provide asynchronous error handling while executing a STAX job. The STAX execution engine may also raise signals for errors such as if a variable is referenced which does not exist or if a function is called that does not exist.

The following table contains the names of signals that may be raised by the STAX execution engine, the conditions in which STAX raises these signals, and a description of the default signal handlers provided by STAX.

Signal Name	**Raised When:**	**Default Signal Handler**
STAXProcessStartError	A specified process cannot be successfully started. The process is bypassed. Information about the process that could not be started is provided in a variable named STAXProcessStartErrorMsg.	Sends a message that includes the variable named STAXProcessStartErrorMsg to the STAX Monitor and logs a message in the STAX Job Log with level 'error'.
STAXProcessStartTimeout	A specified process was not started within the ProcessTimeout parameter value for the STAX service. Information about the process that could not be started within the timeout value is provided in a variable named STAXProcessStartTimeoutMsg.	Sends a message that includes the variable named STAXStartProcessTimeoutMsg to the STAX Monitor and logs a message in the STAX Job Log with level 'error'.
STAXCommandStartError	A specified STAF Command request cannot be successfully started. The <stafcmd> request is bypassed. Information about the command that could not be started is provided in a variable named STAXCommandStartErrorMsg.	Sends a message that includes the variable named STAXCommandStartErrorMsg to the STAX Monitor, logs a message in the STAX Job Log with level 'error', and terminates the job.
STAXPythonEvaluationError	Python cannot successfully evaluate a value, expression, or statement(s). The element is bypassed. Information about the element which could not be successfully evaluated by Python is provided in a variable named STAXPythonEvalMsg.	Sends a message that includes the variable named STAXPythonEvalMsg to the STAX Monitor, logs a message in the STAX Job Log with level 'error', and terminates the job.
STAXFunctionDoesNotExist	A function is called that does not exist. The call request is bypassed. Information about the call request is provided in a variable named STAXFunctionDoesNotExistMsg.	Sends a message that includes the variable named STAXFunctionDoesNotExistMsg to the STAX Monitor, logs a message in the STAX Job Log with level 'error', and terminates the job.
STAXFunctionArgValidate	A function is called with arguments that are not valid. The call request is bypassed. Information about the call request is provided in a variable named STAXFunctionArgValidateMsg.	Sends a message that includes the variable named STAXFunctionArgValidateMsg to the STAX Monitor, logs a message in the STAX Job Log with level 'error', and terminates the job.
STAXBlockDoesNotExist	A block name referenced by a <hold>, <release>, or <terminate> element does not exist. The hold/release/terminate block request is bypassed. Information about the hold/release/terminate block request is provided in a variable named STAXBlockDoesNotExistMsg.	Sends a message that includes the variable named STAXBlockDoesNotExistMsg to the STAX Monitor and logs a message in the STAX Job Log with level 'error'.

STAXInvalidBlockName	A block with the name specified by the <block> element already exists. The <block> request is bypassed. Information about the invalid <block> request is provided in a variable named STAXInvalidBlockNameMsg. Note that this situation can easily occur if you have a block executing in parallel on multiple machines and you specify a literal block name, like name="'BlockA'", instead of one like name="'BlockA_%s' % machName" to uniquely identify each block.	Sends a message that includes the variable named STAXInvalidBlockNameMsg to the STAX Monitor, logs a message in the STAX Job Log with level 'error', and terminates the job.
STAXLogError	A <log> element is encountered but the STAF Log request failed. The <log> element is bypassed. Information about the invalid <log> element is provided in a STAX variable named STAXLogMsg.	Sends a message that includes the variable named STAXLogMsg to the STAX Monitor and logs a message in the STAX Job Log with level 'error'.
STAXTestcaseMissingError	A <tcstatus> element is encountered but there is no <testcase> wrapper element containing it. The <tcstatus> element is bypassed. Information about the invalid <tcstatus> element is provided in a STAX variable named STAXMissingTestcaseMsg.	Sends a message that includes the variable named STAXTestcaseMissingMsg to the STAX Monitor, and logs a message in the STAX Job Log with level 'error'.
STAXInvalidTcStatusResult	A <tcstatus> element is encountered with an invalid result value (not 'pass' or 'fail'). The <tcstatus> element is bypassed. Information about the invalid <tcstatus> element is provided in a STAX variable named STAXInvalidTcStatusResultMsg.	Sends a message that includes the variable named STAXInvalidTcStatusResultMsg to the STAX Monitor, and logs a message in the STAX Job Log with level 'error'.
STAXNoSuchSignalHandler	A <raise> element specifies a signal for which there is no signal handler. The <raise> element is bypassed. Information about the invalid <raise> element is provided in a STAX variable named STAXNoSuchSignalHandlerMsg.	Sends a message that includes the variable named STAXNoSuchSignalHandlerMsg to the STAX Monitor, and logs a message in the STAX Job Log with level 'error'.
STAXInvalidTimerValue	A <timer> element specifies an invalid value for its duration attribute. The element is bypassed. Information about the invalid <timer> element is provided in a variable named STAXInvalidTimerValueMsg.	Sends a message that includes the variable named STAXInvalidTimerValueMsg to the STAX Monitor, logs a message in the STAX Job Log with level 'error', and terminates the job.

STAXEmptyList	An <iterate> or <paralleliterate> element specifies a list which is empty or set to None. Information about the element which specified the empty list is provided in a STAX variable named STAXEmptyListMsg.	Sends a message that includes the variable named STAXEmptyListMsg to the STAX Monitor, and logs a message in the STAX Job Log with level 'error'.
STAXImportError	An error occurred while processing an <import> element. Information about the error is provided in a STAX variable named STAXImportErrorMsg. When a STAXImportError signal is raised, the variable STAXSignalData will be set to a list containing an error type and an error description.	Sends a message that includes the variable named STAXImportErrorMsg to the STAX Monitor, logs a message in the STAX Job Log with level 'error', and terminates the job.
STAXInvalidTestcaseMode	An invalid mode attribute was specified for a <testcase> element. The valid values for the attribute are 'default' and 'strict'.	Sends a message that includes the variable named STAXInvalidTestcaseModeMsg to the STAX Monitor and logs a message in the STAX Job Log with level 'error'.

You may override a default signalhandler by providing your own signalhandler.

raise

The **raise** element may be used to raise a specified signal. Signals may also be raised by the STAX execution engine. A signal interrupts the STAX job's normal flow of execution and allows a signal handler to take control. A signal handler is a function that is called when the corresponding signal occurs. When the signal handler returns, the STAX job continues to execute from the point in the job following where the signal was raised, assuming the signal handler did not terminate the job or block. If a signal handler is not provided for a generated signal, a STAXNoSuchSignalHandler signal is raised.

The **raise** element has a required attribute named signal which specifies the name of the signal being raised.

Usage:

The following example of the raise element raises a signal named 'NonZeroRCError' when a non-zero return code is encountered.

```
<function name="Valid-if-RC-0">
  <if expr="RC != 0">
    <raise signal="'NonZeroRCError'"/>
  </if>
</function>
```

signalhandler

The **signalhandler** element defines how to handle a specified signal. Signalhandlers are inherited from parent threads. The **signalhandler** element contains a single ***task*** element. The **signalhandler** element has a required attribute named signal which is the name of the signal that the signalhandler element handles. A **signalhandler** element can be contained in the **stax** root element as well as anywhere where a ***task*** element can be.

Usage:

The following example of the **signalhandler** element handles signals named 'NonZeroRCError' by setting the testcase status to fail and terminating the job.

```
<signalhandler signal="'NonZeroRCError'">
  <sequence>
    <tcstatus result="'fail'">'RC=%s' % RC</tcstatus>
    <terminate block="'main'"/>
  </sequence>
</signalhander>
```

Logging / Messages: log, message

The log and message elements deal with logging a message to a STAX job user log and sending a message to the STAX Monitor.

log

The **log** element may be used to log a message in the STAX job user log. The log value and log level must evaluate via Python to a string value.

The **log** element has the following optional attributes:

- **level** - is the logging level of the message to be logged. It must evaluate via Python to be one of the STAF logging levels ('fatal', 'error', 'warning', 'info', 'trace', 'trace2', 'trace3', 'debug', 'debug2', 'debug3', 'start', 'stop', 'pass', 'fail', 'status', 'user1', 'user2', 'user3', 'user4', 'user5', 'user6', 'user7', or 'user8'). It defaults to 'info'.
- **if** - is an expression that is evaluated via Python to a boolean value. If it evaluates to false, the message is not logged. It defaults to true ('1').

Refer to the "STAX Logging" section for more information on how to query a STAX job user log.

Usage:

The following example of the **log** element logs a message of level 'info' (the default log level) in the STAX job user log after successfully running a STAF command If the STAF command fails, it logs a message of level 'warning' in the STAX job user log.

```
<sequence>

  <stafcmd>
    <location>machName</location>
    <service>"misc"</service>
    <request>"version"</request>
  </stafcmd>

  <if expr="RC == STAFRC.Ok">
    <log>'Machine %s is running STAF Version %s' % (machName,STAFResult)</log>
    <else>
      <log level="'warning'">'RC=%s on %s' % (RC, machName)</log>
    </else>
  </if>

</sequence>
```

Note that instead of using the "if" element, could use the "if" attribute of the log element as follows:

```
  <log if="RC == STAFRC.Ok">'Machine %s is running STAF Version %s' % (machName,
STAFResult)</log>
  <log if="RC != STAFRC.Ok" level="'warning'">'RC=%s on %s' % (RC, machName)</log>
```

message

The **message** element specifies a message which will be sent to the STAX Monitor and displayed via its GUI. The message value must evaluate via Python to a string value.

The **message** element has the following optional attribute:

- **if** - is an expression that is evaluated via Python to a boolean value. If it evaluates to false, the message is not sent. It defaults to true ('1').

Usage:

The following example of the **message** element makes a message available to the STAX Monitor which displays it in the "Messages" section of the GUI.

```
<function name="Valid-if-RC-0">
  <if expr="RC != 0">
    <message>'RC=%s on machine %s' % (RC, machName)</message>
  </if>
</function>
```

Note that instead of using the "if" element, could use the "if" attribute of the message element as follows:

```
  <message if="RC ! 0">'RC=%s on machine %s' % (RC, machName)</message>
```

System Requirements

A STAX Service machine is where the STAX Service is installed. The STAX Service machine has the following software and hardware requirements:

Software Requirements

- STAF Version 2.6.0 or higher (but less than Version 3.0.0) must be installed on the STAX Service machine.
- The STAX Service code.

- The STAX Service is written in Java and requires Java 1.4 or higher.
- The STAX Monitor requires Java 1.3.0 or higher.
- A file system that supports long file names.
- Any operating system that is supported by STAF. See the STAF User's Guide for a list of supported operating systems.

Hardware Requirements

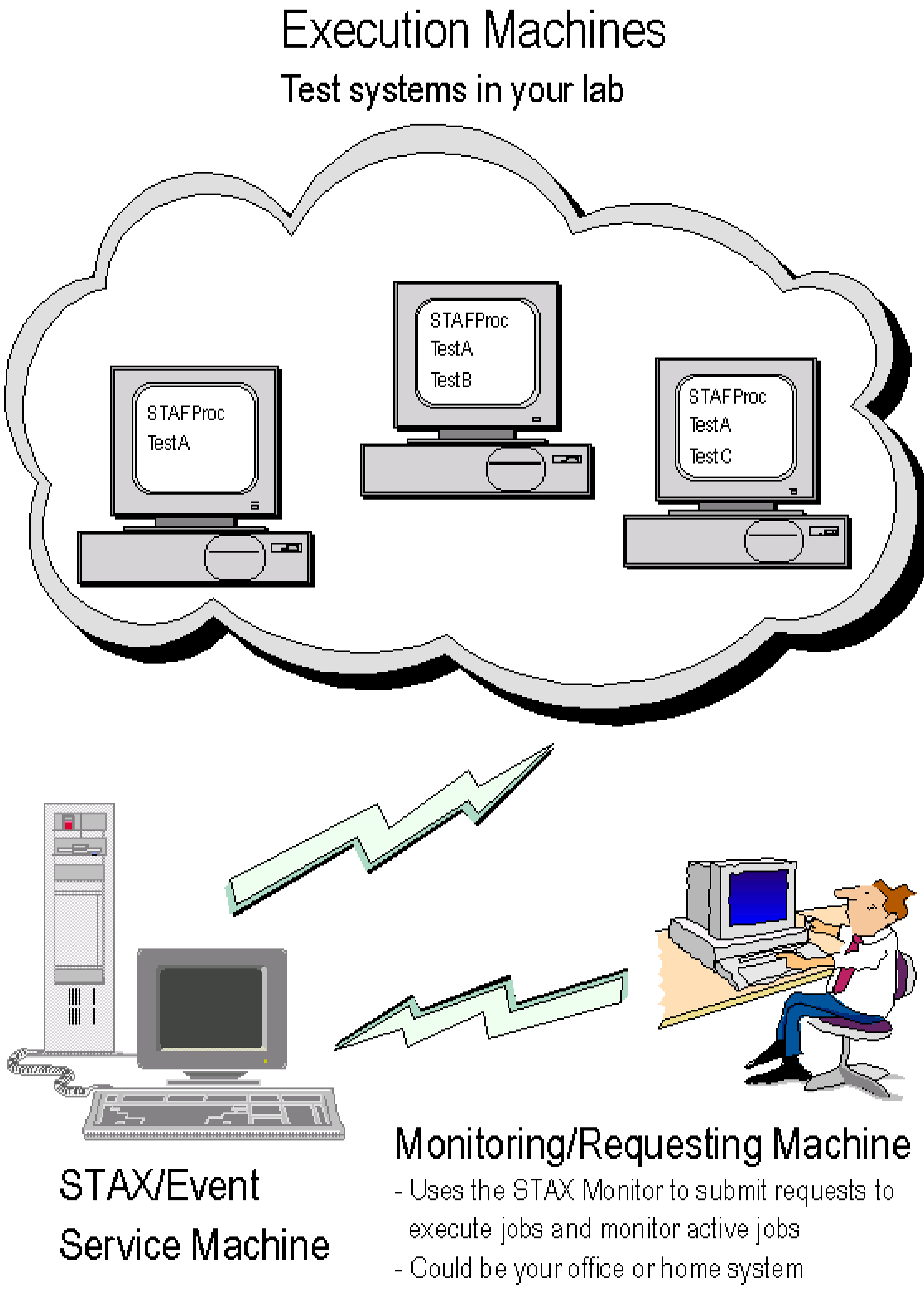
- Pentium 300 Mhz CPU minimum
- 64M memory minimum

Installation and Configuration

Following is a diagram showing a typical configuration when using STAX. Usually, you install a single server-type system to be the STAX service machine. It has STAF installed with the STAX and Event services configured. This system must be up and running STAF while STAX jobs are running on it.

You can request STAX jobs to be executed and monitor them using the STAX Monitor (or via command-line requests). This can be done from any machine, such as your office or home machine, as long as it is running STAF and has TCP/IP network capabilities such that it can communicate with the STAX service machine. Note that the same job can be monitored from multiple systems simultaneously.

The execution machines are the machines where you want tests to be run. Any number of execution machines can be involved in a single STAX job. A STAX job generally consists of processes and STAF commands that make up various testcases that run on any number of execution machines. The <location> element, which is part of each <process> and <stafcmd> element in a job, specifies the execution machine where the process or STAF command is run. You can pass additional information (e.g. a list of execution machines and/or tests) when submitting a STAX job for execution using SCRIPT/SCRIPTFILE parameters or by passing arguments to the starting function for the job.


Execution Machines
Test systems in your lab
STAFProc
TestA
STAFProc
TestA
TestB
STAFProc
TestA
TestC
STAX/Event
Service Machine
Monitoring/Requesting Machine
- Uses the STAX Monitor to submit requests to execute jobs and monitor active jobs
- Could be your office or home system

STAX Service Machine

1. Install Java 1.4 or later. The STAX Service is written in Java, so you need to install a JVM (e.g. Sun or IBM) on the STAX Service machine. IBM employees must download the Sun or IBM JVM from the internal JIM site. Non-IBM users can

download from http://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.

Verify that the PATH environment variable contains the Java bin directory (e.g. C:\j2sdk1.4.2_05\bin). You can type "java -version" to check the version of Java that's in the PATH. Or you can override the version of Java that you want the STAX service to use when registering the STAX service via the OPTION JVM=<Java Path> option when creating a new JVM (e.g. OPTION JVM=C:\j2sdk1.4.2_05\bin\java).

2. Install STAF Version 2.6.0 or higher (but less than Version 3.0.0) by following 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. Install the STAX service:

   a. Extract the STAX tar/zip file into a local directory.
      - On Windows, unzip the STAXV1511.zip file into a local directory (e.g. C:\STAF\services).
      - On Unix, untar the STAXV1511.tar file into a local directory (e.g. /usr/local/staf/services).

   b. Verify that the local directory where you extracted the STAX zip/tar file now includes the following files:
      - STAX.jar
      - STAXMon.jar
      - STAFEvent.jar file.

4. Update the STAF configuration file as follows:

   - Register the STAX service.
   - Register the Event service if the Event Service is also installed on the STAX Service system and you plan to use the STAX Monitor.
   - Register the Log service if you want STAX log files to be created by the STAX service. Note that the default STAF configuration file includes a SERVICELOADER configuration line for STAF's default service loader which can dynamically load the Log Service.
   - Increase the default setting for MAXQUEUESIZE if you plan to run the STAX Monitor on this system.

   Following is the syntax for the lines which should be present in the STAF configuration file:

```
SERVICELOADER LIBRARY STAFDSLS

SERVICE <Name> LIBRARY JSTAF EXECUTE <STAX Jar File Name>
               [OPTION <Name[=Value]>]...
               [PARMS <"> [EVENTSERVICEMACHINE <EventMachine>]
                          [EVENTSERVICENAME <EventName>]
                          [NUMTHREADS <NumThreads>]
                          [PROCESSTIMEOUT <ProcessTimeout>]
                          [CLEARLOGS <Enabled | Disabled>]
                          [LOGTCELAPSEDTIME <Enabled | Disabled>]
                          [LOGTCNUMSTARTS <Enabled | Disabled>]
                          [LOGTCSTARTSTOP <Enabled | Disabled>]
                          [EXTENSIONXMLFILE <Extension XML File> |
                           EXTENSIONFILE <Extension Text File>]
                          [EXTENSION <Extension Jar File>...
```

```
                         <">]

SERVICE <Name> LIBRARY JSTAF EXECUTE <Event Jar File Name>
               [OPTION <Name[=Value]>]...

SET MAXQUEUESIZE 10000
```

where:

- `<Name>` is the name by which the service will be known on this machine. The name of the STAX Service should generally be STAX and the name of the Event Service should generally be Event.

- `<STAX Jar File Name>` is the fully qualified name of the STAX.jar file. On Windows systems, this might be C:\STAF\services\STAX.jar. On Unix systems, this might be /usr/local/staf/services/STAX.jar.

- `<Event Jar File Name>` is the fully qualified name of the STAFEvent.jar file. On Windows systems, this might be C:\STAF\services\STAFEvent.jar. On Unix systems, this might be /usr/local/staf/services/STAFEvent.jar

- `OPTION` specifes a configuration option that will be passed on to the JSTAF Java service proxy library. This is typically used by service proxy libraries to further control the interface to the actual service implementation. You may specify multiple `OPTION`s for a given service. See the STAF User's Guide for more information on options for the JSTAF Java service proxy library.

  Note that if you run long, resource-intensive STAX jobs, you may need to increase the minimum and maximum JVM heap sizes for the STAX Service to the maximum that the system can support based on its physical memory and the memory usage required by other applications running on the system.

- `<EventMachine>` is the name of the Event service machine. This option will resolve STAF variables.

- `<EventName>` is the name by which the Event service will be known to the STAX service. It defaults to "Event" (not case-sensitive) if not specified. This option will resolve STAF variables.

- `<NumThreads>` is the number of physical threads that the STAX Service will use. It must be an integer value of 2 or greater. It defaults to 5 if not specified. This option will resolve STAF variables.

- `<ProcessTimeout>` is the number of milliseconds that the STAX service will wait for processes to start. It must be an integer value of 1000 (1 second) or greater. It's value should be at least the value of the CONNECTTIMEOUT operational parameter set for STAF on the STAX service machine. The default is 60000 milliseconds (1 minute). If a process does not start within the timeout value, a STAXProcessStartTimeout signal will be raised and the job will continue.

- `CLEARLOGS` specifies whether the STAX Job and Job User logs should be deleted before a job is executed. Since STAX job numbers are reused, a specific job log may contain the results for more than one job. Valid values are "Enabled" and "Disabled", not case-sensitive. If you specify "Enabled", the log files will be deleted before a job is executed, in order to ensure that only one job's contents are in the logs. If you specify "Disabled", the job logs will not be deleted. The default is "Disabled". This option will resolve STAF variables.

- `LOGTCELAPSEDTIME` specifies whether to log the elapsed time in the summary "Status" record for each testcase written at the end of the STAX Job log and when you list testcases. Valid values are "Enabled" and "Disabled", not case-sensitive. If you specify "Enabled", the elapsed time will be logged. If you specify "Disabled", the elapsed time will not be logged. The default is "Enabled". This option will resolve STAF variables.

- `LOGTCNUMSTARTS` specifies whether to log the number of starts in the summary "Status" record for each testcase written at the end of the STAX Job log and when you list testcases. Valid values are "Enabled" and "Disabled", not case-sensitive. If you specify "Enabled", the number of starts will be logged. If you specify "Disabled", the number of starts will not be logged. The default is "Enabled". This option will resolve STAF variables.

- `LOGTCSTARTSTOP` specifies whether to log a "Start" record each time a testcase begins and to log a "Stop" record each time a testcase ends in the STAX Job log. Valid values are "Enabled" and "Disabled", not case-sensitive. If you specify "Enabled", the start/stop testcase records will be logged. If you specify "Disabled", the start/stop testcase records will not be logged. The default is "Disabled". This option will resolve STAF variables.

- `EXTENSIONXMLFILE` specifies the fully-qualified name of an XML file that contains the names of the extension jar files to be registered. Refer to the "STAX Extensions" section for more information.

- `EXTENSION` specifies the fully-qualified name of an extension jar file. Refer to the "STAX Extensions" section for more information.

- `EXTENSIONFILE` specifies the fully-qualified name of a text file that contains the names of the extension jar files to be registered. Refer to the "STAX Extensions" section for more information.

Notes:

1. Each `SERVICE` configuration statement must be entered as one continuous line (or you can use a backslash (\) to specify line continuation.

2. If the Event Service machine is the same as the STAX service machine and "Event" is the name used when registering the Event Service, then you don't have to specify the `EVENTSERVICEMACHINE` or the `EVENTSERVICENAME` parameters. The <EventName> defaults to "Event" if it is not specified and the <EventMachine> defaults to the STAX service machine name if it is not specified.

3. The STAX Service requires version 1.3.1 or later (but less then 3.0.0) of the Event service.

4. The `CLEARSLOGS`, `LOGTCELAPSEDTIME`, `LOGTCNUMSTARTS`, and `LOGTCSTARTSTOP` parameters may be changed usingthe SET command, Refer to the "SET" section for more information.

Examples of Service Configuration Lines for the STAX Service System:

- If the STAX and Event services are installed on the same system, the `SERVICE` configuration lines in the STAF configuration file might look like the following:

  Windows (assuming the STAX.jar and STAFEvent.jar files are in C:\STAF\services):

  ```
  SERVICE STAX LIBRARY JSTAF EXECUTE C:\STAF\services\STAX.jar
  SERVICE EVENT LIBRARY JSTAF EXECUTE C:\STAF\services\STAFEvent.jar
  ```

  Unix (assuming the STAX.jar and STAFEvent.jar files are in /usr/local/staf/services):

  ```
  SERVICE STAX LIBRARY JSTAF EXECUTE /usr/local/staf/services/STAX.jar
  SERVICE EVENT LIBRARY JSTAF EXECUTE /usr/local/staf/services/STAFEvent.jar
  ```

- If the Event service is installed on a different system (e.g. machA.austin.ibm.com) and is named "Event", the `SERVICE` configuration lines in the STAF configuration file might look like the following:

```
SERVICE STAX LIBRARY JSTAF EXECUTE C:\STAF\services\STAX.jar \
             PARMS "EVENTSERVICEMACHINE machA.austin.ibm.com"
```

Note that since each `SERVICE` configuration statement must be entered as one continuous line, a backslash (\) can be used at the end of the prior line to indicate you're continuing on the next line.

- If you want the STAX service to use 8 physical threads and you want to enable start/stop testcase logging and the STAX and Event services are installed on the same system, the `SERVICE` configuration lines in the STAF configuration file might look like the following:

```
SERVICE STAX LIBRARY JSTAF EXECUTE C:\STAF\services\STAX.jar \
             PARMS "NUMTHREADS 8 LOGTCSTARTSTOP Enabled"
SERVICE EVENT LIBRARY JSTAF EXECUTE C:\STAF\services\STAFEvent.jar
```

- If you want the STAX service to run in a separate JVM named STAX and you want the JVM to use 256M as the maximum memory allocation size, and the STAX and Event services are installed on the same system, the `SERVICE` configuration lines in the STAF configuration file might look like the following:

```
SERVICE STAX LIBRARY JSTAF EXECUTE C:\STAF\services\STAX.jar \
             OPTION JVMName=STAX OPTION J2=-Xmx256m
SERVICE EVENT LIBRARY JSTAF EXECUTE C:\STAF\services\STAFEvent.jar
```

Note that the -X Java options are non-standard and can vary by Java version. Use the "java -X" command to display the non-standard options available for the Java you are using.

- If you want the STAX and Event services to run in a separate JVM named STAX using the Java executable located in C:\j2sdk1.4.2_05\bin and you want the JVM to use 512M as the maximum memory allocation size, and the STAX and Event services are installed on the same system, the `SERVICE` configuration lines in the STAF configuration file might look like the following:

```
SERVICE STAX LIBRARY JSTAF EXECUTE C:\STAF\services\STAX.jar \
             OPTION JVMName=STAX OPTION JVM=C:\j2sdk1.4.2_05\bin\java OPTION J2=-
Xmx512m
SERVICE EVENT LIBRARY JSTAF EXECUTE C:\STAF\services\STAFEvent.jar OPTION
JVMName=STAX
```

Requesting Machine

Requesting machines are the ones that submit requests (e.g. EXECUTE, QUERY) to the STAX Service. Refer to the "Request Syntax" section for request details.

1. Install STAF by following the installation instructions in the STAF documentation.

2. A requesting machine must give a trust level of at least 4 to the STAX Service machine. The syntax for the `TRUST LEVEL` line that must be in the STAF configuration file on each requesting machine is:

```
     TRUST LEVEL 4 MACHINE <STAX Service machine>
```

Execution Machine

Process execution machines have STAF processes and commands executed on them as defined by the STAX XML file.

1. Install STAF by following the installation instructions in the STAF documentation.

2. A process execution machine must give a trust level of 5 to the STAX Service machine. The syntax for the `TRUST LEVEL` line that must be in the STAF configuration file on each execution machine is:

   ```
   TRUST LEVEL 5 MACHINE <STAX Service machine>
   ```

Monitoring Machine

STAX Monitoring machines can monitor the STAX jobs in progress.

1. Install Java 1.3 or later.

   The STAX Monitor is a Java application, so you need to install a JVM (e.g. Sun or IBM) on the STAX Monitoring machine(s). IBM employees must download the Sun or IBM JVM from the internal JIM site. Non-IBM users can download from http://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.

   Verify that the PATH environment variable contains the Java bin directory (e.g. C:\j2sdk1.4.2_05\bin). You can type "java -version" to check the version of Java that's in the PATH.

2. Install STAF 2.1 or later (but less than STAF 3.0.0) by following 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. Install the STAX Monitor by copying the STAXMon.jar file onto the system. Note that the STAXMon.jar file is obtained by extracting the STAX zip/tar file as was already done on the STAX Service machine. We highly recommend copying it into a first or second level directory, such as services or services/STAX, off of your STAF root directory (e.g. C:\STAF\services or C:\STAF\services\STAX on Windows, /usr/local/staf/services or /usr/local/staf/services/STAX on Unix) or to make starting the STAX Monitor easier.

4. A STAX Monitoring machine must give a trust level of at least 3 to the STAX Service machine. The syntax for the line that must be in the STAF configuration file on each STAX Monitoring machine is:

   ```
   TRUST LEVEL 3 MACHINE <STAX Service machine>
   ```

5. A STAX Monitoring machine must increase its maximum queue size to 10000. The syntax for the line that must be in the STAF configuration file on each STAX Monitoring machine is:

   ```
   SET MAXQUEUESIZE 10000
   ```

Request Syntax

The STAX service provides the following requests:

- EXECUTE - Starts a job running based on a XML document which defines the workflow for a job.
- GET - Gets the STAX DTD (Document Type Definition).
- HELP - Displays a list of requests for the STAX service and how to use them.
- HOLD - Holds a job or a block in a job that is currently running.
- LIST - Lists all currently running jobs or specific information about a single job or the settings for the STAX service.
- QUERY - Queries information about a job that is currently running.
- RELEASE - Releases a job or a block in a job that is in the hold state.
- SET - Sets STAX service options.
- START - Starts a new testcase in a job that is currently running.
- STOP - Stops a testcase in a job that is currently running.
- TERMINATE - Terminates a job or a block in a job.
- UPDATE - Updates the status of a testcase in a job that is currently running.
- VERSION - Displays the version of the STAX service.

EXECUTE

EXECUTE starts the execution of a job based on an input XML document which defines the workflow for a job.

Syntax

```
EXECUTE < <FILE <xml file name> [MACHINE <machine name>]> | DATA <xml data> >
        [JOBNAME <job name>] [FUNCTION <function name>] [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>]
```

`FILE` specifies the fully qualified name of a file containing the XML document for the job to be executed. This option will resolve STAF variables.

`MACHINE` specifies the name of the machine where the `FILE` is located. If not specified, it assumes the file is on the machine submitting the STAX EXECUTE request. This option will resolve STAF variables.

`DATA` specifies a string containing the XML document for the job to be executed.

`JOBNAME` specifies the name of the job. This can aid in the identification of a specific job. The job name defaults to the value of the function name called to start the job. This option will resolve STAF variables.

`FUNCTION` specifies the name of the function element to call to start the job, overriding the defaultcall element, if any, specified in the XML document. The <function name> must be an name of a function element specified in the XML document. This option will resolve STAF variables.

`ARGS` specifies arguments to pass to the function element called to start the job, overriding the arguments, if any, specified for the defaultcall element in the XML document. This option will not resolve STAF variables.

`SCRIPTFILE` specifies the fully qualified name of a file containing Python code to be executed. This is like a <script> element in root <stax> element in the XML document, but defined when submitting an execute request, rather than within the XML document. Note that a SCRIPTFILE parameter specified in an execute request will be executed by the STAX service after any "global" <script> elements (that is, those contained directly within the root <stax> element), but before any SCRIPT parameters are executed. Thus, you can override the value of a variable specified in a global <script> element by using a SCRIPTFILE parameter in the execute request. You may specify as many SCRIPTFILE parameters as needed. This option will resolve STAF variables.

`SCRIPTFILEMACHINE` specifies the name of the machine where the `SCRIPTFILE(s)` are located. If not specified, it defaults to the value specified for MACHINE. If a MACHINE option was not specified, it assumes the script file(s) are on the machine submitting the STAX EXECUTE request. This option will resolve STAF variables.

`SCRIPT` defines Python code to be executed. This is like a <script> element in root <stax> element in the XML document, but defined when submitting an execute request, rather than within the XML document. Note that a script parameter specified in an execute request will be executed by the STAX service after any "global" <script> elements (that is, those contained directly within the root <stax> element) and after any SCRIPTFILE parameters are executes, but before the starting function is called. Thus, you can override the value of a variable specified in a global <script> element or in a SCRIPTFILE by using a script parameter in the execute request. You may specify as many SCRIPT parameters as needed.

`CLEARLOGS` is used to indicate that the STAX Job and Job User logs should be deleted before the job is executed. Since STAX job numbers are reused, a specific job log may contain the results of more than one job. Valid values are "Enabled" and "Disabled", not case-sensitive. If you specify "Enabled" or no value, the log files will be deleted for the job that is about to execute, in order to ensure that only one job's contents are in the log. If you specify "Disabled", the job logs will not be deleted. This overrides the service setting for "Clear Logs". This option will resolve STAF variables.

`HOLD` is used to hold the job after its been successfully parsed and the job id has been returned. This allows you to start the STAX Monitor application and then release the job so that you can monitor the job from its beginning if desired.

`TEST` is used to test whether the execution options specified are valid, if an XML document is well-formed and valid, and if the Python code specified in the XML document compiles successfully. When this option is specified, execution of the job is not started.

`WAIT` is used to specify that the STAX EXECUTE request should not return until the STAX job has completed. If a Timeout value is specified, the WAIT will timeout after the specified number of milliseconds.

`RETURNRESULT` is used to specify that you want the job result returned. The job result is the result returned by the starting function of the job in a "string" format. This option is only valid if the `WAIT` option is specified.

`LOGTCELAPSEDTIME` is used to specify whether to log the elapsed time in the summary "Status" record for each testcase written at the end of the STAX Job log and when you list testcases. Valid values are "Enabled" and "Disabled", not case-sensitive. This option overrides the service setting for "Log TC Elapsed Time". This option will resolve STAF variables.

`LOGNUMSTARTS` is used to specify whether to log the number of starts in the summary "Status" record for each testcase written at the end of the STAX Job log and when you list testcases. Valid values are "Enabled" and "Disabled", not case-sensitive. This option overrides the service setting for "Log TC Num Starts". This option will resolve STAF variables.

`LOGTCSTARTSTOP` is used to specify whether to log a "Start" record each time a testcase begins and to log a "Stop" record each time a testcase ends in the STAX Job log. Valid values are "Enabled" and "Disabled", not case-sensitive. This option overrides the service setting for "Log TC Start/Stop". This option will resolve STAF variables.

Security

This request requires at least trust level 4.

Results

- Upon successful return, if the TEST or RETURNRESULT options are not specified, the result buffer contains the Job ID.
- Upon successful return, if the TEST option is specified, the result buffer contains nothing.
- If the WAIT option is specified with a Timeout, and the request times out before the Job has completed, a RC 37 (Timeout) is returned and the result buffer contains the Job ID.
- If the WAIT and RETURNRESULT options are specified, and the request does not timeout before the Job has completed, the result buffer contains the Job ID and a "string" version of the Job Result, as follows:

  ```
  <Job ID>;<Job Result>
  ```

  For example, if a job's starting function returned 0 (to indicate success), the result buffer would contain

  ```
  5;0
  ```

  (assuming the Job ID is 5 and the wait did not timeout).

  If nothing is returned by a job, the job result is None and the result buffer would contain

  ```
  1;None
  ```

  (assuming the Job ID is 1 and the wait did not timeout). The job may not return anything if a <return> element is not executed in the starting function. This could be due to many reasons, such as an error occurring, the job being terminated, or if the starting function doesn't contain a <return> element.

Examples

Goal: Execute a job defined by an XML file named d:\stax\xml\JobA.xml and give it a job name of JobA.

`EXECUTE FILE D:\stax\xml\JobA.xml JOBNAME JobA` Goal: Test if a XML job file named d:\stax\xml\JobA.xml is valid without actually starting the execution of the job.

```
EXECUTE FILE D:\stax\xml\JobA.xml TEST
```

Goal: Execute a job defined by an XML file named d:\stax\xml\Ogre.xml which is located on machine MachA and assign a literal string 'OgreSrv1' to a Python variable named S1.

```
EXECUTE FILE C:\stax\Ogre.xml MACHINE MachA SCRIPT "S1 = 'OgreSrv1'"
```

Goal: Execute a job defined by an XML file named /tests/test1.xml which is located on machine MachA and execute Python code contained in file /tests/init1.py located on machine MachB.

```
EXECUTE FILE /tests/test1.xml MACHINE MachA SCRIPTFILE /tests/init1.py SCRIPTFILEMACHINE MachB
```

Goal: Execute a job defined by an XML file named d:\stax\xml\ProcessXYZ.xml, starting at ProcessY, and hold the job so that it can be monitored from the beginning.

```
EXECUTE FILE D:\stax\xml\ProcessXYZ.xml JOBNAME JobXYZ FUNCTION ProcessY HOLD
```

Goal: Execute a job defined by an XML file named /test/staxtest.xml located on the local machine, starting at function Main, and pass a list of tests as the arguments to the function.

```
EXECUTE FILE /test/staxtest.xml FUNCTION Main ARGS "['test1', 'test2', 'test3']"
```

Goal: Execute a job defined by an XML file named /test/sample.xml located on the local machine, starting at function InitJob, and pass a map that contains a list of machines and a duration as the arguments to the function.

```
EXECUTE FILE /test/sample.xml FUNCTION InitJob ARGS "{ 'MachList': ['machA', 'machB'], 'duration':
'5m' }"
```

Goal: Execute a job defined by an XML file named d:\stax\xml\JobA.xml and wait indefinitely for the job to complete before returning and return the job result in addition to the job ID in the result buffer.

```
EXECUTE FILE D:\stax\xml\JobA.xml WAIT RETURNRESULT
```

Goal: Execute a job defined by an XML file named d:\stax\xml\JobA.xml and wait for up to 1 hour for the job to complete before returning.

```
EXECUTE FILE D:\stax\xml\JobA.xml WAIT 3600000
```

Goal: Execute a job defined by an XML file named /tests/Scenario1.xml and to clear its job logs before execution and to enable "Log TC Elapsed Time", "Log TC Num Starts", and "Log TC Start/Stops".

```
EXECUTE FILE /tests/Scenario1.xml CLEARLOGS Enabled LOGTCELAPSEDTIME Enabled LOGTCNUMSTARTS Enabled
LOGTCSTARTSTOP Enabled
```

GET

GET DTD displays the DTD (Document Type Definition) for the STAX Service. An XML document executed by the STAX Service is considered valid if the document complies with this DTD. Note that DTDs are all about specifying the structure and syntax of XML documents (not their content).

Syntax

```
GET DTD
```

Security

This request requires at least trust level 2.

Results

The result buffer contains the DTD for the STAX service.

HELP

Help displays the request options and how to use them.

Syntax

```
HELP
```

Results

The result buffer contains the Help messages for the request options for the STAX service.

HOLD

HOLD allows you to hold a job or a currently running block in a job.

Syntax

```
HOLD JOB <JobID> [BLOCK <Block Name>]
```

`JOB` specifies the ID of the job to hold. If no block is specified, then the "main" block in the job (which, by default, encompasses the entire job) is held which means all processes and STAF commands currently running are held until the job is released using a STAX RELEASE JOB <JobID> request.

`BLOCK` specifies a particular block in the job to hold. The block name must correspond to a block element name specified in the XML document. If a block in a job is held, all of the processes and STAF commands currently running within that block are held until that block is released using a STAX RELEASE JOB <JobID> BLOCK <Block Name> request.

A child block cannot be held if it is being held by a parent block.

Security

This request requires at least trust level 4.

Results

Upon successful return, the result buffer does not contain anything.

Examples

Goal: Hold all of job 5.

```
  HOLD JOB 5
```

Goal: Hold block MachineB in job 31.

```
  HOLD JOB 31 BLOCK MachineB
```

LIST

LIST allows you to list all of the jobs currently running or to list information about a specified job's threads, blocks, testcases, processes, or STAF commands. LIST also allows you to list the current operational settings for the service or to list extensions registered for the service, by element name or by extension jar file name.

Syntax

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

`JOBS` lists the jobs that are currently running.

`SETTINGS` lists the current operational settings for the service.

`EXTENSIONS` lists the extension elements registered for the service in alphbetical order.

`EXTENSIONJARFILES` lists the names of the extension jar files registered for the service.

`JOB` specifies the Job ID for a currently running job.

`THREADS` lists the threads that are currently running for the specified job.

`BLOCKS` lists the blocks that are currently running for the specified job.

`TESTCASES` lists the testcases that have had test status (number of passes and fails) recorded for the specified job.

`PROCESSES` lists the processes that are currently running for the specified job.

`STAFCMDS` lists the STAF commands that are currently running for the specified job.

`SUBJOBS` lists the sub-jobs that are currently running for the specified job (initiated via the <job> element).

`<List Type>` lists the extensions of the specified type that are currently running for the specified job.

Security

This request requires at least trust level 2.

Results

Upon successful return, the result buffer will contain information about the request in the following format:

`LIST JOBS`

```
<Job ID>;<Job Name>;<Function>;<Function Arguments>
```

The <Job Name> value is set to <N/A> if a job name was not specified for the job.

The <Function> value is set to the "function" parameter specified on the EXECUTE request or if one wasn't specified, it is set to the defaultcall element's function value.

The <Function Arguments> value is set to None if no arguments were specified.

LIST SETTINGS

```
Event Machine      : <Event Machine>
Event Service Name : <Event Service Name>
Number of Threads  : <Num Threads>
Process Timeout    : <Process Timeout>
Clear Logs         : <Enabled | Disabled>
Log TC Elapsed Time: <Enabled | Disabled>
Log TC Num Starts  : <Enabled | Disabled>
Log TC Start/Stop  : <Enabled | Disabled>
```

<Event Machine> is the name of the Event service machine used by the STAX Service.

<Event Service Name> is the name by which the Event service is known to the STAX service.

<Num Threads> is the number of physical threads used by the STAX Service.

<Process Timeout> is the number of milliseconds that the STAX service waits for processes to start.

LIST EXTENSIONS

```
<Extension Element>;<Extension Jar File>
```

`<Extension Element>` is the name of each extension element registered for the STAX Service (enclosed in < and >).

`<Extension Jar File>` is the name of the extension jar file which contains the extension.

A line for each extension element registered is displayed.

LIST EXTENSIONJARFILES

```
<Extension Jar File>;<Version>;<Description>
```

`<Extension Jar File>` is the name of the extension jar file which contains the extension.

`<Version>` is the version of the extensions provided in the extension jar file. It is obtained from extension jar file's manifest via the "Extension-Version" attribute. If the version is not provided in the manifest, <Not Provided> is displayed.

`<Description>` is the description of the extensions provided in the extension jar file. It is obtained from the extension jar file's manifest via the "Extension-Description" attribute. If the description is not provided in the manifest, <Not Provided> is displayed.

A line for each extension jar file registered is displayed.

LIST JOB <Job ID> THREADS

```
<Thread ID>;<Thread State>
```

A line for each thread currently running for the specified Job is displayed. If a thread has a parent thread, it will be indented with

".." to indicate its parent thread hierarchy.

LIST JOB <Job ID> BLOCKS

```
<Block Name>;<Status>
```

A line for each block that is currently in a running or held state for the specified Job is displayed. The block names are displayed in a ParentBlock.ChildBlock format. Note that every job contains a default "main" block which includes the entire job.

LIST JOB <Job ID> TESTCASES

```
<Testcase Name>;<NumPasses>;<NumFails>[;<ElapsedTime>][;<NumStarts>]
```

A line for each testcase that has had a testcase status recorded for the specified Job is displayed.

<Testcase Name> is the testcase name in a ParentTestcase.ChildTestcase format.

<NumPasses> is the total number of passes for a testcase

<NumFails> is the total number of fails for a testcase

<ElapsedTime> is the committed elapsed time for a testcase. It is provided if "Log TC Elapsed Time" is enabled for the job. The format is HH:MM:SS (or HHH:MM:SS if over 99 hours) or <Pending> if the testcase has not yet been stopped in the job via a </testcase> or STOP request.

<NumStarts> is the number of times the testcase has been started in the job via a <testcase> element and/or a START request. It is provided if "Log TC Num Starts" is enabled for the job.

LIST JOB <Job ID> PROCESSES

```
<Name>;<Location>;<Handle>;<Command>;<Parms>
```

A line for each process that is currently running in the specified Job is displayed.

LIST JOB <Job ID> STAFCMDS

```
<Name>;<Location>;<RequestNum>;<Service>;<Request>
```

A line for each STAF command that is currently running in the specified Job is displayed.

LIST JOB <Job ID> SUBJOBS

```
<Job ID>;<Job Name>;<Block Name>;<Function>;<Function Arguments>
```

A line for each sub-job that is currently running in the specified Job is displayed. The <Job Name> value is set to <N/A> if a job name was not specified for the sub-job.

Examples

Goal: List all of the jobs.

```
  LIST JOBS
```

Output:

```
3;JobXYZ;FunctionX;None
4;OgreJob;OgreFunction;0
6;<N/A>;FunctionA;{ 'MachList': ['machA', 'machB'], 'duration': '2m' }
```

Goal: List the STAX service operational settings.

```
LIST SETTINGS
```

Output:

```
Event Machine : server1.austin.ibm.com
Event Service Name : Event
Number of Threads : 5
Process Timeout : 60000
Clear Logs : Disabled
Log TC Elapsed Time: Enabled
Log TC Num Starts : Enabled
Log TC Start/Stop : Disabled
```

Goal: List the STAX service extension elements:

```
LIST EXTENSIONS
```

Output:

```
<email>;C:/STAXExt/EmailExt.jar
<ext-delay>;C:/STAF/services/STAX/ExtDelay.jar
<ext-sleep>;C:/STAF/services/STAX/ExtDelay.jar
<ext-wait>;C:/STAF/services/STAX/ExtDelay.jar
```

Goal: List the extension jar files registered for the STAX version:

```
LIST EXTENSIONJARFILES
```

Output:

```
C:/STAXExt/ExtDelay.jar;1.0.0;Delay STAX Extensions
C:/STAXExt/ExtMessageText.jar;1.0.0;Message Text STAX Monitor Extension
C:/STAXExt/email.jar;1.0.0;eMail STAX Service Extension
```

Goal: List all of the threads running for a job with Job ID 12.

```
LIST JOB 12 THREADS
```

Output:

```
1;Blocked
..2;Blocked
....3;Blocked
....6;Blocked
..4;Blocked
....5;Blocked
```

Goal: List all of the blocks running for a job with Job ID 35.

```
LIST JOB 35 BLOCKS
```

Output:

```
main;Running
main.BlockA;Running
main.BlockA.BlockB;Held
main.BlockB;Running
```

Goal: List all of the testcases that have recorded testcase status for a job with Job ID 7 and with "Log TC Elapsed Time" and "Log TC Num Starts" enabled.

```
LIST JOB 7 TESTCASES
```

Output:

```
TestA.TestB;1;0;00:20:13;1
TestA.TestC;5;1;01:32:05;3
TestD;0;1;<Pending>;1
```

Goal: List all of the testcases that have recorded testcase status for a job with Job ID 7 and with "Log TC Elapsed Time" and "Log TC Num Starts" disabled.

```
LIST JOB 7 TESTCASES
```

Output:

```
TestA.TestB;1;0
TestA.TestC;5;1
TestD;0;1
```

Goal: List all of the processes running for a job with Job ID 41.

```
LIST JOB 41 PROCESSES
```

Output:

```
Process0;machineB;196;java;com.ibm.staf.service.stax.TestProcess 2 5 100
Process2;machineB;213;java;ICC_ProfileRGB
Process3;machineB;216;sol;
Process4;machineC;223;notepad;
```

Goal: List all of the STAF commands running for a job with Job ID 37.

```
LIST JOB 37 STAFCMDS
```

Output:

```
STAFCommand22;serverA;343;service;list services
STAFCommand23;serverA;349;echo;echo Hello World
STAFCommand24;machineC;351;service;list requests
```

Goal: List all sub-jobs for a job with Job ID 1.

```
LIST JOB 1 SUBJOBS
```

Output:

```
2;JobXYZ;Main;Function1;None
3;MyJob;Main;FunctionXYZ;0
4;<N/A>;WinNT_Block;FunctionA;{ 'MachList': ['machA', 'machB'], 'duration': '2m' }
```

QUERY

QUERY allows you to query a job, an extension jar file, or all extension jar files.

Syntax

```
QUERY JOB <Job ID> [BLOCK <Block Name> | THREAD <ThreadID> | TESTCASE <Test Name> |
                   PROCESS <Location:Handle> | STAFCMD <Request#> |
                   <Query Type> <Type Value>] |
      EXTENSIONJARFILE <Jar File Name> | EXTENSIONJARFILES
```

`JOB` specifies the ID of a job to query that is currently running. Basic information about the job is returned if `BLOCK, THREAD, TESTCASE, PROCESS, or STAFCMD` are not specified.

`BLOCK` specifies the name of a currently running block in the job to query. The request returns more detailed information about the specified block.

`THREAD` specifies a currently running Thread ID in the job to query. The request returns more detailed information about the specified thread.

`TESTCASE` specifies a testcase name in the job to query. The request returns more detailed information about the specified testcase.

`PROCESS` specifies a location and handle number (separated by a colon) that uniquely identifies a currently running process in the job to query. The request returns more detailed information about the specified process.

`STAFCMD` specifies a request number that uniquely identifies a currently running STAF command in the job to query. The request returns more detailed information about the specified STAF command.

`<Query Type>` specifies a value for the extension type that uniquely identifies a currently running extension of the specified type in the job to query.

`EXTENSIONJARFILE` specifies the name of an extension jar file to query. The request returns more detailed information about the specified extension jar file, such as what elements it provides and what version of the STAX service is required. Note that the name of the extension jar file is case sensitive and must be specified exactly as it appears in the output from LIST EXTENSIONS or LIST EXTENSIONJARFILES.

`EXTENSIONJARFILES` specifies to query all of the extension jar files registered for the STAX service. The request returns more detailed information about all of the extension jar files, such as what elements each provides and what version of the STAX service is required.

Security

This request requires at least trust level 2.

Results

Upon successful return, the result buffer will contain information about the request in the following format:

`QUERY JOB <Job ID>`

```
Job ID        :
Job Name      :
XML File Name :
File Machine  :
Function      :
```

```
Arguments     :
Script #n     :       (One for each SCRIPT parameter specified)
Script File #n:       (One for each SCRIPTFILE parameter specified)
Script Machine:       (If one or more SCRIPTFILE parameters were specified)
Start Date    :
Start Time    :
Source Machine:
Blocks Running:
Blocks Held   :
Clear Logs         : <Enabled | Disabled>
Log TC Elapsed Time: <Enabled | Disabled>
Log TC Num Starts  : <Enabled | Disabled>
Log TC Start/Stop  : <Enabled | Disabled>
```

```
QUERY JOB <Job ID> BLOCK <Block Name>
```

```
Block Name:
State     :
Thread ID :
Start Date:
Start Time:
```

```
QUERY JOB <Job ID> THREAD <Thread ID>
```

```
Thread ID  :
Parent TID :
Start Date :
Start Time :

Call Stack :

  Block: <Block Name>
  Break:
  Call: <Function Name>
  Continue:
  Script: <Value>
  Function: <Function Name>
  Hold:
  If: <If Expression>
  Iterate: <Index of Current Item in List>  <List (1st 40 chars)>
  Log: <Message Value (1st 40 chars)>
  Loop: var=<Var Index Value> to <Value> [by <Value>] [while <Value>] [until <Value>]
  Message: <Message Value (1st 40 chars)>
  Nop:
  Process: <Process Name>
  Parallel: <Number of Tasks>
  ParallelIterate: <Number of Iterations>/<State>
  Raise: <Signal Name>
  Release:
  Sequence: <Index fo Current Task>/<Number of Tasks>
  SignalExecutionAction: <Signal Name>
  STAFCommand: <STAF Command Name>
  Terminate:
```

```
  Testcase: <Testcase Name>
  TestcaseStatus: <Status Result>

Condition Stack:

   <Condition>: Source=<Source>, Priority=<Priority>
```

```
QUERY JOB <Job ID> TESTCASE <Test Name>
```

```
Testcase Name    :
Passes           :
Fails            :
Last Status      :
Last Status Date:
Last Status Time:
Elapsed Time     :
Number of Starts:
```

The format for "Elapsed Time" is HH:MM:SS (or HHH:MM:SS if over 99 hours) or <Pending> if the testcase has not yet been stopped in the job via a </testcase> or STOP request.

```
QUERY JOB <Job ID> PROCESS <Location:Handle>
```

```
Name            :
Location        :
Handle          :
Block Name      :
Thread ID       :
Start Date      :
Start Time      :
Command         :
Command Mode    :
Parms           :   (if specified)
Title           :   (if specified)
WorkDir         :   (if specified)
Workload        :   (if specified)
Var             :   (if specified - multiple vars can be specified)
Env             :   (if specified - multiple envs can be specified)
UseProcessVars  :   (if specified - set to true)
UserName        :   (if specified)
Password        :   (if specified)
DisabledAuth    :   (if specified)
STDIN           :   (if specified)
STDOUT          :   (if specified - prefixed with its mode)
STDERR          :   (if specified - prefixed with its mode)
ReturnSTDOUT    :   (if specified - set to true)
ReturnSTDERR    :   (if specified - set to true)
ReturnFile      :   (if specified - multiple returnfiles can be specified)
StopUsing       :   (if specified - set to true)
Console         :   (if specified)
StaticHandleName:   (if specified)
Other           :   (if specified)
```

QUERY JOB <Job ID> STAFCMD <Request#>

```
Name      :
Location  :
RequestNum:
Service   :
Request   :
Block Name:
Thread ID :
Start Date:
Start Time:
```

QUERY EXTENSIONJARFILE <Jar File Name>

```
Jar File Name         :
Version               :
Description           :
Service Version Prereq: (if the extension jar file contains service extensions)
Monitor Version Prereq: (if the extension jar file contains monitor extensions)
Parameter #n          : (one for each parameter specified in the extension xml file)
Element Name #n       : (one for each extension element)
Monitor Extension #n  : (one for each monitor extension)
```

`Jar File Name` is set to the name of the extension jar file.

`Version` is set to the version specified for the extension jar file in its manifest via the "Extension-Version" attribute. If not provided, it is set to <Not Provided>.

`Description` is set to the description specified for the extension jar file in its manifest via the "Extension-Description" attribute. If not provided, it is set to <Not Provided>.

`Service Version Prereq` is set to the minimum version of the STAX service required for this extension. It is obtained from the "Required-Service-Version" attribute in the manifest of the extension jar file. It not provided, it is set to <None>. If this extension jar file does not provide any extension elements, this line is not displayed.

`Monitor Version Prereq` is set to the minimum version of the STAX monitor required for this extension. It is obtained from the "Required-Monitor-Version" attribute in the manifest of the extension jar file. It not provided, it is set to <None>. If this extension jar file does not provide any Monitor Extensions, this line is not displayed.

`Parameter #n` is set to a Name=Value pair for each <parameter> element specified for this extension via an extension xml file. There will be a line for each parameter entry or no "Parameter #n" lines if no parameters are specified.

`Element Name #n` is set to the name of an element provided for this extension. There will be a line for each extension element or no "Element Name #n" lines if no elements are provided. If the element name is excluded in the extension xml file, <N/A> is displayed in front of the element name.

`Monitor Extension #n` is set to the name of a monitor extension provided for this extension. There will be a line for each monitor extension provided or no "Monitor Extension #n" lines if no monitor extensions are provided.

QUERY EXTENSIONJARFILES

```
Jar File Name         :
Version               :
Description           :
Service Version Prereq: (if the extension jar file contains service extensions)
Monitor Version Prereq: (if the extension jar file contains monitor extensions)
Element Name #n       : (one for each extension element)
Monitor Extension #n  : (one for each monitor extension)

Jar File Name         :
Version               :
Description           :
Service Version Prereq: (if the extension jar file contains service extensions)
Monitor Version Prereq: (if the extension jar file contains monitor extensions)
Element Name #n       : (one for each extension element)
Monitor Extension #n  : (one for each monitor extension)

... (repeated for each extension jar file)
```

Examples

Goal: Query a job with ID 5.

```
  QUERY JOB 5
```

Output:

```
  Job ID        : 5
  Job Name      : Ogre
  XML File Name : c:\stax\xml\Ogre.xml
  File Machine  : machA.austin.ibm.com
  Function      : OgreA
  Arguments     : None
  Script #1     : server1='serverA'
  Script #2     : server2='serverB'
  Script File#1 : c:\stax\python\init.py
  Script Machine: machA.austin.ibm.com
  Start Date    : 20031017
  Start Time    : 13:16:08
  Source Machine: machB.austin.ibm.com
  Blocks Running: 5
  Blocks Held   : 0
  Clear Logs         : Disabled
  Log TC Elapsed Time: Enabled
  Log TC Num Starts  : Enabled
  Log TC Start/Stop  : Disabled
```

Goal: Query a job with ID 5 displaying more detailed information about a block named main.Ogre.OgreA.

```
  QUERY JOB 5 BLOCK main.Ogre.OgreA
```

Output:

```
  Block Name: main.Ogre.OgreA
  State     : Running
  Thread ID : 1
  Start Date: 20010724
  Start Time: 13:15:21
```

Goal: Query a job with ID 8 displaying more detailed information about a thread whose thread id is 2.

```
QUERY JOB 8 THREAD 2
```

Output:

```
  Thread ID  : 2
  Parent TID : 1
  Start Date : 20010808
  Start Time : 17:33:43

  Call Stack :

    Block: main.local
    Testcase: local
    Function: TestElementsFunction
    Block: main.local.BlockWithManyElements
    Testcase: local.TestWithManyElements
    Loop: var=1 to 2
    Sequence: 1/4
    STAFCommand: STAFCommand0

  Condition Stack:

    HoldThread: Source=STAFCommand, Priority=1000
    HoldThread: Source=Block, Priority=1000
```

Goal: Query a job with ID 5 displaying more detailed information about a testcase named machA.OgreTest.

```
QUERY JOB 5 TESTCASE machA.OgreTest
```

Output:

```
  Testcase Name   : machA.OgreTest
  Passes          : 3
  Fails           : 1
  Last Status     : fail
  Last Status Date: 20031017
  Last Status Time: 13:15:53
  Elapsed Time    : 00:15:20
  Number of Starts: 1
```

Goal: Query a job with ID 4 displaying more detailed information about a process running at location machine1 with handle 91.

```
QUERY JOB 4 PROCESS machine1:91
```

Output:

```
 Name              : TestProcess
 Location          : machine1
 Handle            : 91
 Block Name        : main.machine1
 Thread ID         : 2
 Start Date        : 20010907
 Start Time        : 00:41:39
 Command           : java
 Command Mode      : default
 Parms             : com.ibm.staf.service.stax.TestProcess 2 5 100
 Title             : Test Process Title
 Var               : STAFDemo/ResourcePoolMachine=machA
 Var               : STAF/Service/Log/Mask=Error
 Env               : CLASSPATH={STAFDemo/JavaAppClassPath}
 STDOUT            : replace c:\temp\aProcess.out
 STDERR            : append c:\temp\aProcess.out
```

Goal: Query a job with ID 8 displaying more detailed information about a STAF command with request number 2019.

```
QUERY JOB 8 STAFCMD 2019
```

Output:

```
 Name      : STAFCommand3
 Location  : local
 RequestNum: 2019
 Service   : delay
 Request   : delay 5484
 Block Name: main.local.STAFCommandBlock0
 Thread ID : 2
 Start Date: 20010808
 Start Time: 16:58:47
```

Goal: Query an extension jar file with name C:/STAF/services/ExtDelay.jar displaying more detailed information about the Delay extension.

```
QUERY EXTENSIONJARFILE C:/STAF/servcies/ExtDelay.jar
```

Output:

```
Jar File Name          : C:/STAF/services/ExtDelay.jar
Version                : 1.0.0
Description            : Delay STAX Extensions
Service Version Prereq: 1.4.0
Monitor Version Prereq: 1.4.1
Parameter #1           : delay=5
Element Name #1        : ext-delay
Element Name #2        : <N/A> ext-sleep
Element Name #3        : <N/A> ext-wait
Monitor Extension #1   : ext-delay
```

Goal: Query all extension jar files displaying more detailed information about the STAX extensions.

```
  QUERY EXTENSIONJARFILES
```

Output:

```
Jar File Name          : C:/STAF/services/ExtDelay.jar
Version                : 1.0.0
Description            : Delay STAX Extensions
Service Version Prereq: 1.4.0
Monitor Version Prereq: 1.4.1
Parameter #1           : delay=5
Element Name #1        : ext-delay
Element Name #2        : <N/A> ext-sleep
Element Name #3        : <N/A> ext-wait
Monitor Extension #1   : ext-delay

Jar File Name          : C:/STAF/services/ExtMessageText.jar
Version                : 1.0.0
Description            : Message Text STAX Monitor Extension
Monitor Version Prereq: 1.4.1
Monitor Extension #1   : message
```

RELEASE

RELEASE allows you to release a job or a block in a job. The job or job block to be released has to be in HELD state.

Syntax

```
RELEASE JOB <JobID> [BLOCK <Block Name>]
```

JOB specifies the ID of the job.

BLOCK specifies a particular block in the job to release. The block name must correspond to a block element name in the XML document.

If a BLOCK is not specified, then the "main" block in the job (which, by default, encompasses the entire job) is released.

A block will not resume execution until all holds affecting it have been released.

Security

This request requires at least trust level 4.

Results

Upon successful return, the result buffer does not contain anything.

Examples

Goal: Release all of job 5.

```
  RELEASE JOB 5
```

Goal: Release job 23 in block MachineB.
```
  RELEASE JOB 23 BLOCK MachineB
```

SET

SET allows you to change the operational parameters of the STAX service. The same parameters for the SET request can be specified when registering the STAX service. Note that a setting change only effects any new STAX jobs, not STAX jobs that are already running.

Syntax

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

See section "Installation and Configuration, STAX Service Machine" for a description of these options.

Security

This request requires at least trust level 5.

Results

The result buffer will contain the option(s) requested to be set.

Examples

Goal: Enable clear logs.
```
  SET CLEARLOGS Enabled
```
Results:
```
  Clear Logs         : Enabled
```

Goal: Enable all additioal testcase logging.
```
  SET LOGTCELAPSEDTIME Enabled LOGTCNUMSTARTS Enabled LOGTCSTARTSTOP Enabled
```
Results:
```
  Log TC Elapsed Time: Enabled
  Log TC Num Starts  : Enabled
  Log TC Start/Stop  : Enabled
```

Goal: Disable logging "Start" and "Stop records each time a testcase begins and ends.
```
  SET LOGTCSTARTSTOP Disabled
```
Results:
```
  Log TC Start/Stop  : Disabled
```

START

START allows you to start a new testcase in a STAX job that is currently running. It performs basically the same action as a <testcase> element with the 'strict' mode, including logging a "Start" level message in the STAX Job log if "Log TC Start/Stop" is enabled for the job. A START request sets the start time for the testcase from which its elapsed time will be calculated. For example, a START request can be submitted from a shell script or Java program, etc. that is run via a <process> element to start a testcase.

Syntax

```
START JOB <JobID> TESTCASE <Testcase Name> [KEY <Key>]
```

JOB specifies the ID of the job.

TESTCASE specifies the fully qualified name of the testcase to be started. If a testcase with the same name is currently active, you must specify a key to uniquely identify the active testcase. You may want to prefix the name of the testcase with the value provided in the STAXCurrentTestcase Python variable to maintain the testcase hierarchy.

KEY specifies a unique identifier for this testcase when combined with the Testcase name. A key must be specified if a testcase with the same name is already currently active (started and not yet stopped).

For example, if your STAX job contains a <paralleliterate> element that iterates over a list of machines and this element contains a <process> element and the process submits a START request for a testcase, a key must be specified in the START request to uniquely identify the testcase. For example, you may want to specify the machine name where the process is running and the process handle for the key, e.g. KEY machineA:48.

We strongly recommend that you always specify a KEY so that the testcase can be run in parallel.

Security

This request requires at least trust level 3.

Results

Upon successful return, the result buffer does not contain anything.

Examples

Goal: Start a testcase named "Scenario01.Test2.MachineA" in job 3.

```
  START JOB 3 TESTCASE "Scenario01.Test2.MachineA"
```

Goal: Start a testcase named "Scenario01.Test3" in job 12 in parallel on machine clientA (process handle 48) and on machine clientB (process handle 86).

```
  START JOB 12 TESTCASE Scenario01.Test3 KEY clientA:48
  START JOB 12 TESTCASE Scenario01.Test3 KEY clientB:86
```

STOP

STOP allows you to stop an active testcase (which was started via a START request) in a STAX job that is currently running. It performs basically the same action as a </testcase> element, including logging a "Stop" level message in the STAX Job log if "Log TC Start/Stop" is enabled. A STOP request sets the stop time for the testcase from which its elapsed time will be calculated.

For example, a STOP request can be submitted from a shell script or Java program, etc. that is run via a <process> element to stop a testcase.

Syntax

```
STOP JOB <JobID> TESTCASE <Testcase Name> [KEY <Key>]
```

JOB specifies the ID of the job.

TESTCASE specifies the fully qualified name of the active testcase to be stopped. The testcase name specified must already exist and must have been started via a START request.

KEY specifies a unique identifier for the testcase when combined with the Testcase name. If the testcase was started with a key specified, the same key must be specified in the testcase's corresponding STOP request.

Security

This request requires at least trust level 3.

Results

Upon successful return, the result buffer does not contain anything.

Examples

Goal: Stop a testcase named "Scenario01.Test2.MachineA" in job 3.

```
  STOP JOB 3 TESTCASE "Scenario01.Test2.MachineA"
```

Goal: Stop a testcase named "Scenario01.Test3" in job 12 that is running in parallel on machines clientA (process handle 48) and clientB (process handle 86).

```
  STOP JOB 12 TESTCASE Scenario01.Test3 KEY clientA:48
  STOP JOB 12 TESTCASE Scenario01.Test3 KEY clientB:86
```

TERMINATE

TERMINATE allows you to terminate a job or a block in a job.

Syntax

```
TERMINATE JOB <JobID> [BLOCK <Block Name>]
```

JOB specifies the ID of the job to terminate. If a BLOCK is not specified, then the "main" block in the job (which, by default, encompasses the entire job) is terminated which means all processes and STAF commands currently running are terminated.

BLOCK specifies a particular block in the job to terminate. The block name must correspond to a block element name in the XML document. If a block in a job is terminated, the processes and STAF commands that are currently running in the block are terminated.

Security

This request requires at least trust level 4.

Results

Upon successful return, the result buffer does not contain anything.

Examples

Goal: Terminate all of job 5.
```
  TERMINATE JOB 5
```

Goal: Terminate job 23 in block MachineB.
```
  TERMINATE JOB 23 BLOCK MachineB
```

UPDATE

UPDATE allows you to update the status of a testcase in a job that is currently running. It performs basically the same action as a <tcstatus> element contained in a job's XML file. For example, an update request can be performed within a process running in a job to update the status for a testcase.

Syntax

```
UPDATE JOB <JobID> TESTCASE <Testcase name> STATUS <Status> [MESSAGE <Message text>] [FORCE]
```

JOB specifies the ID of the job.

TESTCASE specifies the fully qualified name of the testcase whose status is to be updated. The testcase specified must already exist, unless FORCE is also specified. If you wanted a process defined by a <process> element in a job to update the status of the current testcase, the value of the STAXCurrentTestcase Python variable could be passed to the process via the <parms> element or <env> element and an UPDATE request could be submitted from within the process, specifying the current testcase value passed in for the TESTCASE option.

STATUS specifies whether the testcase passed or failed. The result must be Pass or Fail (not case-sensitive). The pass or fail status for the testcase will be incremented accordingly.

MESSAGE specifies additional information about the status for a testcase. It is optional.

FORCE specifies that if the testcase does not already exist, it should be added to the testcase map. Note that without a FORCE option, you cannot update the status of a testcase without the testcase already existing.

An event will be generated that sends a message to the STAX Job Monitor with the testcase status information.

Testcase status information is displayed in the "Testcase Information" section of the STAX Monitor GUI. A summary of the number of passes and fails for each testcase, as well as any additional information specified about the status of a testcase is logged in the STAX Job Log.

Security

This request requires at least trust level 3.

Results

Upon successful return, the result buffer does not contain anything.

Examples

Goal: Increment the pass status for existing testcase "Scenario01.Test2.MachineA" in job 3.

```
  UPDATE JOB 3 TESTCASE "Scenario01.Test2.MachineA" STATUS pass
```

Goal: Increment the fail status for testcase "Memory Test" in job 21 and record a message about the cause of the failure. If the testcase does not exist, create it.

```
  UPDATE JOB 21 TESTCASE "Memory Test" STATUS Fail MESSAGE "RC=1. Open a defect" FORCE
```

VERSION

VERSION displays the version level of the STAX service.

Syntax

```
VERSION
```

Security

This request requires at least trust level 2.

Results

The result buffer contains the version level of the STAX service.

STAX Monitoring

A Monitor application is available for the STAX Service. This application displays a real-time graphical representation of the currently running elements of a given STAX job.

The STAX Monitor application displays a list of all active jobs and an indication of which jobs are currently being monitored. The STAX Monitor application also provides a graphical user interface for the EXECUTE request which allows you to submit new jobs for execution and monitor the job from its beginning, if desired.

For each job that is monitored, the STAX Monitor application displays all currently executing <process>, <stafcmd>, <block>, and <job> elements for a STAX job. The graphical representation is a tree format, in order to show the hierarchy of the currently executing elements. The STAX Monitor application allows you to control the execution of the job by selecting a <block> element to hold, release, or terminate.

The STAX Monitor application also displays any testcases that have been executed and their status as well as any messages that have been sent via <message> elements or from the STAX service itself.

Multiple jobs can be monitored at the same time.

Starting the STAX Monitor

You can start the STAX Monitor using the -jar option, without adding the STAXMon.jar file to your CLASSPATH environment variable, if your STAXMon.jar file is in a first or second level directory off your STAF root directory (such as C:\STAF\services or

C:\STAF\services\STAX on Windows or /usr/local/staf/services or /usr/local/staf/services/STAX on Unix systems). Note that when using the -jar option, Java does not use the CLASSPATH environment variable or the -cp option.

If your current directory contains STAXMon.jar, then to start the STAX Monitor, type:

```
   java -jar STAXMon.jar
```

Or, if you are in another directory, fully qualify the STAXMon.jar file. So, if the STAXMon.jar file is in C:\STAF\services, type:

```
   java -jar C:\STAF\services\STAXMon.jar
```

Or, you can start the STAX Monitor by specifying the main Java class name (which is case sensitive). When using this method, you must make sure that the STAXMon.jar and JSTAF.jar files are in your classpath and then type:

```
   java com.ibm.staf.service.stax.STAXMonitor
```

The STAX Monitor accepts the following command line parameters:

```
   -job <jobNumber> [-closeonend]
   -jobparms <jobParmsFile> [-closeonend]
   -extensions
   -properties [-staxMachine <machineName>] [-staxServiceName <serviceName>]
               [-eventMachine <machineName>] [-eventServiceName <serviceName>]
               [-noStart]
   -version
   -help
```

`-job` specifies an existing job ID to monitor. This option will resolve variables.

`-closeonend` specifies that the STAX Monitor GUI should be closed when the monitored job ends. By default, the STAX Monitor GUI remains open when the monitored job ends.

`-jobparms` specifies a Job Parameters File for which a new STAX job should be submitted (and monitored, if specified in the Job Parameters File). A Job Parameters File can be created by using the File Save or File Save As option on the STAX Monitor's "Start Job Parameters" window. This option will resolve variables.

`-extensions` displays the monitor extensions that are registered with the STAX Monitor.

`-properties` specifies to update one or more properties for the STAX Monitor.

`-staxMachine` specifies the name of the machine where the STAX service is running. This option will resolve variables.

`-staxServiceName` specifies the name used to register the STAX service. This option will resolve variables.

`-noStart` specifies to not start the STAX Monitor after updating its properties.

`-version` displays the version of the STAX Monitor.

`-help` displays help information for the STAX Monitor.

Examples Starting the STAX Monitor Using Parameters

Start the STAX Monitor specifying to monitor a current running STAX job with job ID 2:

```
java -jar STAXMon.jar -job 2
```

Start the STAX Monitor specifying to submit a new STAX job defined by Job Parameters File "C:\Documents and Settings\Administrator\Test6":

```
java -jar STAXMon.jar -jobParms "C:\Documents and Settings\Administrator\Test6"
```

Update properties for the STAX Monitor, such as setting the STAX service machine to server1.ibm.com and start the STAX Monitor:

```
java -jar STAXMon.jar -properties -staxMachine server1.ibm.com
```

Update properties for the STAX Monitor, such as setting the name of the STAX service to STAX2, but don't start the STAX Monitor:

```
java -jar STAXMon.jar -properties -staxServiceName STAX2 -noStart
```

Setting STAX Monitor Properties

The first time you start the STAX Monitor, the "STAX Monitor Properties" window will be displayed (unless you specify the -properties option when starting the STAX Monitor).

The "STAX Monitor Properties" window contains five main tabs to allow properties for the STAX Monitor to be specified.

1. **"Machine Info" tab:**

   This tab allows you to specify the machine and service name of the STAX Service to which the STAX Monitor will be communicating.

   - Specify the machine and service name of the STAX Service machine:

     a. **STAX Machine -** Specify the name of the machine where the STAX service is running. It defaults to the local machine (the machine where you're running the STAX Monitor).

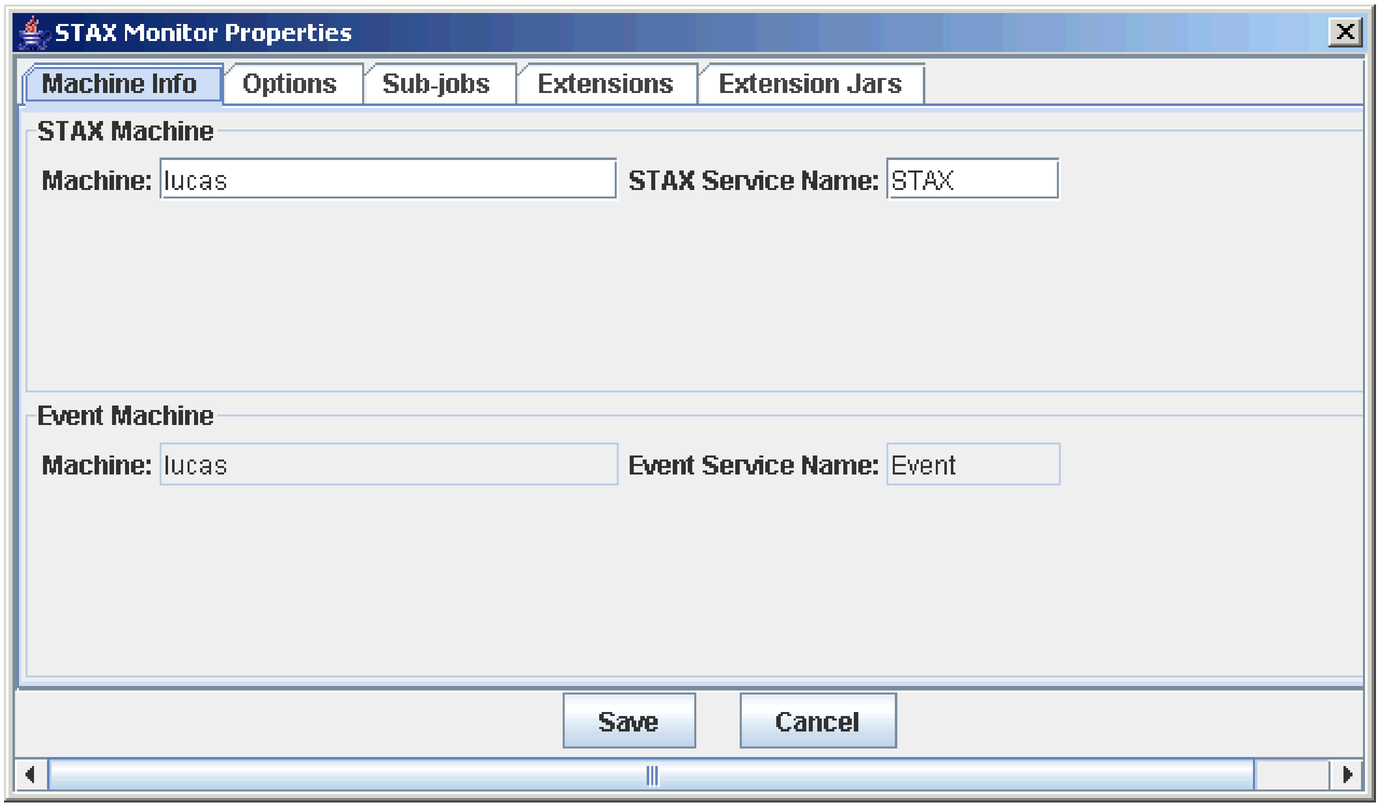
     b. **STAX Service Name -** Specify the name used to register the STAX service. It defaults to STAX. Typically, this value does not need to be changed.

   - Displays the machine name and service name of the Event Service used by the STAX Service:

     a. **Event Machine -** Displays the name of the Event Service machine used by the STAX service. It defaults to the specified STAX Machine if it cannot be obtained by submitting a "LIST SETTINGS" request to the specified STAX machine and STAX service. This field cannot be edited.

     b. **Event Service Name -** Displays the name of the Event Service used by the STAX service. It defaults to Event if it cannot be obtained by submitting a "LIST SETTINGS" request to the specified STAX machine and STAX service. This field cannot be edited.

   Following is an example of the "Machine Info" tab:



STAX Machine

Machine: lucas

STAX Service Name: STAX

Event Machine

Machine: lucas

Event Service Name: Event

2. **"Options" tab:**

   This tab allows you to specify options for the STAX Monitor:

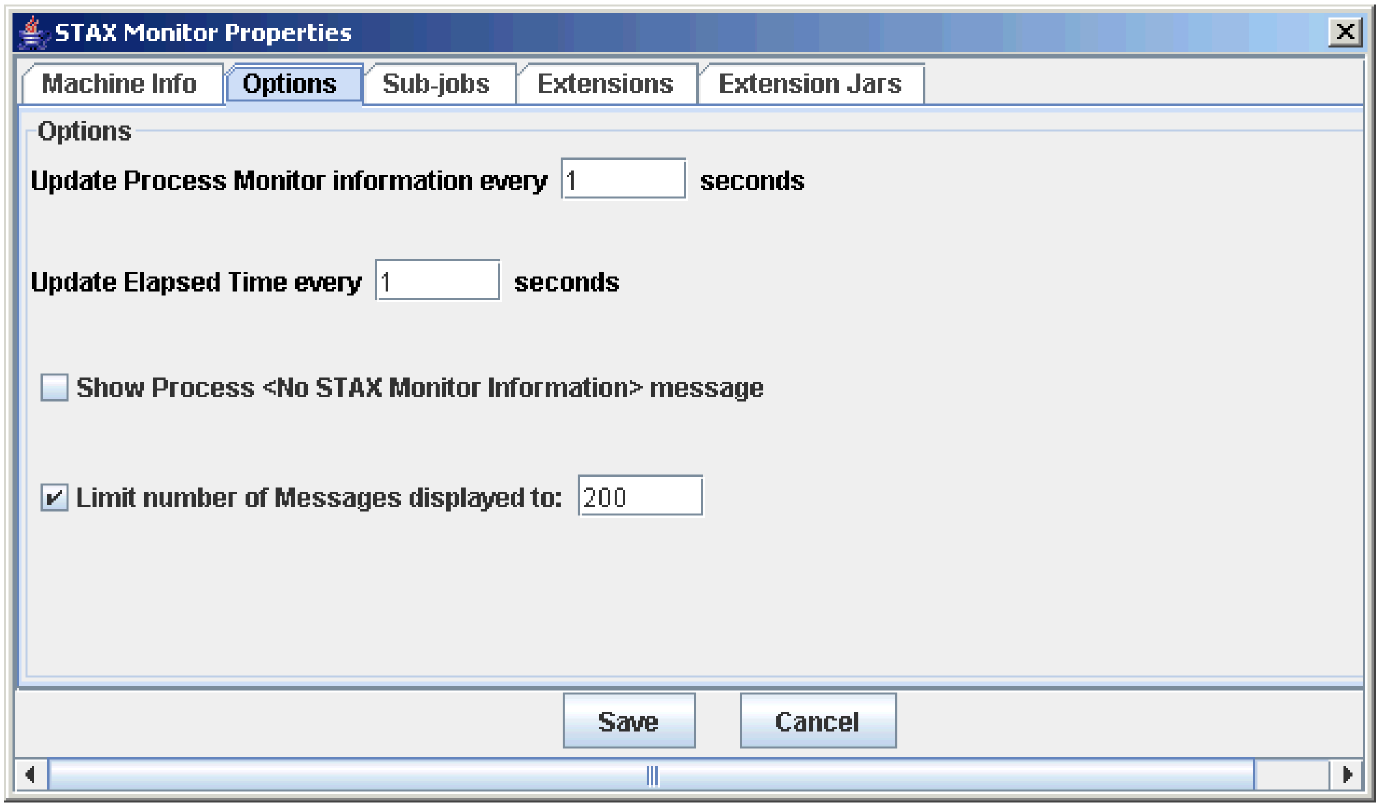
   a. **Update Process Monitor information every x seconds -** Specify how frequently to refresh the Monitor information for processes. The default is 1 (every second). The valid values are 0 or greater (0 indicates that the process monitor information should never be displayed). Saved updates to the seconds value will apply to all new STAX Monitor windows.

   b. **Update Elapsed Time every x seconds -** Specify how frequently to refresh the elapsed times with the STAX Monitor. The default is 1 (every second). The valid values are 0 or greater (0 indicates that the elapsed times should never be displayed). Saved updates to the seconds value will apply to all new STAX Monitor windows.

   c. **Show Process <No STAX Monitor Information> message -** Specify whether to show the "<No STAX Monitor Information>" message for processes that have no Monitor Information available. The default is to not display this message.

   d. **Limit number of Messages displayed to -** Specify the limit of the number of messages that can be displayed in the Messages table. The default is to limit the number of messages to 200 (only the most recent 200 messages will be displayed in the Messages table).

   Following is an example of the "Options" tab with its default values:



Options

Update Process Monitor information every 1 seconds

Update Elapsed Time every 1 seconds

☐ Show Process <No STAX Monitor Information> message

☑ Limit number of Messages displayed to: 200

3. **"Sub-jobs" tab:**

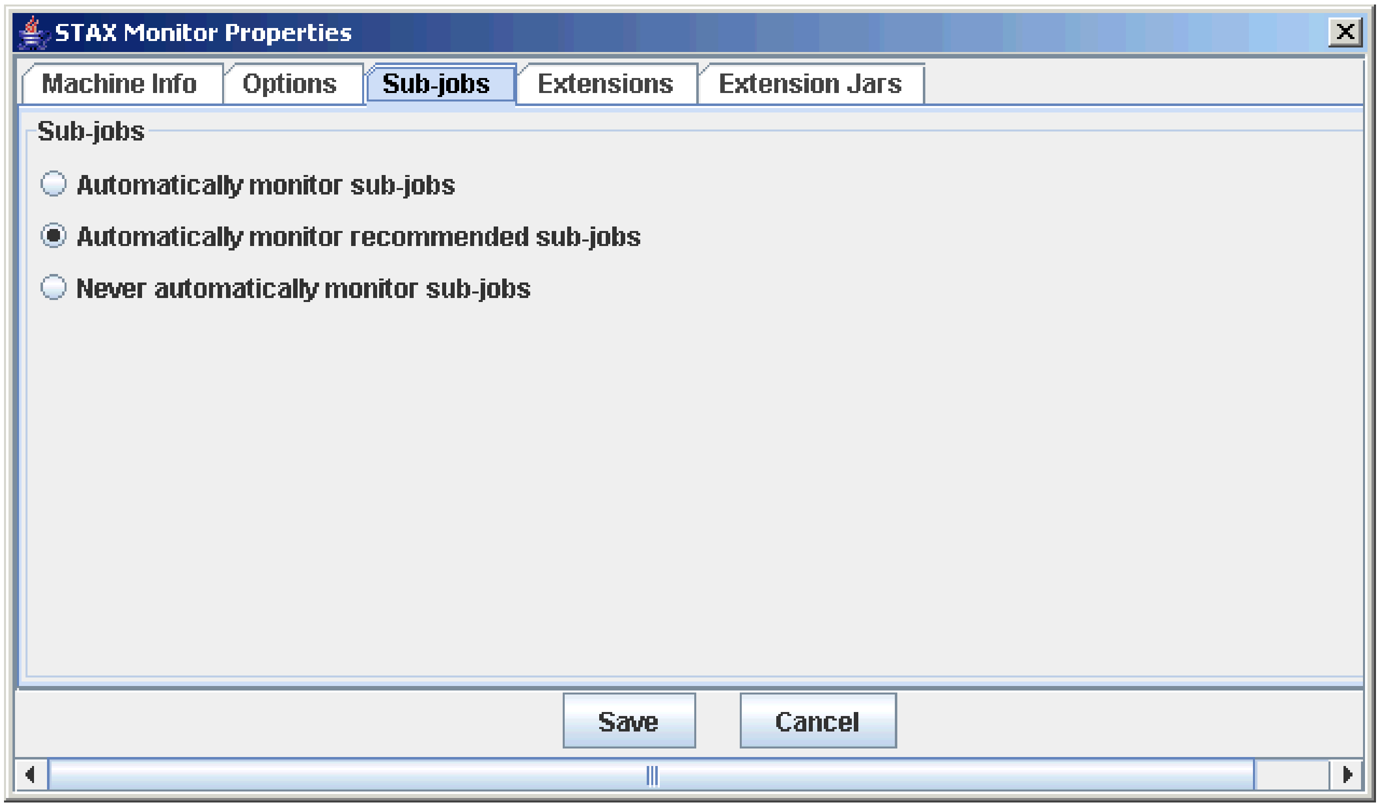
   This tab allows you to specify whether to automatically monitor sub-jobs. Choose one of the following options:

   a. **Automatically monitor sub-jobs -** A new STAX Monitor window will be opened for every sub-job that is started by the current job.

   b. **Automatically monitor recommended sub-jobs -** A new STAX Monitor window will be opened only for sub-jobs whose <job> element attribute "monitor" is set to a true value. This is the default.

   c. **Never automatically monitor sub-jobs -** Sub-jobs will never be automatically monitored.

   Following is an example of the "Sub-jobs" tab:



Sub-jobs

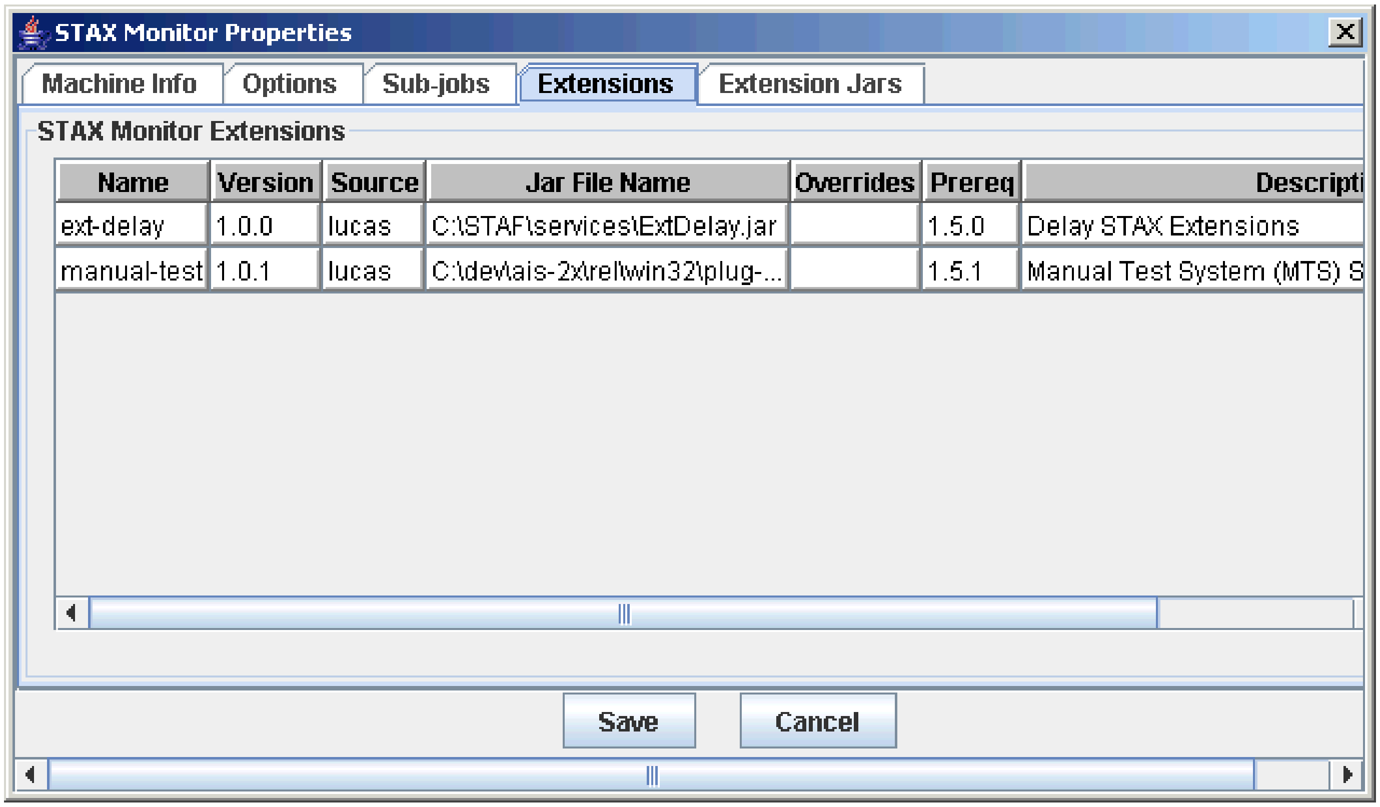
Automatically monitor sub-jobs [✓ Automatically monitor recommended sub-jobs] Never automatically monitor sub-jobs

4. **"Extensions" tab:**

   This tab allows you to display the monitor extensions that are registered with the STAX Monitor.

   - **Name** - The name of the extension specified in the extension jar file's manifest.
   - **Version** - The version of the extension, if provided in the extension jar file's manifest, or if not provided.
   - **Source** - The source machine where the extension jar file containing this monitor extension resides. It will either be the name of the STAX service machine or local if the extension was specified via the "Extension Jars" tab.
   - **Jar File Name** - The name of the extension jar file that contains this monitor extension.
   - **Overrides** - If not blank, the name of the extension jar file that contains an overridden monitor extension.
   - **Prereq** - The minimum required version of the STAX Monitor that this extension requires, if provided in the extension jar file's manifest, or if not provided.
   - **Description** - A description of the extension jar file, if provided in the extension jar file's manifest, or if not provided

   Following is an example of the "Extensions" tab with some monitor extensions registered:



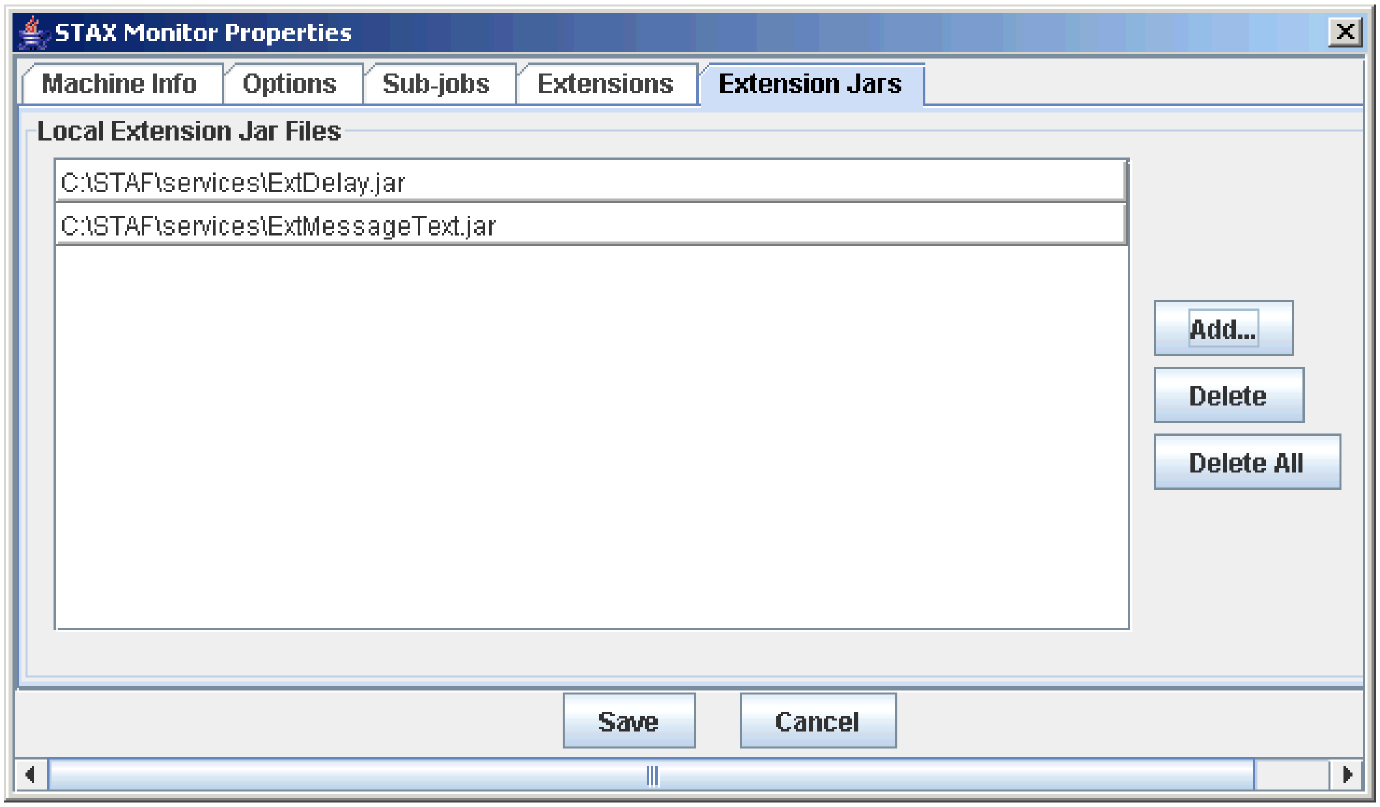
Name	Version	Source	Jar File Name	Overrides	Prereq	Descripti
ext-delay	1.0.0	lucas	C:\STAF\services\ExtDelay.jar		1.5.0	Delay STAX Extensions
manual-test	1.0.1	lucas	C:\dev\ais-2x\rel\win32\plug-...		1.5.1	Manual Test System (MTS) S

5. **"Extension Jars" tab:**

   This tab allows you to specify any local extension jar files containing STAX Monitor extensions that you want to register. Note that as of STAX V1.5.0, any STAX monitor extensions that are registered with the STAX service will be automatically made available to the STAX Monitor. You should only specify local extension jar files that are not already registered with the STAX service or that contain monitor extensions that you want to override (e.g. with a later version of the extension).

   - **Local Extension Jar Files** - Specify the fully-qualified names of jar files on the local system that contain monitor extensions to be registered. You may specify as many extension jar files as needed. This is an optional field.
     - Click on the "Add" button to add a new extension jar file.
     - To delete an Extension Jar File that you already added, select it from the list and click on the "Delete" button.
     - Click on the "Delete All" button to delete all extension jar files in the list.

   Following is an example of the "Extension Jars" tab with some extension jar files added:



C:\STAF\services\ExtDelay.jar
C:\STAF\services\ExtMessageText.jar

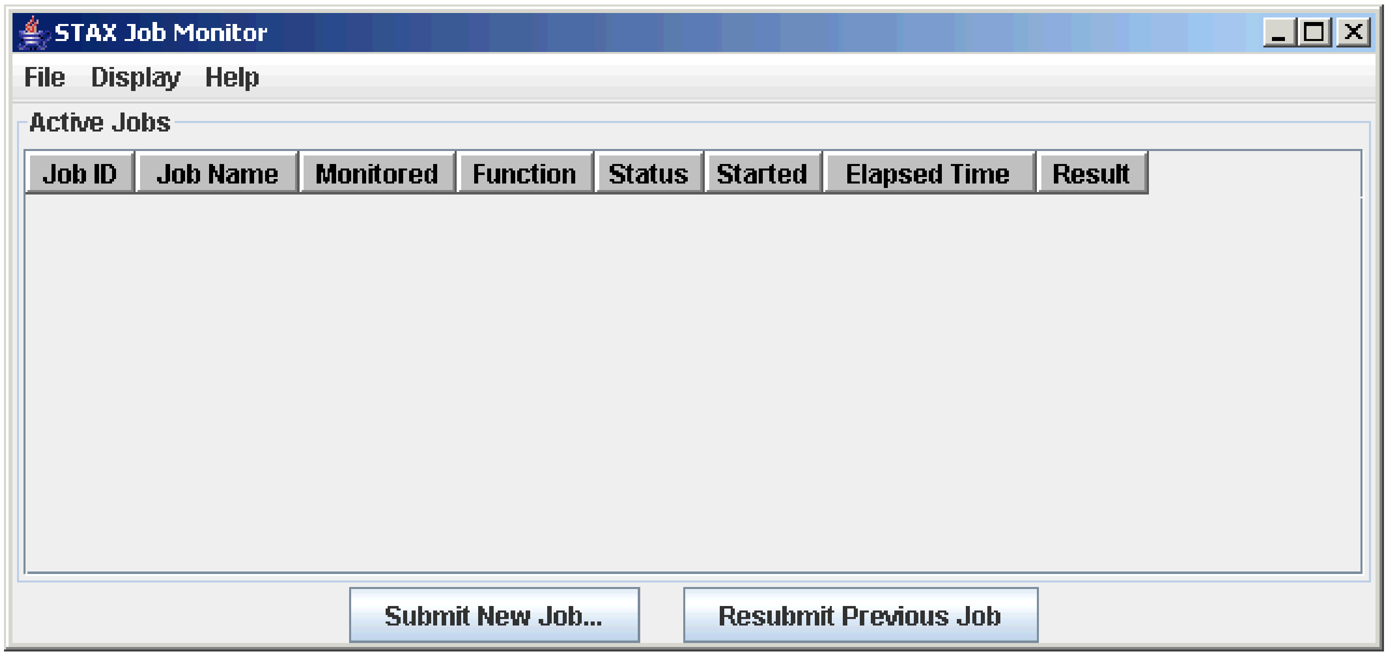
Click on the "Save" button to save any changes (or the "Cancel" button to end without saving changes). The next window that will be displayed is the "STAX Job Monitor" window which contains a list of active jobs.

When you close and restart the STAX Monitor, the "STAX Properties" panel options last entered are restored.

Note that if this is is not the first time the STAX Monitor has been started, the "STAX Job Monitor" window is displayed first instead of the "STAX Monitor Properties" window. To update the properties from the "STAX Job Monitor" window, select "File" from the menu bar and then select "Properties..." in order to display the "STAX Monitor Properties" window. You must stop and restart the STAX Monitor before the properties update will take place.

Displaying a List of Active Jobs

Following is an example of the "STAX Job Monitor" window with no jobs currently running or monitored:



The "STAX Job Monitor" displays STAX jobs that are currently running on the specified STAX machine (as well as completed jobs which are currently being monitored) in the "Active Jobs" table. Sub-jobs will appear as separate jobs and are also displayed in the "STAX Job Monitor" window.

The **File** menu bar contains the following menu items:

- **Properties... -** This option displays the STAX Job Monitor Properties dialog.
- **Submit New Job... -** This option displays the Start New Job dialog.
- **Resubmit Previous Job -** This option resubmits that last job that was submitted.
- **Exit -** Selecting this option closes the STAX Job Monitor window.

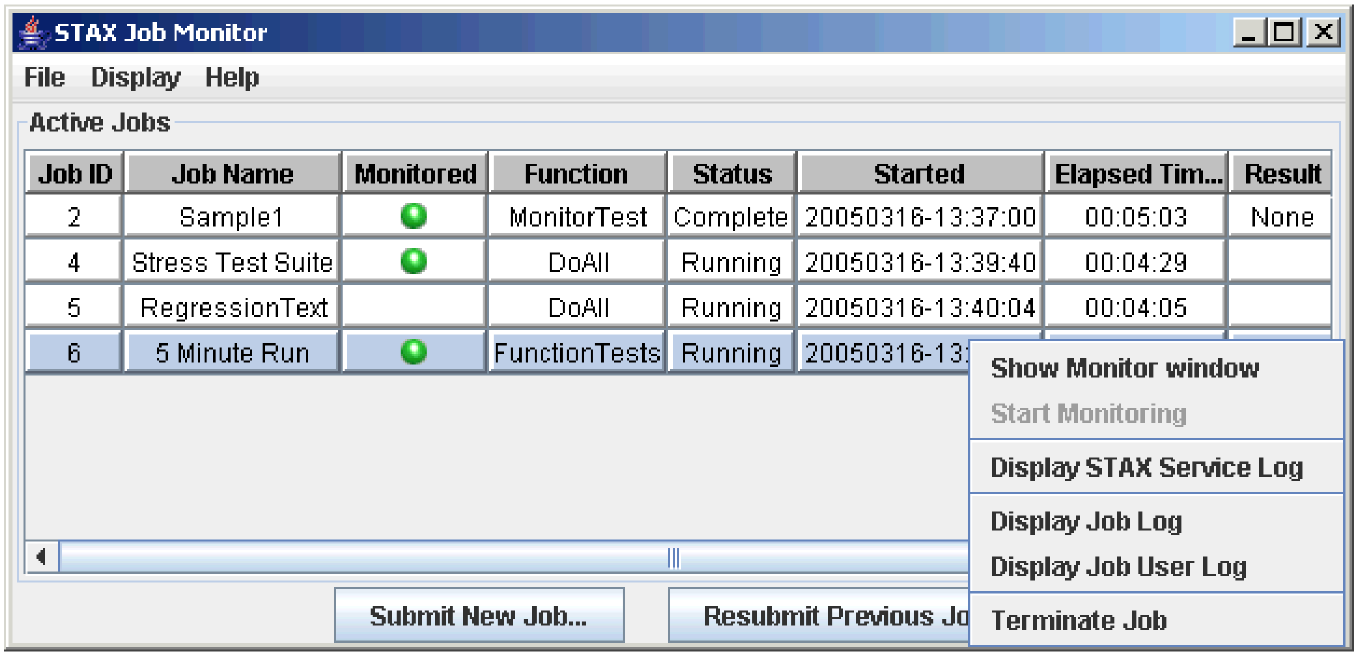
The **Display** menu bar contains the following menu items:

- **Display STAX Service Log -** Selecting this option causes the STAX Service Log to be displayed.
- **Display Selected Job's Log -** Selecting this option causes the Job Log for the currently selected Job to be displayed.
- **Display Selected Job's User Log -** Selecting this option causes the Job User Log for the currently selected Job to be displayed. If there are no entries in the Job User Log, then an informational message will be displayed to indicate that there are no entries in the log.
- **Display Job Log... -** Selecting this option allows you to display the Job Log for any Job ID.
- **Display Job User Log... -** Selecting this option allows you to display the Job User Log for any Job ID. If there are no entries in the Job User Log, then an informational message will be displayed to indicate that there are no entries in the log.

The "Active Jobs" table shows the following information for each active job:

- **Job ID -** Job number for an active job.
- **Job Name -** Name specified for the job, or <N/A> if no job name was specified when submitting the job for execution.
- **Monitored -** A green ball indicates that the job is currently being monitored. No green ball indicates that the job is currently running but is not being monitored.
- **Function -** Name of the function that was called to start the job.
- **Status -** "Running" if the job is currently running or "Complete" if the job has completed but is still being monitored.
- **Started -** Date and time that the job was started. The format is YYYYMMDD-HH:MM:SS.
- **Elapsed Time -** The elapsed time that the job has been running (or if the status is complete, the elapsed time that the job ran). The format is HH:MM:SS. If the elapsed time exceeds 99 hours, the format is HHH:MM:SS.
- **Result -** A "string" version of the Job Result.

Following is the "STAX Job Monitor" window which shows four active jobs:



Job ID	Job Name	Monitored	Function	Status	Started	Elapsed Tim...	Result
2	Sample1	●	MonitorTest	Complete	20050316-13:37:00	00:05:03	None
4	Stress Test Suite	●	DoAll	Running	20050316-13:39:40	00:04:29	
5	RegressionText		DoAll	Running	20050316-13:40:04	00:04:05	
6	5 Minute Run	●	FunctionTests	Running	20050316-13:		

If you right-mouse-click on a job in the "Active Jobs" table, a popup menu is displayed with the following options:

- **Show Monitor window** - This option is only enabled if you are currently monitoring the selected job. Selecting this option causes the "Job Monitor" window for this job to be brought to the foreground.
- **Start Monitoring** - This option is only enabled if you are not currently monitoring the selected job. Selecting this option causes a Job Monitor window to be created for this job.
- **Display STAX Service Log** - This option allows you to view the STAX Service Log.
- **Display Job Log** - This option is always enabled. Selecting this option causes the Job Log to be displayed.
- **Display Job User Log** - This option is always enabled. Selecting this option causes the Job User Log to be displayed. If there are no entries in the Job User Log, then an informational message will be displayed to indicate that there are no entries in the log.
- **Terminate Job** - This option is enabled for all running Jobs. Selecting this option causes the job to be terminated. Terminating a parent job will terminate any sub-jobs as well.

Submitting a New Job for Execution

The STAX Monitor provides a graphical user interface for submitting an EXECUTE request (as an alternative to issuing an EXECUTE request via the command line). To submit a new job for execution from the "STAX Job Monitor" window, click on the "Submit New Job..." button or select "File" from the menu bar and then select "Submit New Job...". A "Start Job Parameters" window will be displayed.

Note that when you close and restart the STAX Monitor, the options on the "Start Job Parameters" panel last entered are restored.

The "STAX Job Parameters" window contains five main tabs to allow job execution parameters to be specified.

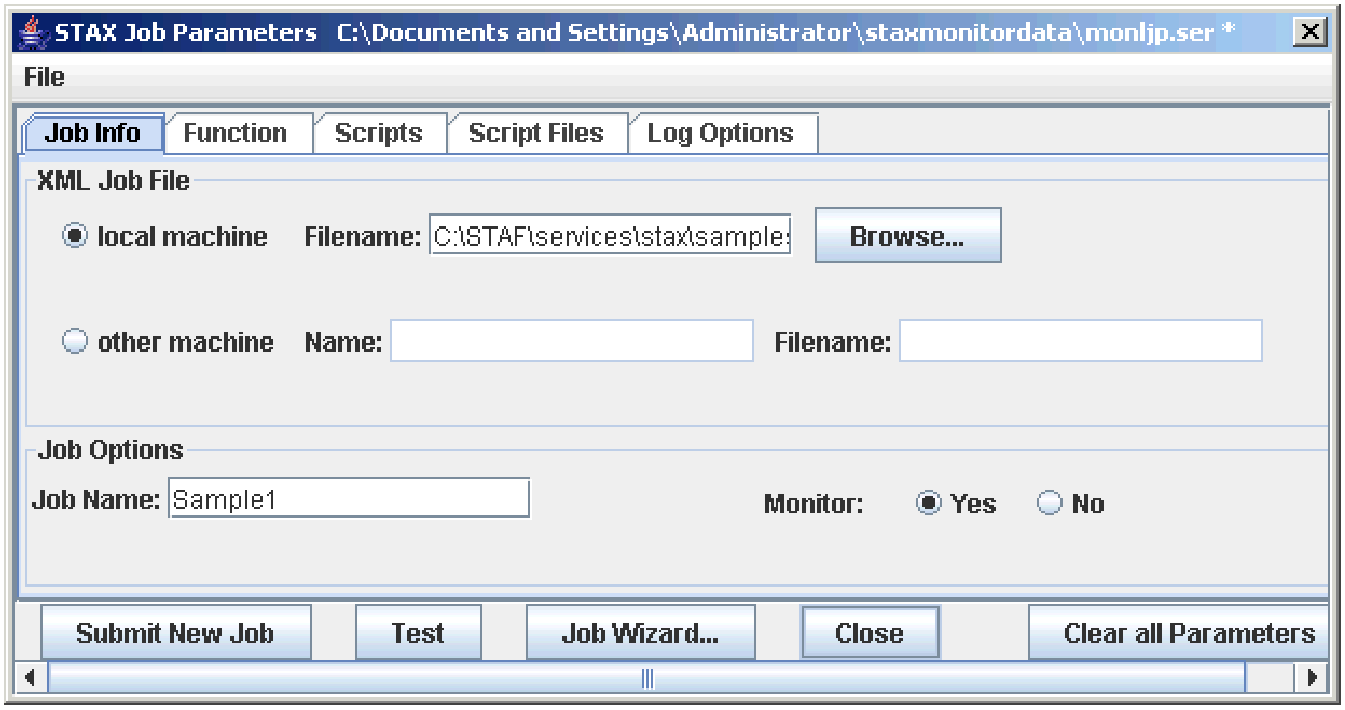
1. **"Job Info" tab:**

   This tab allows you to specify the following job execution parameters:

   - **XML Job File** - Specify the fully-qualified name of the XML file to be executed. You may select "local machine" and enter the local file name, or select "other machine" and specify the machine and file names. For XML files on local machines, a "Browse..." button is provided which displays a "Select an XML Job Definition File" panel to allow you to select the file.

- **Job Name** - Specify a name which identifies the job. This field is optional.
- **Monitor** - Select "Yes" or "No" to indicate whether you want to monitor the job you are submitting.

Following is an example of the "Start Job Parameters" panel with the "Job Info" tab selected and with some fields filled in:


STAX Job Parameters C:\Documents and Settings\Administrator\staxmonitordata\monljp.ser *

XML Job File

local machine [✓] Filename: C:\STAF\services\stax\sample:

other machine Name: Filename:

Job Options

Job Name: Sample1

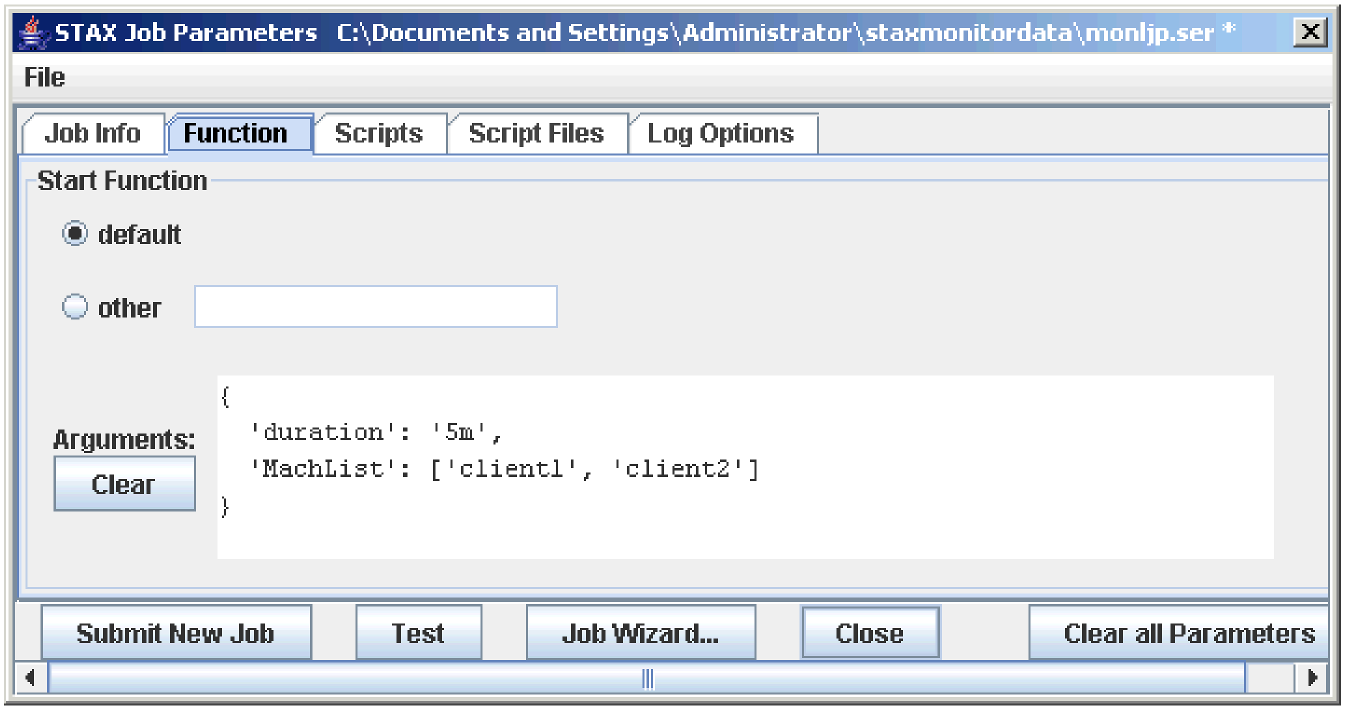
Monitor: [✓ Yes] No

2. **"Function" tab:**

This tab allows you to specify the start function parameter for the job:

- **Start Function** - Select "default" to start executing the job at its default start function. Select "other" to specify the name of the function to be called to begin the job. If you select "default" and the XML document does not contain a <defaultcall> element, you will get a STAXInvalidStartFunctionException if you try to execute the job. You can pass parameters to the Start Function by specifying the parameters in the Arguments field.

Following is an example of the "Start Job Parameters" panel with the "Function" tab selected and with some fields filled in:



Start Function: [✓ default] other

Arguments:

```
{
  'duration': '5m',
  'MachList': ['client1', 'client2']
}
```

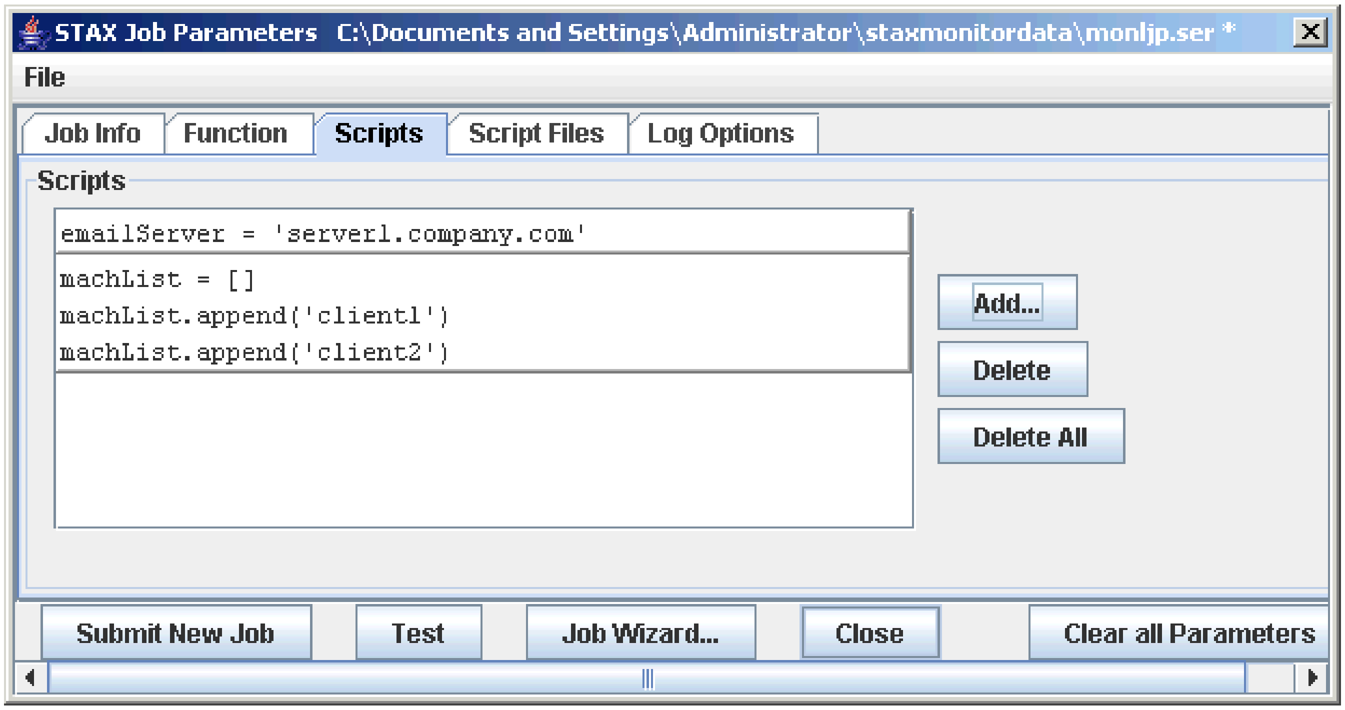
3. **"Scripts" tab:**

   This tab allows you to specify Script parameters:

   - **Scripts -** Specify any Python code to be executed. You may specify as many Script parameters as needed. This is an optional field. See the "EXECUTE" section for more information on a SCRIPT parameter.

     Click on the "Add" button to add a new Script parameter. To edit a script already in the list, double click on the script. To delete a Script parameter that you already added, select it from the list and click on the "Delete" button. Click on the "Delete All" button to delete all Scripts in the list.

   Following is an example of the "Start Job Parameters" panel with the "Scripts" tab selected and with some fields filled in:



```
emailServer = 'server1.company.com'
```

```
machList = []
machList.append('client1')
machList.append('client2')
```

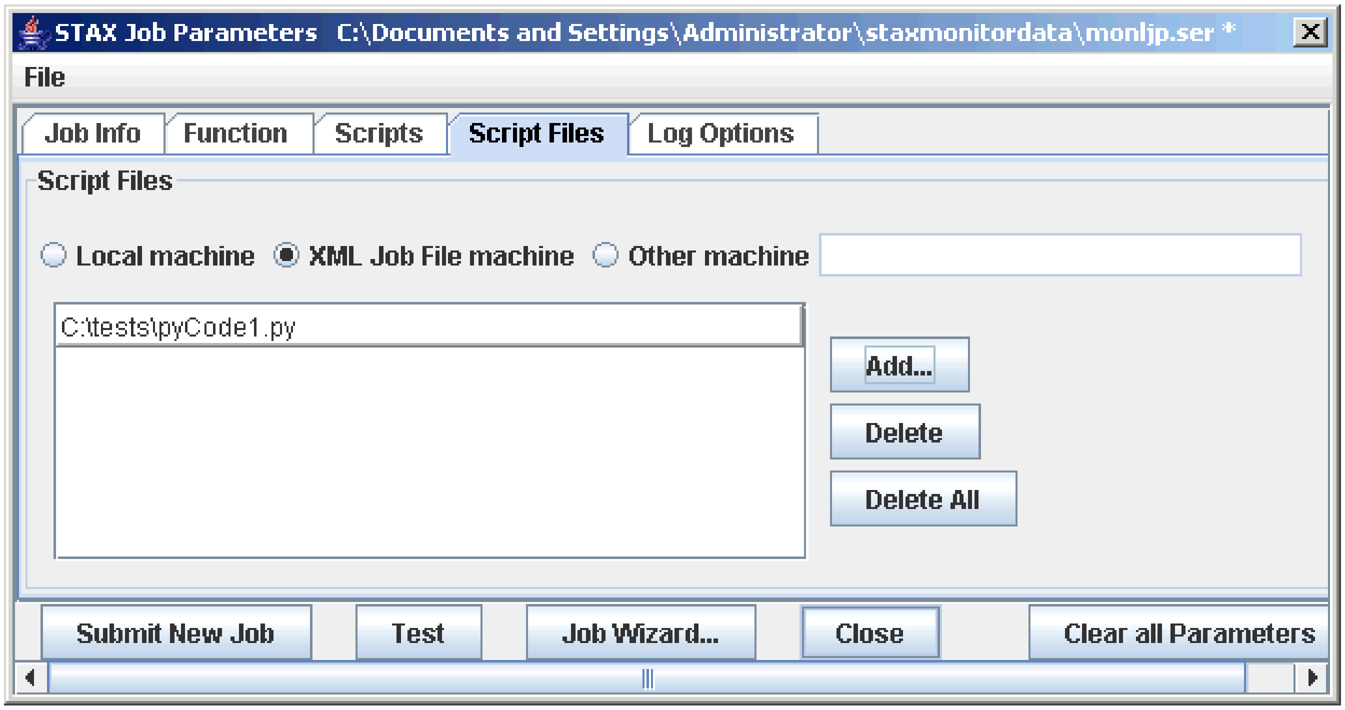
4. **"Script Files" tab:**

   This tab allows you to specify Script File parameters:

   - **Script File Machine -** Specify the machine where the script files are located. You may select one of the following:
     - **Local machine -** to indicate the script files are located on the local STAX Monitor machine.
     - **XML Job File machine -** to indicate the script files are located on the same machine specified for the XML Job File. This is the default.
     - **Other machine -** to specify the name of the machine where the script files are located

   - **Script Files -** Specify the fully-qualified names of script files that contain Python code to be executed. You may specify as many script files as needed. This is an optional field. See the "EXECUTE" section for more information on a SCRIPTFILE parameter.
     - Click on the "Add" button to add a new script file.
     - To edit the name of a script file already in the list, double click on the script file name.
     - To delete a script file that you already added, select it from the list and click on the "Delete" button.
     - Click on the "Delete All" button to delete all script files in the list.

   Following is an example of the "Start Job Parameters" panel with the "Script Files" tab selected and with some fields filled in:


STAX Job Parameters C:\Documents and Settings\Administrator\staxmonitordata\monljp.ser *

Script Files

( ) Local machine (●) XML Job File machine ( ) Other machine

C:\tests\pyCode1.py

5. **"Log Options" tab:**

   This tab allows you to specify the log parameters:

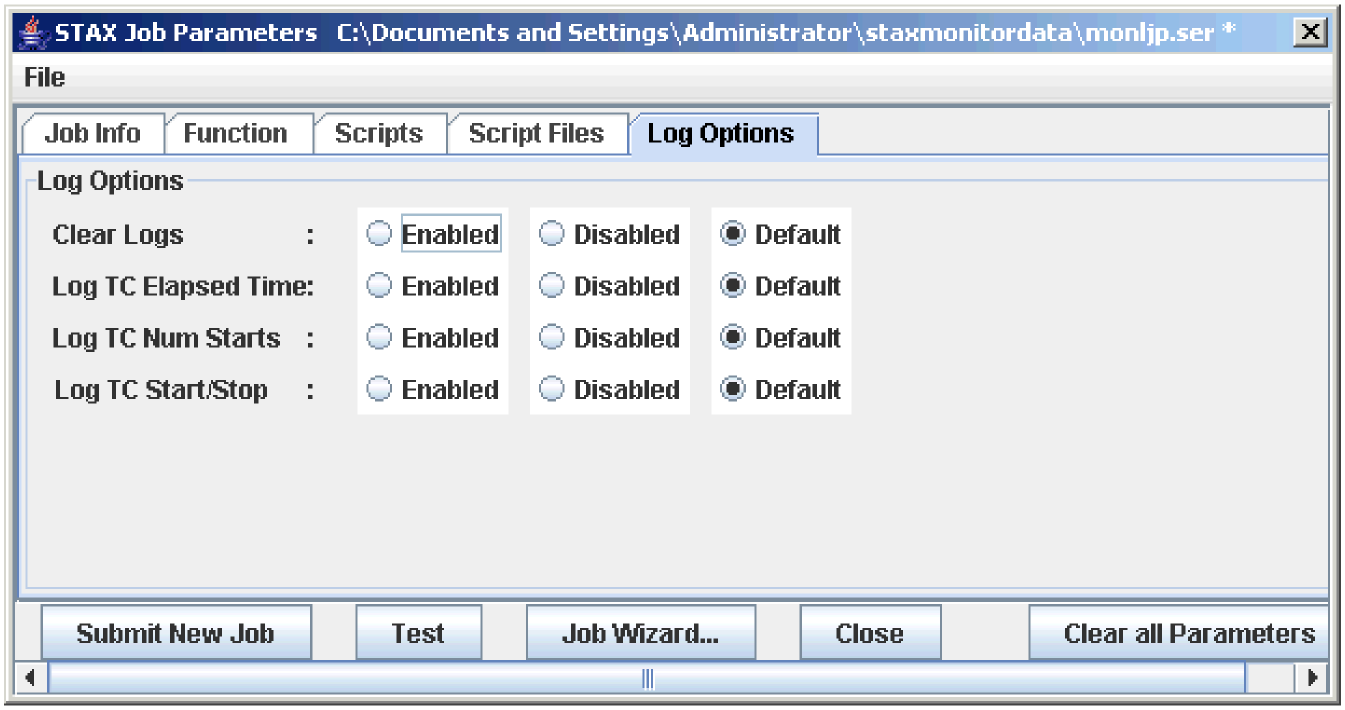
   - **Clear Logs -** Select "Enabled", "Disabled", or "Default" to indicate whether the STAX Job and Job User logs should be deleted before the job is executed. Selecting "Enabled" specifies to delete the logs. Selecting "Disabled" specifies not to delete the logs. Selecting "Default" specifies to use the STAX service setting for "Clear Logs".

   - **Log TC Elapsed Time -** Select "Enabled, "Disabled", or "Default" to indicate whether to log the elapsed time in the testcase summary records at the end of the STAX Job log and in a LIST TESTCASES request. Selecting "Enabled" specifies to log the elapsed time for testcases. Selecting "Disabled" specifies not to log the elapsed time for testcases. Selecting "Default" specifies to use the STAX service setting for "Log TC Elapsed Time".

- **Log TC Num Starts -** Select "Enabled, "Disabled", or "Default" to indicate whether to log the number of times a testcase is started in the testcase summary records at the end of the STAX Job log and in a LIST TESTCASES request. Selecting "Enabled" specifies to log the number of testcase starts. Selecting "Disabled" specifies not to log the number of testcase starst. Selecting "Default" specifies to use the STAX service setting for "Log TC Num Starts".

- **Log TC Start/Stop -** Select "Enabled, "Disabled", or "Default" to indicate whether to log "Start" and "Stop" level records each time a testcase begins or ends in the STAX Job log. Selecting "Enabled" specifies to log the testcase "Start" and "Stop" records. Selecting "Disabled" specifies not to log the testcase "Start" and "Stop" records. Selecting "Default" specifies to use the STAX service setting for "Log TC Start/Stop".

Following is an example of the "Start Job Parameters" panel with the "Log Options" tab selected:



Log Options

Clear Logs :	○ Enabled	○ Disabled	● Default
Log TC Elapsed Time:	○ Enabled	○ Disabled	● Default
Log TC Num Starts :	○ Enabled	○ Disabled	● Default
Log TC Start/Stop :	○ Enabled	○ Disabled	● Default

Saving Execute Information in a Job Parameters File

You may save the information you specify to start a new job in a Job Parameters File. This is particularly useful when you plan on submitting the job with the parameters specified (or similar parameters) multiple times. To save the job parameters in a file, select "File" from the menu bar on the "Start Job Parameters" panel, and then select "Save As...". A "Save Current Job Parameters as" panel will be displayed which lets you specify the name and directory of the Job Parameters File to create.

Later, to display the "Start Job Parameters" panel with the information you previously stored in a Job Parameters File, select "File" from the menu bar on the "Start Job Parameters" panel, and then select "Open". An "Open Job Parameters File" window will be displayed which lets you specify or browse for the Job Parameters File you wish to use. Or, if the Job Parameters File you wish to use is one of the last ten Job Parameters Files specified, it can be selected directly when you select "File" from the menu bar on the "Start Job Parameters" panel.

If you want to save any changed information for an existing Job Parameters File that you have opened, select "File" from the menu bar on the "Start Job Parameters" panel, and then select "Save".

Buttons on the "Start Job Parameters" Panel

The following buttons are available at the bottom of the "Start Job Parameters" panel:

- **Submit New Job -** click on this button to submit the job for execution. If the execution request is valid, the new job will appear in the "Active Jobs" table on the "STAX Job Monitor" window.

  If you selected "Yes" for the Monitor option, a "STAX Job Monitor" window for the job is then displayed.

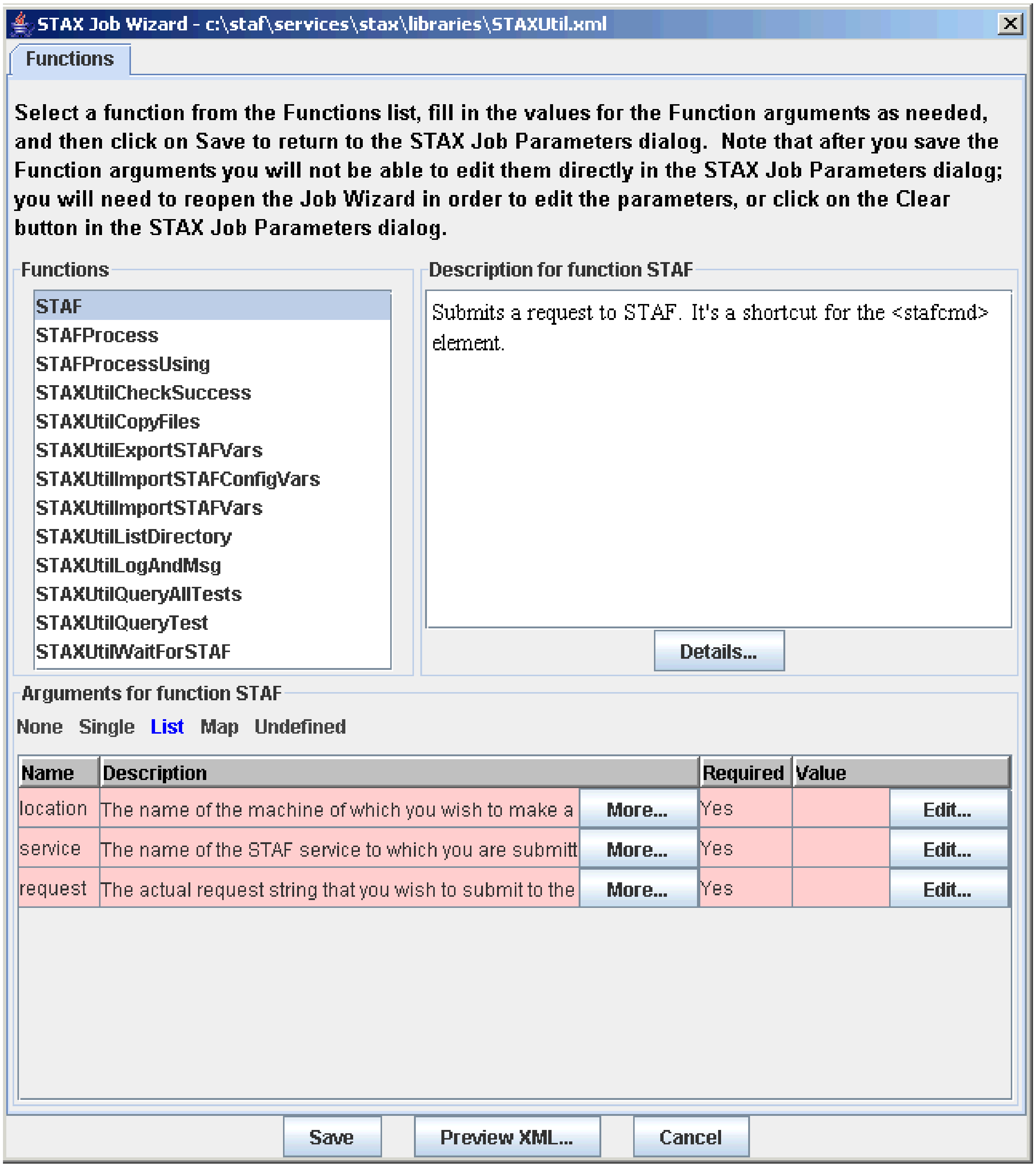
- **Test -** click on this button to test whether the XML document is well-formed and valid and that the specified execution options are valid. The job will not be submitted for execution.
- **Job Wizard -** click on this button to use the Job Wizard to help you select the starting function and to specify function arguments for it. See the "Using the Job Wizard" section for more information on how to use the Job Wizard.
- **Close -** click on this button if you want to close the "STAX Job Parameters" window without submitting the job.
- **Clear all Parameters -** click on this button to clear all of the job execution parameters.

Using the Job Wizard

The STAX Monitor provides a Job Wizard to help you select the starting function for a job you want to execute and to help you specify function arguments for it, if needed. The Job Wizard lists all of the available functions in a given STAX XML file, as well as the arguments that can be specified for each function.

To use the Job Wizard, specify the fully-qualified name of the XML file to be executed in the "XML Job File" section of the "STAX Job Parameters" dialog (and the machine where it resides if not on the local machine), and then click on the "Job Wizard..." button.

Here is an example of the "STAX Job Wizard" dialog that will be displayed (when specifying C:\STAF\services\libraries\STAXUtil.xml as the XML filename).



STAX Job Wizard - c:\staf\services\stax\libraries\STAXUtil.xml

Functions

Select a function from the Functions list, fill in the values for the Function arguments as needed, and then click on Save to return to the STAX Job Parameters dialog. Note that after you save the Function arguments you will not be able to edit them directly in the STAX Job Parameters dialog; you will need to reopen the Job Wizard in order to edit the parameters, or click on the Clear button in the STAX Job Parameters dialog.

Functions

- STAF
- STAFProcess
- STAFProcessUsing
- STAXUtilCheckSuccess
- STAXUtilCopyFiles
- STAXUtilExportSTAFVars
- STAXUtilImportSTAFConfigVars
- STAXUtilImportSTAFVars
- STAXUtilListDirectory
- STAXUtilLogAndMsg
- STAXUtilQueryAllTests
- STAXUtilQueryTest
- STAXUtilWaitForSTAF

Description for function STAF

Submits a request to STAF. It's a shortcut for the <stafcmd> element.

Arguments for function STAF

None Single **List** Map Undefined

Name	Description		Required	Value	
location	The name of the machine of which you wish to make a	More...	Yes		Edit...
service	The name of the STAF service to which you are submitt	More...	Yes		Edit...
request	The actual request string that you wish to submit to the	More...	Yes		Edit...

In the "Functions" list you will see an alphabetical list of all the functions that are available within the specified XML file. If the XML file designates a default starting function via the <defaultcall> element, then the default function will be shown as "functionname (default)" in the list.

The function that is selected by default in the "Functions" list when the "STAX Job Wizard" dialog is first displayed is determined as follows:

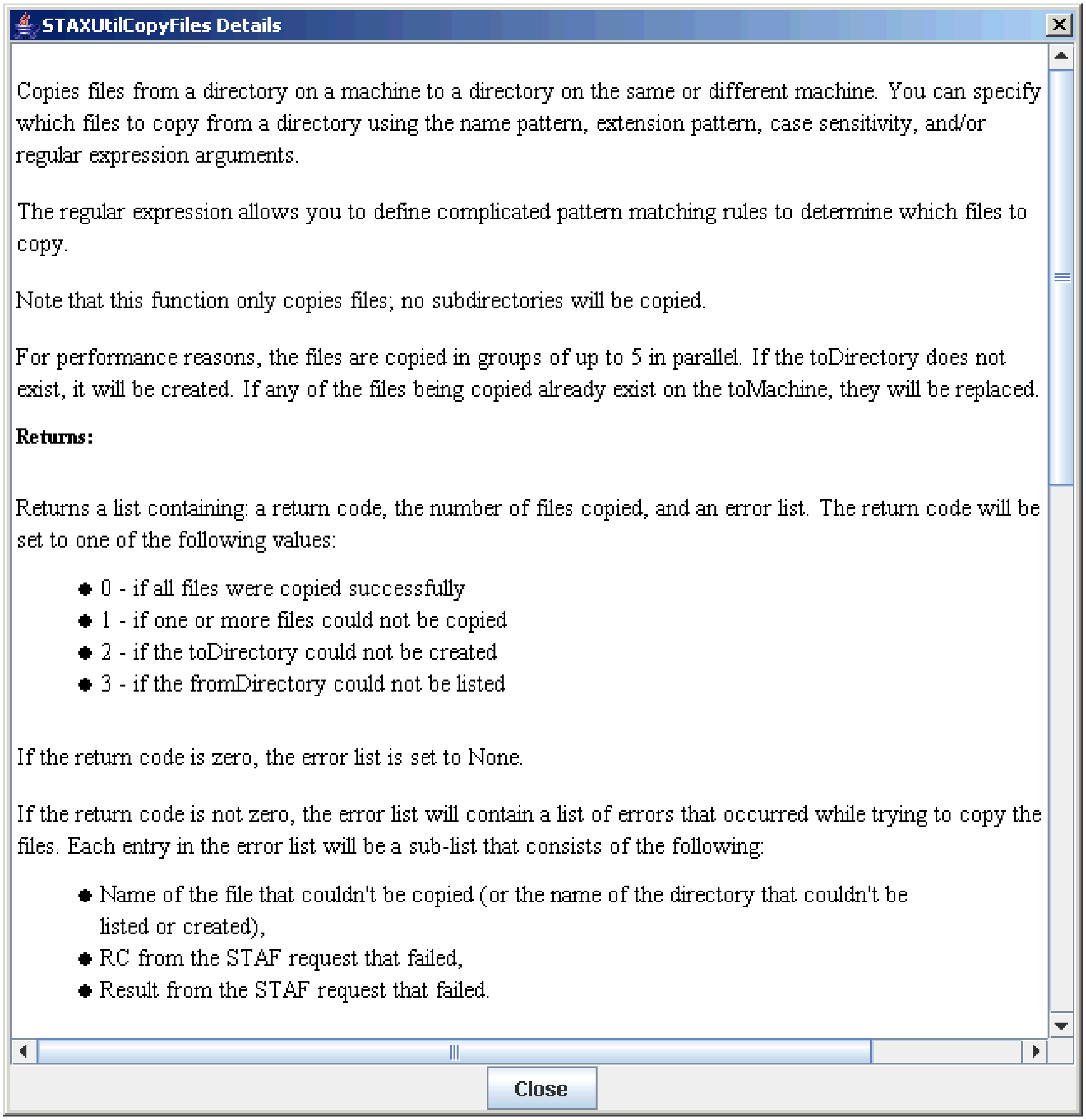
- If "default" was selected for the "Start Function" on the "Function" tab of the "STAX Job Parameters" dialog:
  - If the XML file designates a default starting function via the <defaultcall> element, the default function is selected.

    - Otherwise, if the XML file does not contain a <defaultcall> element, the first function in the list is selected.
- Otherwise, if "other" is selected for the "Start Function" with a valid function name, that function is selected.

Select the function you want to be the starting function for the job. When you select a function from the list of functions, the "Description for function <Function Name>" and "Arguments for function <Function Name>" sections of the "STAX Job Wizard" dialog will be updated within the description and arguments for the selected function.

The "Description for function <Function Name>" section to the right of the list of functions displays additional information for that function if a <function-prolog> or <function-description> element is provided for that function in the specified XML file.

If you click on the "Details..." button in the "Description for function <Function Name>" section, a new dialog will be shown which includes not only the information from the function's <function-prolog>/<function-description> element, but also the information from the function's <function-epilog> element, if provided in the specified XML file.



Copies files from a directory on a machine to a directory on the same or different machine. You can specify which files to copy from a directory using the name pattern, extension pattern, case sensitivity, and/or regular expression arguments.

The regular expression allows you to define complicated pattern matching rules to determine which files to copy.

Note that this function only copies files; no subdirectories will be copied.

For performance reasons, the files are copied in groups of up to 5 in parallel. If the toDirectory does not exist, it will be created. If any of the files being copied already exist on the toMachine, they will be replaced.

Returns:

Returns a list containing: a return code, the number of files copied, and an error list. The return code will be set to one of the following values:

- 0 - if all files were copied successfully
- 1 - if one or more files could not be copied
- 2 - if the toDirectory could not be created
- 3 - if the fromDirectory could not be listed

If the return code is zero, the error list is set to None.

If the return code is not zero, the error list will contain a list of errors that occurred while trying to copy the files. Each entry in the error list will be a sub-list that consists of the following:

- Name of the file that couldn't be copied (or the name of the directory that couldn't be listed or created),
- RC from the STAF request that failed,
- Result from the STAF request that failed.

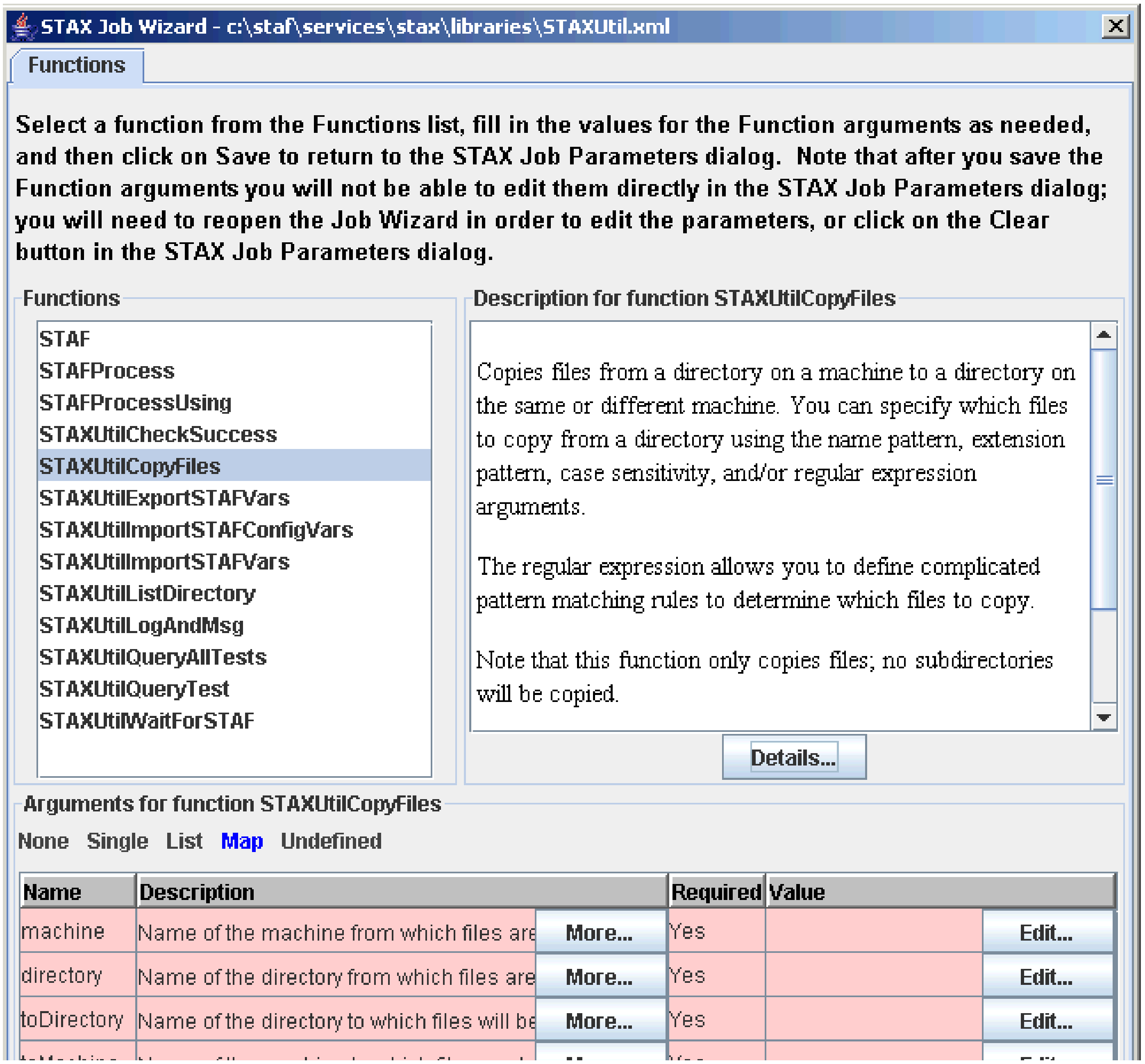
The lower part of the Job Wizard dialog shows the arguments that can be specified for the selected function.

Arguments for the selected function will be one of the following types (the type will be highlighted in blue in the Job Wizard):

- None - No arguments may be specified
- Single - A single argument may be specified
- List - A list of arguments may be specified
- Map - A map of arguments may be specified
- Undefined - The function has defined no arguments (but the STAXArg argument may be specified)

When the "STAF" function is selected (as shown in the previous "STAX Job Wizard" dialog), it shows that it accepts a "List" of three required arguments.

The following "STAX Job Wizard" panel would be shown if you selected the "STAXUtilCopyFiles" function from the "Functions" list. It's "Arguments for function <Function Name>" section shows that it accepts a "Map" of arguments, both required and optional arguments.


STAX Job Wizard - c:\staf\services\stax\libraries\STAXUtil.xml
Functions

Select a function from the Functions list, fill in the values for the Function arguments as needed, and then click on Save to return to the STAX Job Parameters dialog. Note that after you save the Function arguments you will not be able to edit them directly in the STAX Job Parameters dialog; you will need to reopen the Job Wizard in order to edit the parameters, or click on the Clear button in the STAX Job Parameters dialog.

Functions

- STAF
- STAFProcess
- STAFProcessUsing
- STAXUtilCheckSuccess
- STAXUtilCopyFiles
- STAXUtilExportSTAFVars
- STAXUtilImportSTAFConfigVars
- STAXUtilImportSTAFVars
- STAXUtilListDirectory
- STAXUtilLogAndMsg
- STAXUtilQueryAllTests
- STAXUtilQueryTest
- STAXUtilWaitForSTAF

Description for function STAXUtilCopyFiles

Copies files from a directory on a machine to a directory on the same or different machine. You can specify which files to copy from a directory using the name pattern, extension pattern, case sensitivity, and/or regular expression arguments.

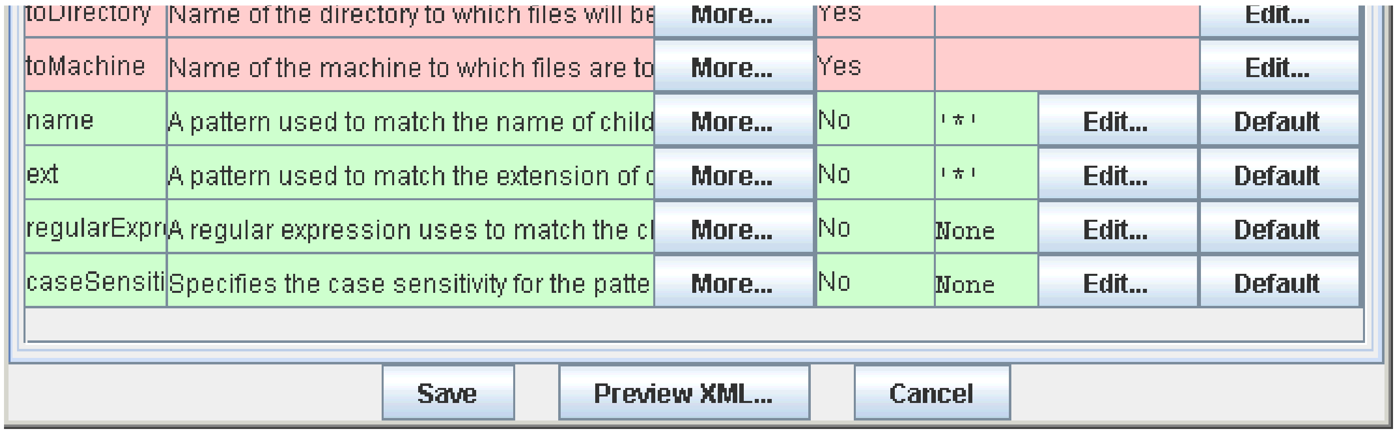
The regular expression allows you to define complicated pattern matching rules to determine which files to copy.

Note that this function only copies files; no subdirectories will be copied.

Arguments for function STAXUtilCopyFiles

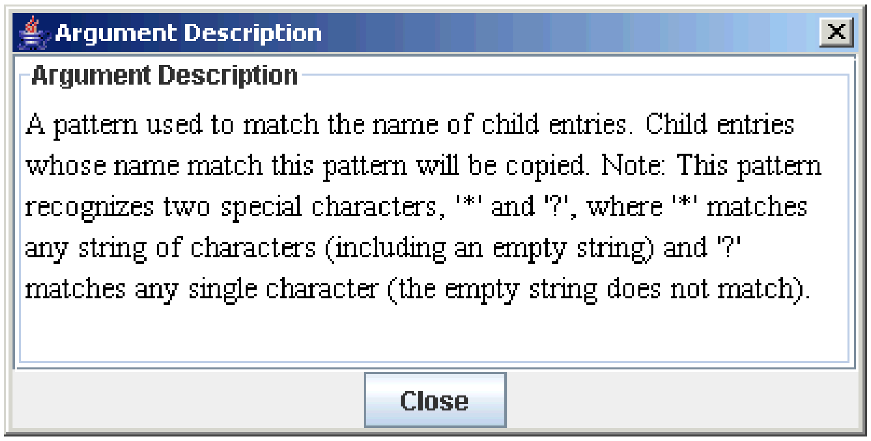
None Single List **Map** Undefined

Name	Description		Required	Value	
machine	Name of the machine from which files are	More...	Yes		Edit...
directory	Name of the directory from which files are	More...	Yes		Edit...
toDirectory	Name of the directory to which files will be	More...	Yes		Edit...



toDirectory	Name of the directory to which files will be	More...	Yes			Edit...
toMachine	Name of the machine to which files are to	More...	Yes			Edit...
name	A pattern used to match the name of child	More...	No	'*'	Edit...	Default
ext	A pattern used to match the extension of c	More...	No	'*'	Edit...	Default
regularExpr	A regular expression uses to match the cl	More...	No	None	Edit...	Default
caseSensiti	Specifies the case sensitivity for the patte	More...	No	None	Edit...	Default

If the Description for an argument does not fit in the column, it will end with "..." and the full description can be viewed by clicking the "More..." button.

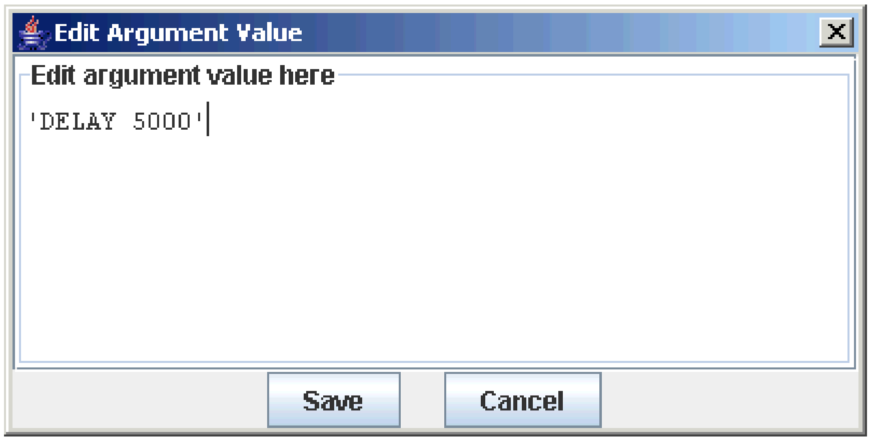


Argument Description

A pattern used to match the name of child entries. Child entries whose name match this pattern will be copied. Note: This pattern recognizes two special characters, '*' and '?', where '*' matches any string of characters (including an empty string) and '?' matches any single character (the empty string does not match).

Required arguments will initially have a red background. After a value has been specified for the required argument, its background will change to green. Optional arguments will have a green background.

To specify a value for a parameter, you may either click on the Edit... button for the argument, which will display a multi-line input dialog, or click in the Value column for the argument and type in the single-line value (note that you must press Enter to save the changes).



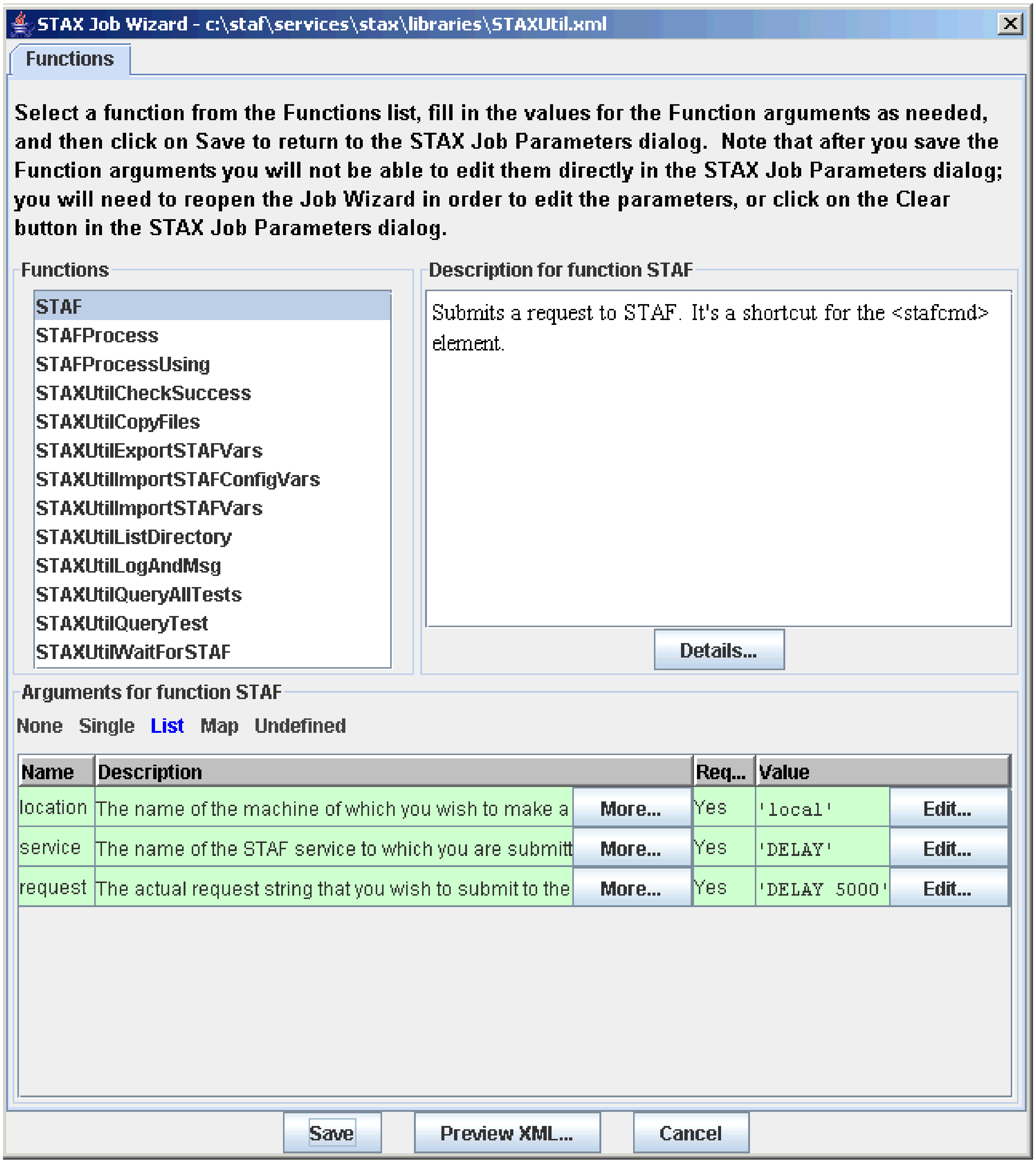
Edit argument value here

```
'DELAY 5000'
```

If the default value for an optional argument has been changed, and you wish to restore the default value, click on the "Default" button.

After you have filled in all of the required arguments (and modified the values of any optional arguments), all the arguments will have a green background.

Here's the "STAX Job Wizard" panel with function "STAF" selected after values have been entered for all of its required arguments:


STAX Job Wizard - c:\staf\services\stax\libraries\STAXUtil.xml
Functions

Select a function from the Functions list, fill in the values for the Function arguments as needed, and then click on Save to return to the STAX Job Parameters dialog. Note that after you save the Function arguments you will not be able to edit them directly in the STAX Job Parameters dialog; you will need to reopen the Job Wizard in order to edit the parameters, or click on the Clear button in the STAX Job Parameters dialog.

Functions

- STAF
- STAFProcess
- STAFProcessUsing
- STAXUtilCheckSuccess
- STAXUtilCopyFiles
- STAXUtilExportSTAFVars
- STAXUtilImportSTAFConfigVars
- STAXUtilImportSTAFVars
- STAXUtilListDirectory
- STAXUtilLogAndMsg
- STAXUtilQueryAllTests
- STAXUtilQueryTest
- STAXUtilWaitForSTAF

Description for function STAF

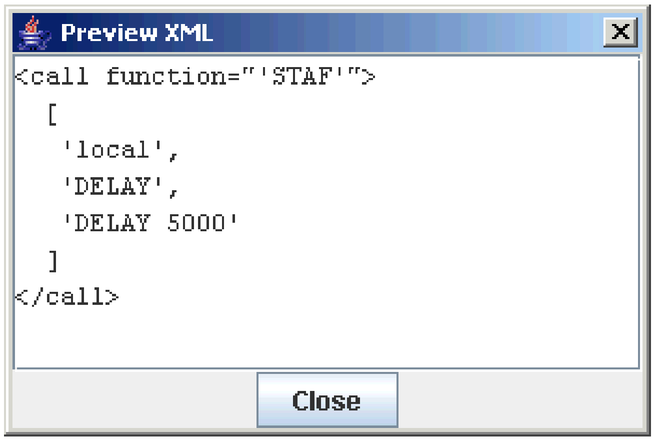
Submits a request to STAF. It's a shortcut for the <stafcmd> element.

Arguments for function STAF

None Single **List** Map Undefined

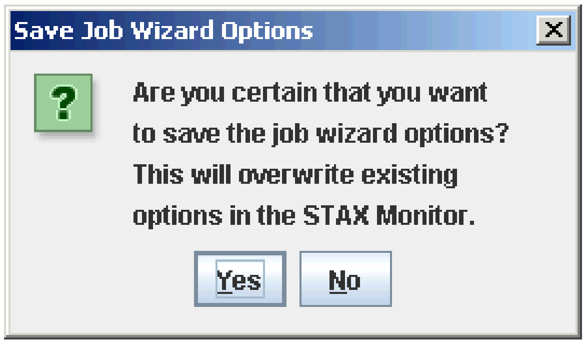
Name	Description		Req...	Value	
location	The name of the machine of which you wish to make a	More...	Yes	'local'	Edit...
service	The name of the STAF service to which you are submitt	More...	Yes	'DELAY'	Edit...
request	The actual request string that you wish to submit to the	More...	Yes	'DELAY 5000'	Edit...

To see an example of the XML syntax for calling the selected function, click on the "Preview XML..." button.



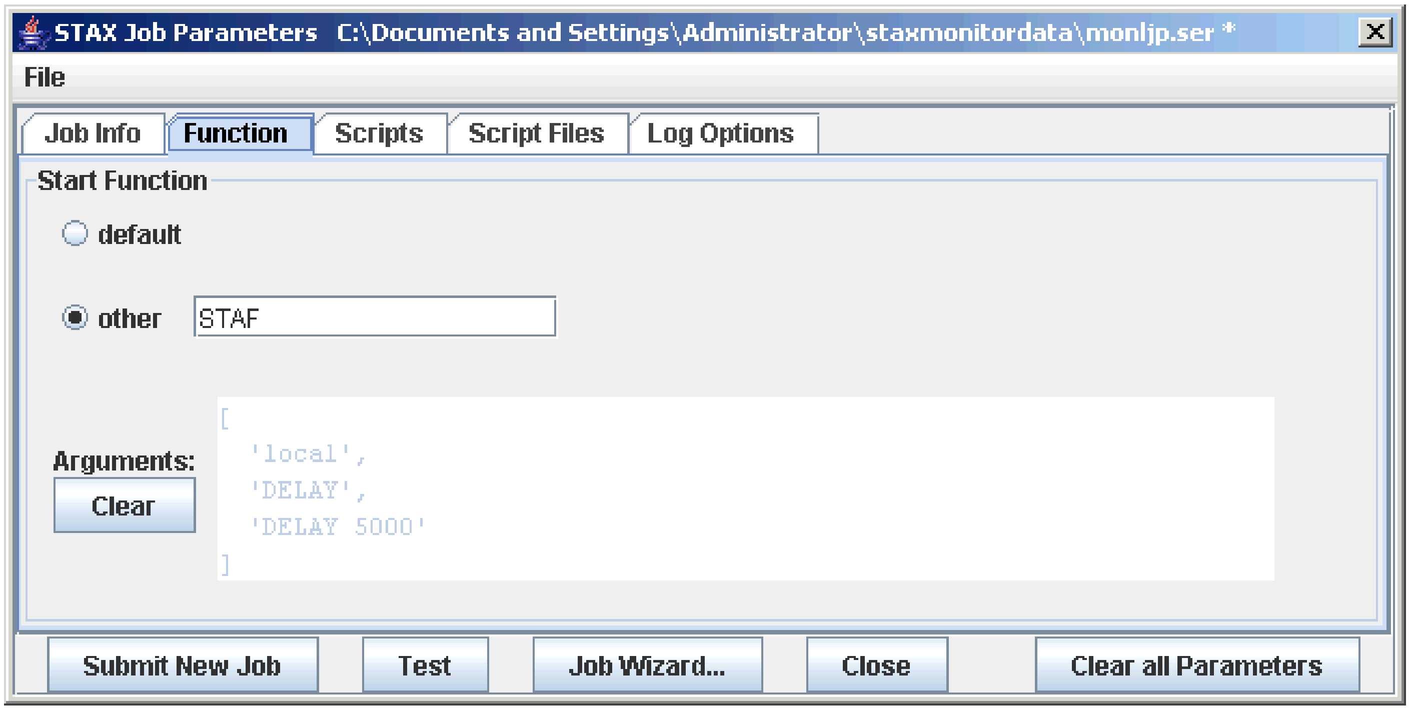
```
<call function="'STAF'">
  [
   'local',
   'DELAY',
   'DELAY 5000'
  ]
</call>
```

To save the function and argument values you have specified to the STAX Job Parameters dialog, click on the "Save" button. A confirmation dialog will be displayed.



Are you certain that you want to save the job wizard options? This will overwrite existing options in the STAX Monitor.

The values you specified will be saved in the STAX Job Parameters dialog.



other: STAF

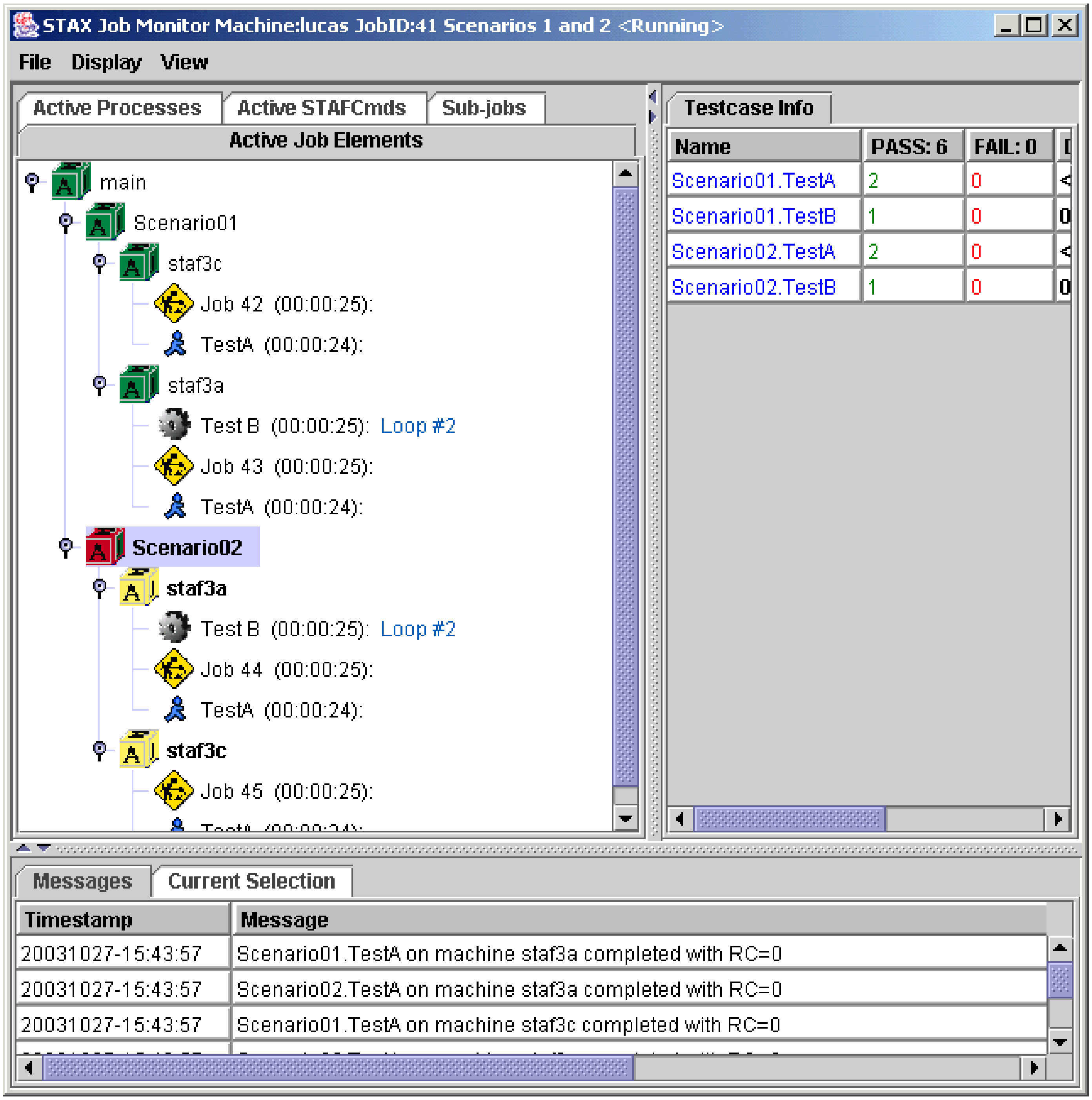
Arguments:

```
[
  'local',
  'DELAY',
  'DELAY 5000'
]
```

Note that after you save the arguments in the Job Wizard, you will no longer be able to directly edit the arguments in the STAX Job Parameters dialog. If you re-open the Job Wizard, all the arguments you specified will automatically be filled in, and you can modify them. If you wish to update the arguments in the STAX Job Parameters dialog, you must click on the "Clear" button and manually specify the arguments.

Monitoring a Job

The following is an example of the "STAX Job Monitor" window for a job:



```
main
  Scenario01
    staf3c
      Job 42  (00:00:25):
      TestA  (00:00:24):
    staf3a
      Test B  (00:00:25):  Loop #2
      Job 43  (00:00:25):
      TestA  (00:00:24):
  Scenario02
    staf3a
      Test B  (00:00:25):  Loop #2
      Job 44  (00:00:25):
      TestA  (00:00:24):
    staf3c
      Job 45  (00:00:25):
```

Name	PASS: 6	FAIL: 0
Scenario01.TestA	2	0
Scenario01.TestB	1	0
Scenario02.TestA	2	0
Scenario02.TestB	1	0

Timestamp	Message
20031027-15:43:57	Scenario01.TestA on machine staf3a completed with RC=0
20031027-15:43:57	Scenario02.TestA on machine staf3a completed with RC=0
20031027-15:43:57	Scenario01.TestA on machine staf3c completed with RC=0

The **File** menu bar contains the following menu items:

- **Exit -** This option is always enabled. Selecting this option closes the STAX Job Monitor window.

The **Display** menu bar contains the following menu items:

- **Display Job Log -** This option is always enabled. Selecting this option causes the Job Log to be displayed.
- **Display Job User Log -** This option is always enabled. Selecting this option causes the Job User Log to be displayed. If

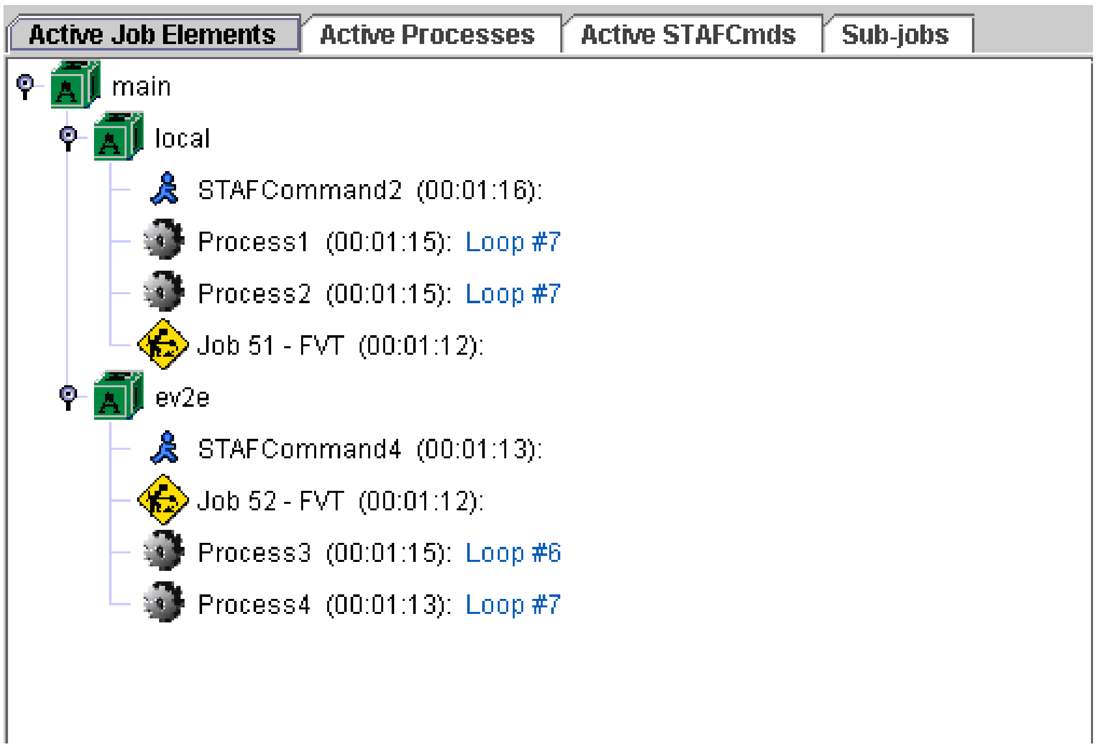
there are no entries in the Job User Log, then an informational message will be displayed to indicate that there are no entries in the log.

The **View** menu bar contains the following menu items, in three sections, where each section represents the panel where the item will be displayed. The first section corresponds to tabs that will show up in the top-left panel. The second section corresponds to tabs that will show up in the top-right panel. The third section corresponds to tabs that will show up in the bottom panel.

- **Active Job Elements**
- **Active Processes**
- **Active STAFCmds**
- **Sub-jobs**
- **TestCase Info**
- **Messages**
- **Current Selection**

The "STAX Job Monitor" window contains seven main tabs:

1. **Active Job Elements**:



```
main
  local
    STAFCommand2 (00:01:16):
    Process1 (00:01:15): Loop #7
    Process2 (00:01:15): Loop #7
    Job 51 - FVT (00:01:12):
  ev2e
    STAFCommand4 (00:01:13):
    Job 52 - FVT (00:01:12):
    Process3 (00:01:15): Loop #6
    Process4 (00:01:13): Loop #7
```

This tab displays the currently executing <block>, <process>, <stafcmd>, and <job> elements for a job in a tree format, in order to show the hierarchy of the currently executing elements.

For each <process>, <stafcmd>, and <job> element represented in the tree, the elapsed time that the process, stafcmd, or job element has been running is displayed to the right of the process, stafcmd, or job name in parenthesis. The format for displaying the elapsed time is HH:MM:SS. If the elapsed time exceeds 99 hours, the format is HHH:MM:SS.

For each <process> element represented in the tree, if the process generates STAF Monitor Service messages (see section 8.8 of the STAF User's Guide), the current monitor message is displayed to the right of the process name and elapsed time. The monitor information is periodically refreshed while the process is running. If the process has not written any STAF Monitor information, the text "No STAF Monitor information available" is displayed.

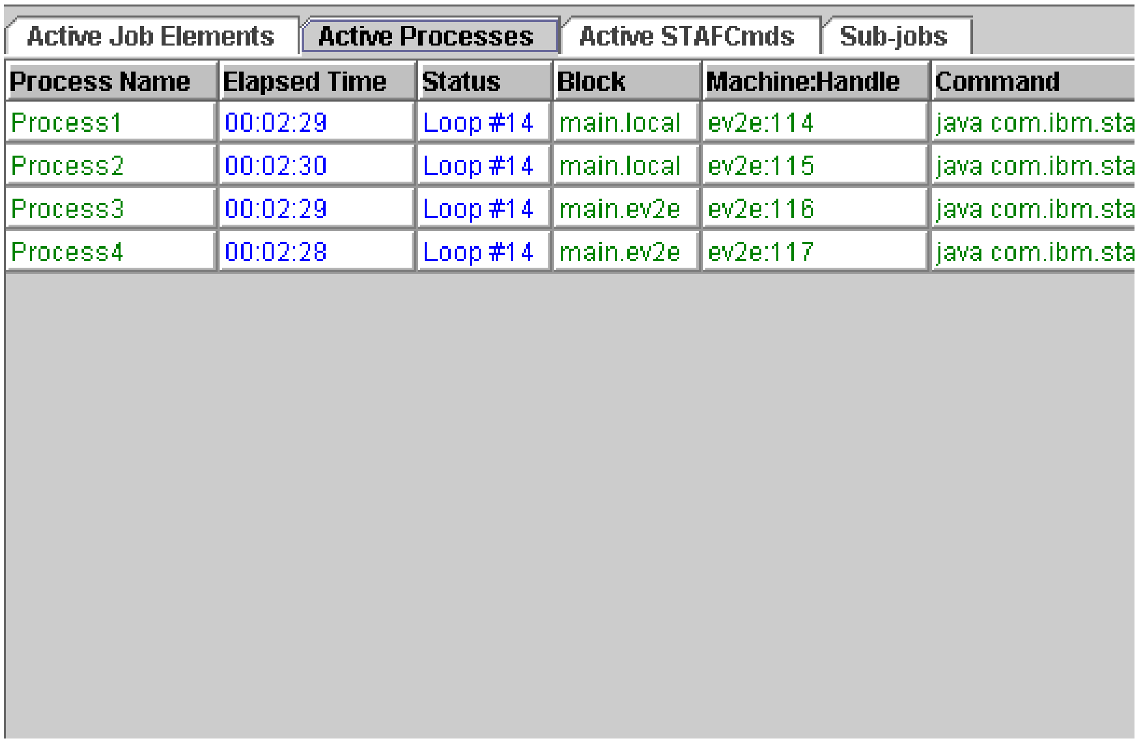
If you right-mouse-click on a block in the "Active Job Elements" panel, a popup menu is displayed with the options "Hold", "Release", and "Terminate" which allows you to control the execution of the block. Holding/Releasing a block which contains sub-jobs will not hold/release the sub-jobs. Terminating a block which contains sub-jobs will terminate the sub-jobs.

The color of the block icon for each block indicates the state of the block:

- A block that is currently running has a green block icon.
- A block that has been held has a red block icon.
- A block that is held by a parent block has a yellow block icon.

If you right-mouse-click on a Sub-job in the "Active Job Elements" panel, a popup menu is displayed with the options "Start Monitoring", "Display Job Log", "Display Job User Log", and "Terminate Job".

2. **Active Processes**:



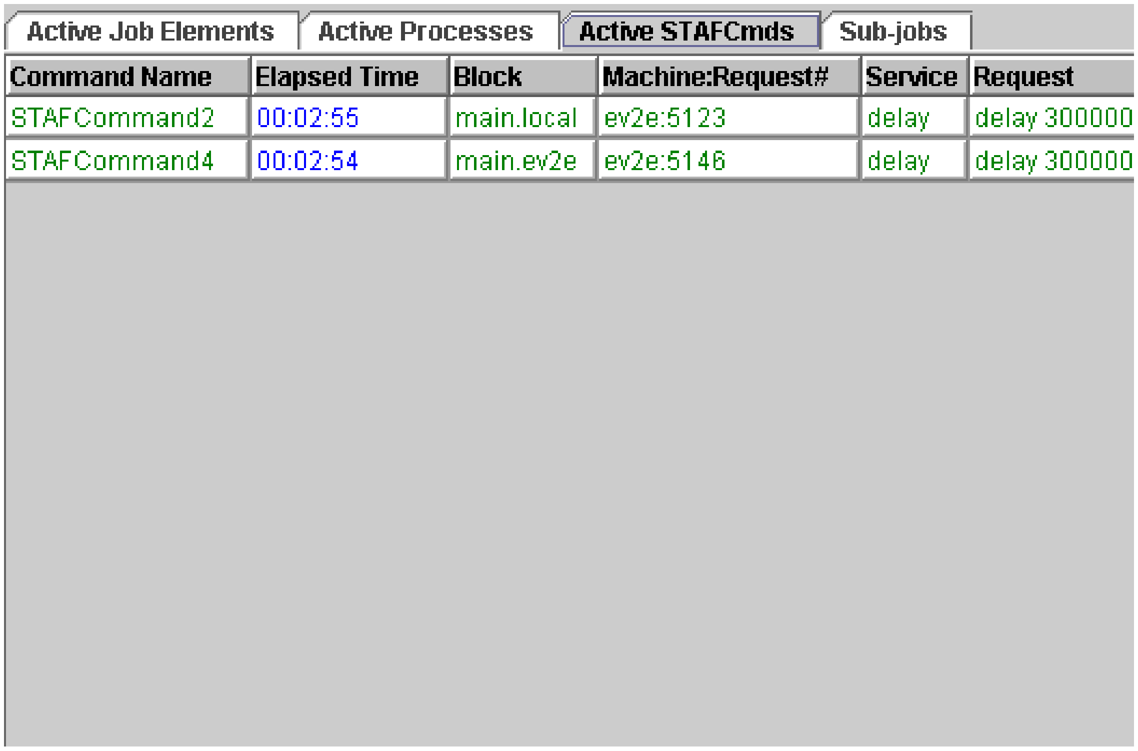
Process Name	Elapsed Time	Status	Block	Machine:Handle	Command
Process1	00:02:29	Loop #14	main.local	ev2e:114	java com.ibm.sta
Process2	00:02:30	Loop #14	main.local	ev2e:115	java com.ibm.sta
Process3	00:02:29	Loop #14	main.ev2e	ev2e:116	java com.ibm.sta
Process4	00:02:28	Loop #14	main.ev2e	ev2e:117	java com.ibm.sta

This tab displays the currently executing <process> elements for a job in a table.

For each <process> the elapsed time (HH:MM:SS) that the process element has been running is displayed in the table. The format for displaying the elapsed time is HH:MM:SS. If the elapsed time exceeds 99 hours, the format is HHH:MM:SS.

For each <process> element represented in the table, if the process generates STAF Monitor Service messages, the current monitor message is displayed in the "Status" column. The monitor information is periodically refreshed while the process is running.

3. **Active STAFCmds**:



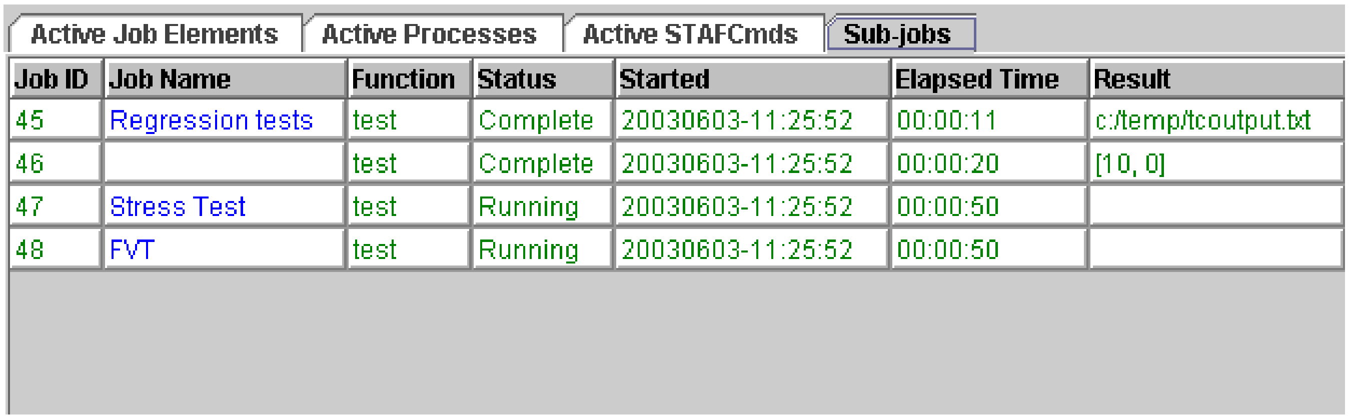
Command Name	Elapsed Time	Block	Machine:Request#	Service	Request
STAFCommand2	00:02:55	main.local	ev2e:5123	delay	delay 300000
STAFCommand4	00:02:54	main.ev2e	ev2e:5146	delay	delay 300000

This tab displays the currently executing <stafcmd> elements for a job in a table.

For each <stafcmd> the elapsed time (HH:MM:SS) that the stafcmd element has been running is displayed in the table. The format for displaying the elapsed time is HH:MM:SS. If the elapsed time exceeds 99 hours, the format is HHH:MM:SS.

Note that this tab is not displayed by default. To view this tab, select the "View" menu bar and then select "Active STAFCmds".

4. **Sub-jobs**:

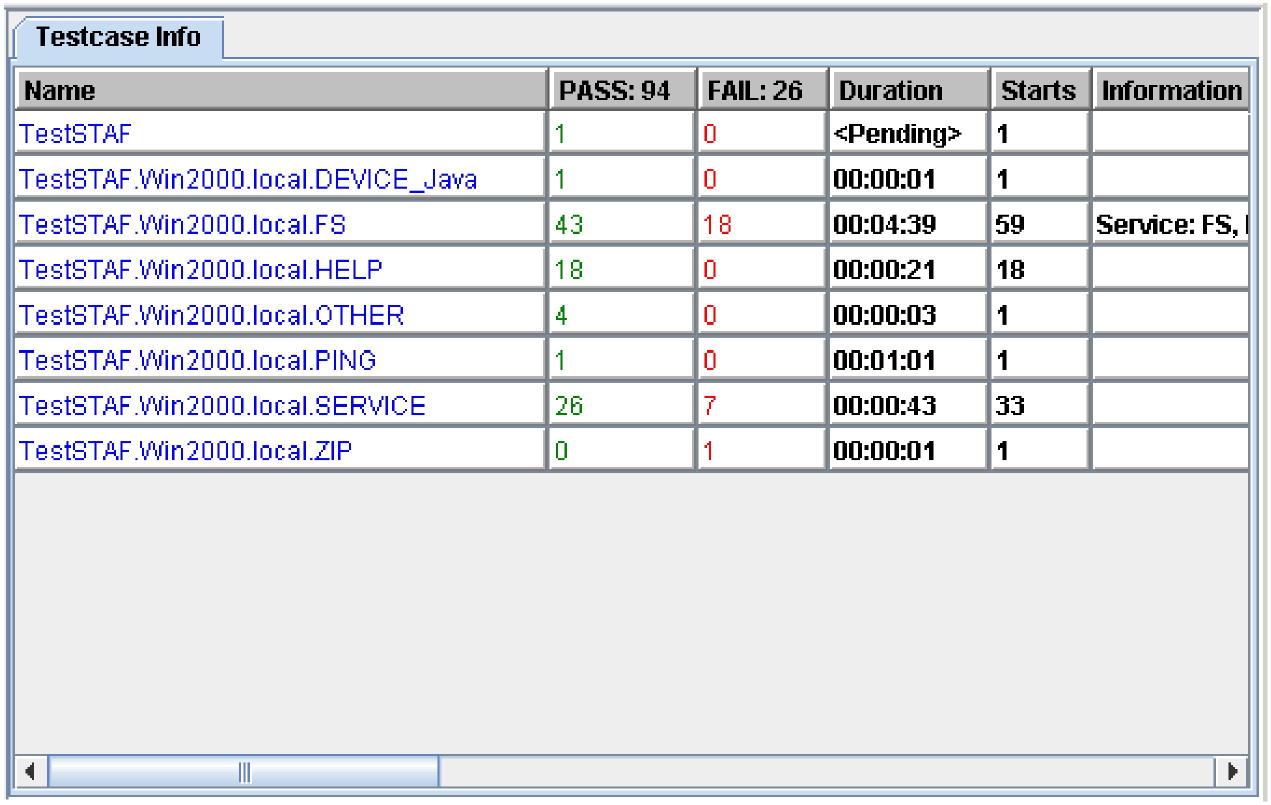


Job ID	Job Name	Function	Status	Started	Elapsed Time	Result
45	Regression tests	test	Complete	20030603-11:25:52	00:00:11	c:/temp/tcoutput.txt
46		test	Complete	20030603-11:25:52	00:00:20	[10, 0]
47	Stress Test	test	Running	20030603-11:25:52	00:00:50	
48	FVT	test	Running	20030603-11:25:52	00:00:50	

This tab displays information about both active and completed sub-jobs.

If you right-mouse-click on a Sub-job in the "Sub-jobs" tab, a popup menu is displayed with the options "Start Monitoring", "Display Job Log", "Display Job User Log", and "Terminate Job".

5. **TestCase Info**:



Name	PASS: 94	FAIL: 26	Duration	Starts	Information
TestSTAF	1	0	<Pending>	1	
TestSTAF.Win2000.local.DEVICE_Java	1	0	00:00:01	1	
TestSTAF.Win2000.local.FS	43	18	00:04:39	59	Service: FS, I
TestSTAF.Win2000.local.HELP	18	0	00:00:21	18	
TestSTAF.Win2000.local.OTHER	4	0	00:00:03	1	
TestSTAF.Win2000.local.PING	1	0	00:01:01	1	
TestSTAF.Win2000.local.SERVICE	26	7	00:00:43	33	
TestSTAF.Win2000.local.ZIP	0	1	00:00:01	1	

This tab displays the following information for each testcase:

- Name of each <testcase> element that has been executed
- Cummulative number of its passes from when the job started executing
- Cummulative number of its fails from when the job started executing
- Cummulative duration (e.g. committed elapsed time) for the testcase since the beginning of the job. The format is HH:MM:SS (or HHH:MM:SS if over 99 hours) or <Pending> if the testcase has not yet ended at least once yet via a </testcase> or STOP TESTCASE request.

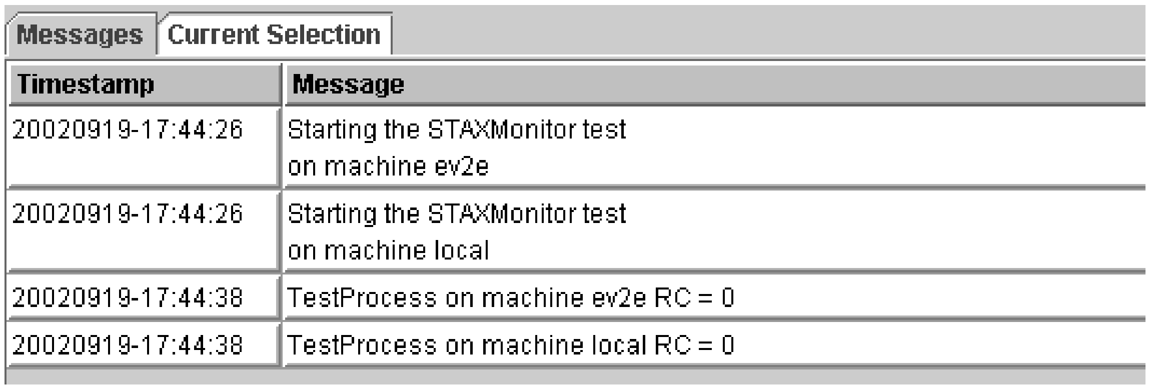
  If a testcase is started more than once via a <testcase> element or a START TESTCASE request, the elapsed times for each testcase started are accumulated. So, it is possible that the elapsed time for a testcase is more than the elapsed time for a job if the same testcase is run in parallel multiple times.

  If you start monitoring an existing job and "Log TC Elapsed Time" was not enabled for the job, the duration for testcases which have already been started appears as ??:??:?? until a status for the testcase is recorded (or the testcase ends).
- Number of times the testcase has been started since the beginning of the job (via a <testcase> element or via a START TESTCASE request or via UPDATE TESTCASE FORCE request)

  If you start monitoring an existing job and "Log TC Num Starts" was not enabled for the job, the number of starts for testcases which have already been started appears as ? until a status for the testcase is recorded (or the testcase ends).
- Last status message if provided via a <tcstatus> element or via an UPDATE TESTCASE request.

Note that at least one testcase status (pass or fail) must be recorded via a <tcstatus> element in order for a testcase to appear in the "Testcase Information" panel (if the testcase's mode is not set to 'strict').

6. **Messages**:

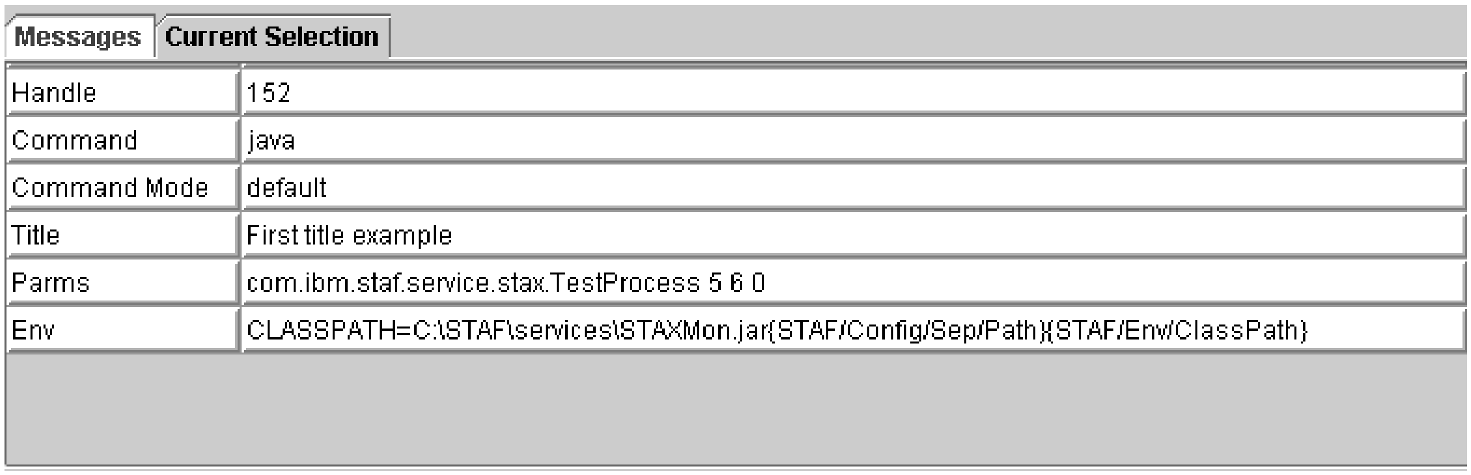


Messages | Current Selection

Timestamp	Message
20020919-17:44:26	Starting the STAXMonitor test on machine ev2e
20020919-17:44:26	Starting the STAXMonitor test on machine local
20020919-17:44:38	TestProcess on machine ev2e RC = 0
20020919-17:44:38	TestProcess on machine local RC = 0

This tab displays the messages (and their timestamps) from each message sent by a <message> element in the XML document. The STAX service can also generate messages via its default signal handlers.

7. **Current Selection**:



Messages | Current Selection

Handle	152
Command	java
Command Mode	default
Title	First title example
Parms	com.ibm.staf.service.stax.TestProcess 5 6 0
Env	CLASSPATH=C:\STAF\services\STAXMon.jar{STAF/Config/Sep/Path}{STAF/Env/ClassPath}

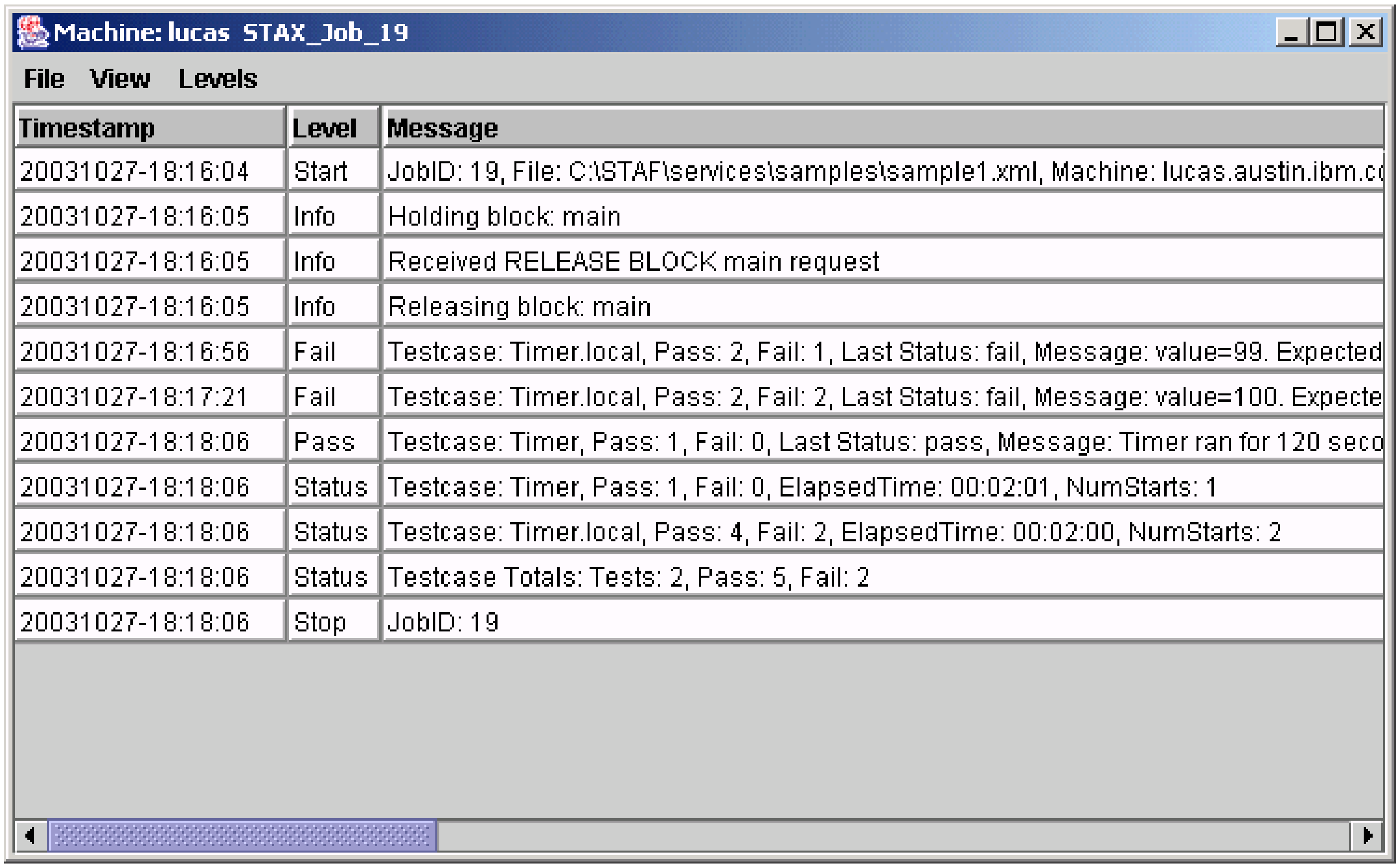
This tab shows more details about an element displayed in the "Active Job Elements", "Active Processes", "Active STAFCmds", or "Sub-jobs" panels when you click on that element in the panel.

Notes:

- When you start monitoring a job that is already executing, the "Active Job Elements", "Active Processes", "Active STAFCmds", and "TestCase Info" tabs will show the same information as if you had been monitoring the job from its beginning. However, the "Messages" tab only displays messages that have occurred **after** you started monitoring the job.
- All of the tables in the Job Monitor are sortable:
  - To sort the table in ascending order by a column, click on the table's column header for that column.
  - To sort the table in descending order by a column, hold the Shift key while clicking on the table's column header for that column.

Displaying a Job Log

There are several menu options to display Job Logs. Here is an example of the dialog that is shown when a Log is displayed.



Timestamp	Level	Message
20031027-18:16:04	Start	JobID: 19, File: C:\STAF\services\samples\sample1.xml, Machine: lucas.austin.ibm.c
20031027-18:16:05	Info	Holding block: main
20031027-18:16:05	Info	Received RELEASE BLOCK main request
20031027-18:16:05	Info	Releasing block: main
20031027-18:16:56	Fail	Testcase: Timer.local, Pass: 2, Fail: 1, Last Status: fail, Message: value=99. Expected
20031027-18:17:21	Fail	Testcase: Timer.local, Pass: 2, Fail: 2, Last Status: fail, Message: value=100. Expecte
20031027-18:18:06	Pass	Testcase: Timer, Pass: 1, Fail: 0, Last Status: pass, Message: Timer ran for 120 seco
20031027-18:18:06	Status	Testcase: Timer, Pass: 1, Fail: 0, ElapsedTime: 00:02:01, NumStarts: 1
20031027-18:18:06	Status	Testcase: Timer.local, Pass: 4, Fail: 2, ElapsedTime: 00:02:00, NumStarts: 2
20031027-18:18:06	Status	Testcase Totals: Tests: 2, Pass: 5, Fail: 2
20031027-18:18:06	Stop	JobID: 19

The **File** menu bar contains the following menu items:

- **Exit** - This option closes the Job Log.

The **View** menu bar contains the following menu items:

- **Refresh** - This option refreshes the Job Log (the latest log information will be queried and the table will be updated).

The **Levels** menu bar contains the following menu items (note that when any of the level options are changed, the information in the table will be refreshed with the latest information from the Log):

- **All** - Selecting this option controls whether all of the log levels will be displayed in the table.
- **Fatal** - Selecting this option controls whether Fatal log messages will be displayed in the table.
- **Error** - Selecting this option controls whether Error log messages will be displayed in the table.
- **Warning** - Selecting this option controls whether Warning log messages will be diplayed in the table.
- **Info** - Selecting this option controls whether Info log messages will be displayed in the table.
- **Trace** - Selecting this option controls whether Trace log messages will be displayed in the table.
- **Trace2** - Selecting this option controls whether Trace2 log messages will be displayed in the table.
- **Trace3** - Selecting this option controls whether Trace3 log messages will be displayed in the table.
- **Debug** - Selecting this option controls whether Debug log messages will be displayed in the table.
- **Debug2** - Selecting this option controls whether Debug2 log messages will be displayed in the table.
- **Debug3** - Selecting this option controls whether Debug3 log messages will be displayed in the table.
- **Start** - Selecting this option controls whether Start log messages will be displayed in the table.
- **Stop** - Selecting this option controls whether Stop log messages will be displayed in the table.
- **Pass** - Selecting this option controls whether Pass log messages will be displayed in the table.
- **Fail** - Selecting this option controls whether Fail log messages will be diplayed in the table.
- **Status** - Selecting this option controls whether Status log messages will be displayed in the table.

- **User1** - Selecting this option controls whether User1 log messages will be displayed in the table.
- **User2** - Selecting this option controls whether User2 log messages will be displayed in the table.
- **User3** - Selecting this option controls whether User3 log messages will be displayed in the table.
- **User4** - Selecting this option controls whether User4 log messages will be displayed in the table.
- **User5** - Selecting this option controls whether User5 log messages will be displayed in the table.
- **User6** - Selecting this option controls whether User6 log messages will be displayed in the table.
- **User7** - Selecting this option controls whether User7 log messages will be displayed in the table.
- **User8** - Selecting this option controls whether User8 log messages will be displayed in the table.

STAX Logging

STAX uses the STAF Log Service for logging, so the Log Service must be registered in order for STAX logs to be created.

There are three types of STAX logs:

- **STAX service log** - This log contains messages generated by the STAX Service for all jobs. This log contains a start and stop message for each STAX Job. The job start message includes additional information about the job such as job ID, XML file name, machine name, starting function, job name, and script parameters, if any.

- **STAX job logs** - There is a STAX job log generated by the STAX Service for each STAX job ID. Each log contains messages for the job such as a start and stop message for the job. In addition, a message is logged if:

  - A job is held, released, or terminated.

  - A signal is raised by the STAX service during job execution and the default signal handler is invoked.

  - A <tcstatus> element specifies a message, it is logged.

  - If "Log TC Start/Stop" is enabled for the job and a <testcase> element is encountered or a START request for a testcase is successfully executed. This logs a message with level "Start" and the format of the message text is:

    ```
    Testcase: <Testcase Name>
    ```

  - If "Log TC Start/Stop" is enabled for the job and a </testcase> element is encountered or a STOP request for a testcase is successfully executed. This logs a message with level "Stop" and the format of the message text is:

    ```
    Testcase: <Testcase Name>, ElapsedTime: <Elapsed Time>"
    ```

  - A job terminates, a summary of the number of times each testcase passed or failed is logged with level "Status". If "Log TC Elapsed Time" is enabled for the job, the elapsed time for each testcase is also logged. If "Log TC Num Starts" is enabled for the job, the number of times each testcase was started is also logged. The format of the message text is:

    ```
    Testcase: <Testcase Name>, Pass: <NumPasses>, Fail: <NumFails>[, ElapsedTime:
    <ElapsedTime>][, NumStarts: <NumStarts>]
    ```

    Notes:

1. If the default mode is specified for a testcase for which no passes or fails were recorded, no summary for that testcase is logged.

2. If a testcase is started more than once via a <testcase> element or a START testcase request, the elapsed times for each testcase started are accumulated. So, it is possible that the elapsed time for a testcase is more than the elapsed time for a job if the same testcase is run in parallel multiple times.

3. The format for elapsed time is HH:MM:SS or if the elapsed time exceeds 99 hours, the format is: HHH:MM:SS.

   - A job terminates, a summary of the job's total number of testcases, passes, and fails is logged.

- **STAX job user logs -** If a STAX job uses the <log> element, then a STAX job user log is generated for the job ID. The messages specified by the <log> elements are written to this log. STAF variables are allowed in the message. However, they will only be resolved if one of the STAF Log service options which specify to resolve variables is set.

Note: If a STAX job contains a <script> element which consists of more than 100K of Python code, you may want to increase the maximum log record size on the STAX service machine. When a STAXPythonEvaluationError signal is raised, a message is written to the STAX job log which includes all the Python code contained in the <script> element where the error occurred, followed by the line number and a description of the problem. If you do not specify a larger maximum record size for the Log service, the message logged will be truncated at 100,000 bytes (the default maximum record size for the Log service). To increase the maximum log record size, see the "Log Service" section in the STAF User's Guide for a description of how to specify a MAXRECORDSIZE parameter when registering the Log Service.

Listing/Querying STAX Service Logs

The STAX logs are machine logs. The names of the STAX logs begin with the STAX service name in uppercase (e.g. STAX). To get a list of all the logs on a STAX service machine named "stax1", use the following STAF log list request. The log files listed whose name begins with STAX are the STAX logs. For example:

```
STAF stax1 LOG LIST MACHINE stax1
```

An example of output from listing STAX logs is:

```
Response
--------
STAX_Service............................  20010718-19:58:22  Size=3109
STAX_Job_1..............................  20010718-19:58:30  Size=3293
STAX_Job_1_User.........................  20010718-15:31:18  Size=2860
STAX_Job_2..............................  20010718-19:58:22  Size=3676
STAX_Job_3..............................  20010718-16:17:06  Size=1924
STAX_Job_3_User.........................  20010718-16:17:06  Size=1064
STAX_Job_4..............................  20010718-21:15:12  Size=656
```

To query a STAX service log, use the following STAF log query machine request (assuming STAX is the registered name for the service):

```
STAF stax1 LOG QUERY MACHINE stax1 LOGNAME STAX_Service
```

An example of the output from a query of a STAX service log is:

```
Response
```

```
--------
20010718-10:30:51|stax1|6|STAX/Job/1|Start|JobID: 1, File: d:/stax/MyTest1.xml,
Machine: machA.austin.ibm.com, Function: Main, JobName: Test1Job
20010718-10:31:00|stax1|9|STAX/Job/2|Start|JobID: 2, File: d:/stax/MyTest2.xml,
Machine: machB.austin.ibm.com, Function: FunctionA, JobName:
20010718-10:31:18|stax1|6|STAX/Job/1|Stop|JobID: 1
20010718-10:31:22|stax1|9|STAX/Job/2|Stop|JobID: 2
20010718-11:16:45|stax1|16|STAX/Job/3|Start|JobID: 3, File: d:/stax/MyTest1.xml,
 Machine: machA.austin.ibm.com, Function: Main, JobName: Test1Job
20010718-11:17:04|stax1|16|STAX/Job/3|Stop|JobID: 3
20010718-14:56:25|stax1|8|STAX/Job/1|Start|JobID: 1, File: d:/stax/J13Test.xml,
Machine: machC.austin.ibm.com, Function: J13auto, JobName:
20010718-14:56:28|stax1|8|STAX/Job/1|Stop|JobID: 1
20010718-14:57:54|stax1|10|STAX/Job/2|Start|JobID: 2, File: d:/stax/Looper.xml,
Machine: machA.austin.ibm.com, Function: MainLoop, JobName: LoopJob
20010718-14:58:21|stax1|10|STAX/Job/2|Stop|JobID: 2
```

Querying STAX Job Logs

To query a STAX job log for Job ID 22, use the following STAF log query machine request (assuming stax1 is the STAX service machine and STAX is the registered name for the service):

```
STAF stax1 LOG QUERY MACHINE stax1 LOGNAME STAX_Job_22
```

An example of the output from a query of a STAX job log is:

```
Response
--------
20031027-18:26:13|stax1|248|STAX/Job/22|Start|JobID: 22, File: C:\STAF\services\
samples\sample1.xml, Machine: lucas.austin.ibm.com, Function: MonitorTest, Args:
 { 'duration': '2m', 'MachList': ['local',], 'STAXJarFile':  '{STAF/Config/STAFR
oot}/services/STAXMon.jar' }, JobName: Sample 1, ScriptFile: c:\tests\pyCode1.py
, ScriptFileMachine: lucas.austin.ibm.com
20031027-18:26:13|stax1|248|STAX/Job/22|Info|Holding block: main
20031027-18:26:14|stax1|248|STAX/Job/22|Info|Received RELEASE BLOCK main request
20031027-18:26:14|stax1|248|STAX/Job/22|Info|Releasing block: main
20031027-18:27:01|stax1|248|STAX/Job/22|Error|STAXProcessStartError signal raise
d. Continuing job.
 with location=machA request=start command :4:java not
ify onend parms :44:com.ibm.staf.service.stax.TestProcess 3 4 99 env :104:CLASSP
ATH=c:\dev\sf\rel\win32\staf\retail/services/STAXMon.jar{STAF/Config/Sep/Path}{S
TAF/Env/ClassPath} sameconsole
RC=16 STAFResult=select() timeout: 0
20031027-18:27:01|stax1|248|STAX/Job/22|Fail|Testcase: Timer.local, Pass: 2, Fai
l: 1, Last Status: fail, Message: value=16. Expected 0.
20031027-18:27:26|stax1|248|STAX/Job/22|Fail|Testcase: Timer.local, Pass: 2, Fai
l: 2, Last Status: fail, Message: value=100. Expected 0.
20031027-18:28:14|stax1|248|STAX/Job/22|Pass|Testcase: Timer, Pass: 1, Fail: 0,
Last Status: pass, Message: Timer ran for 120 seconds
20031027-18:28:14|stax1|248|STAX/Job/22|Status|Testcase: Timer, Pass: 1, Fail: 0
, ElapsedTime: 00:02:00, NumStarts: 1
20031027-18:28:14|stax1|248|STAX/Job/22|Status|Testcase: Timer.local, Pass: 4, F
ail: 2, ElapsedTime: 00:02:00, NumStarts: 2
```

```
20031027-18:28:14|stax1|248|STAX/Job/22|Status|Testcase Totals: Tests: 2, Pass:
5, Fail: 2
20031027-18:28:14|stax1|248|STAX/Job/22|Stop|JobID: 22
```

To query a STAX job log for Job ID 2, but just its last "Testcase Totals:" entry, use the following STAF log query machine request (assuming stax1 is the STAX service machine and STAX is the registered name for the service):

```
STAF stax1 LOG QUERY MACHINE stax1 LOGNAME STAX_Job_2 CONTAINS "Testcase Totals:"
LAST 1
```

An example of the output from this query of a STAX job log is:

```
Response
--------
20010718-14:58:21|stax1|10|STAX/Job/2|Status|Testcase Totals: Tests: 3, Pass: 2,
Fail: 7
```

Querying STAX Job User Logs

To query a STAX job user log for Job ID 3, use the following STAF log query machine request (assuming stax1 is the STAX service machine and STAX is the registered name for the service):

```
STAF stax1 LOG QUERY MACHINE stax1 LOGNAME STAX_Job_3_User
```

An example of the output from a query of a STAX job user log is:

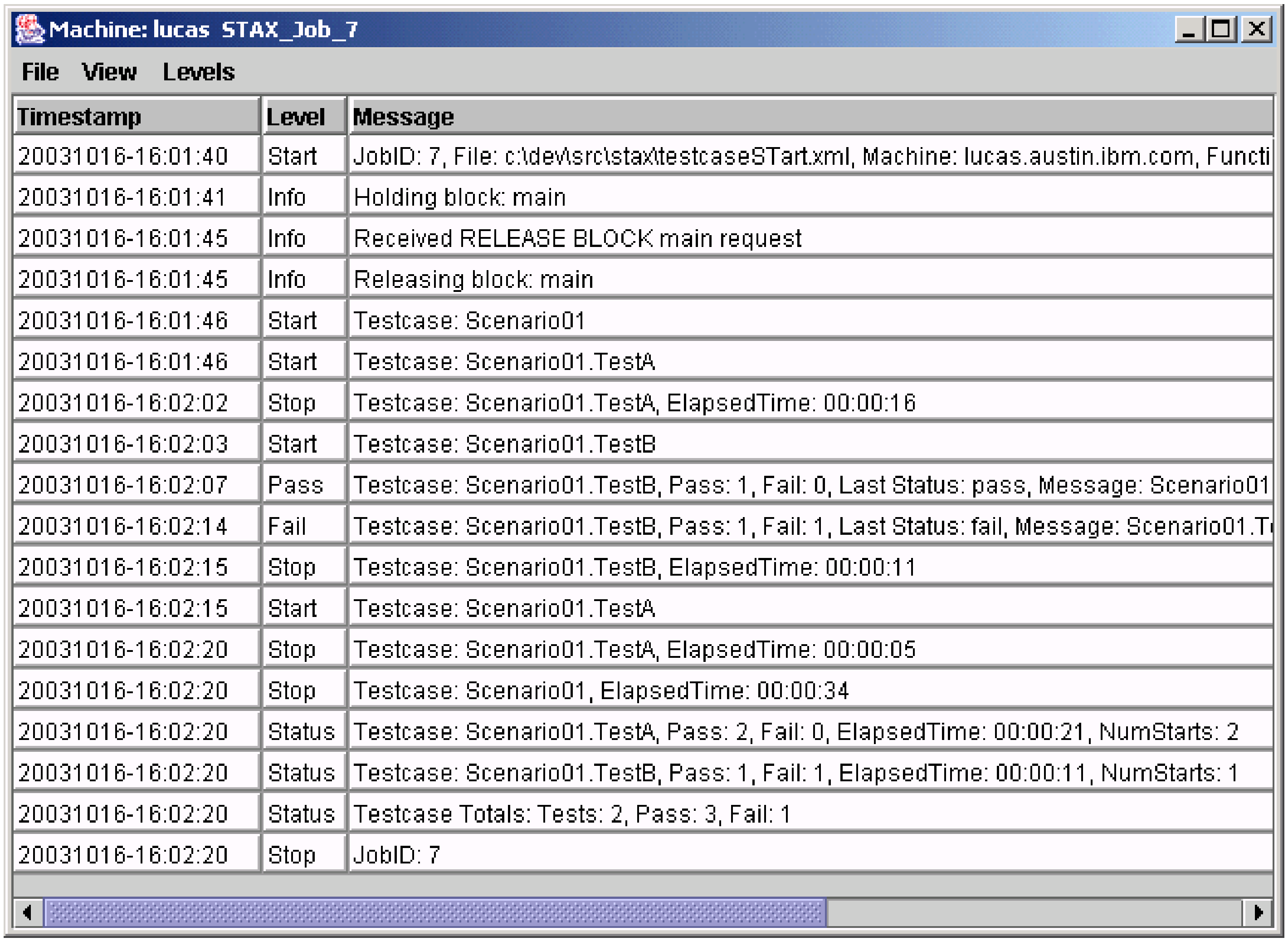
```
Response
--------
20010718-11:16:46|stax1|16|STAX/Job/3|Start|Starting the STAXMonitor test on mac
hine machA
20010718-11:16:51|stax1|16|STAX/Job/3|Info|TestProcess on machine machA RC = 0
20010718-11:16:51|stax1|16|STAX/Job/3|Info|Delaying 1453 on machine machA
20010718-11:16:53|stax1|16|STAX/Job/3|Info|Machine machA is running STAF Version
 2.1.0
20010718-11:16:57|stax1|16|STAX/Job/3|Info|TestProcess on machine machA RC = 1
20010718-11:17:04|stax1|16|STAX/Job/3|Stop|Finished the STAXMonitor Test on mach
ine machA
```

Displaying STAX Logs via a GUI

You can use the STAX Monitor Java application to display STAX logs in a formatted, graphical representation. Refer to the "STAX Monitoring" section for more information on how to display a STAX log.

Enabling/Disabling Testcase Logging

Here's an example of the contents of the STAX Job Log for a job run with "Log TC Elapsed Time", "Log TC Num Starts", and "Log TC Start/Stop" enabled. Note that "Start" and "Stop" records are logged each time a testcase begins and ends and the elapsed time and number of starts for each testcase are provided in the "Status" records written at the end of the job. This also shows how the log is displayed via the STAX Monitor.



Timestamp	Level	Message
20031016-16:01:40	Start	JobID: 7, File: c:\dev\src\stax\testcaseSTart.xml, Machine: lucas.austin.ibm.com, Functi
20031016-16:01:41	Info	Holding block: main
20031016-16:01:45	Info	Received RELEASE BLOCK main request
20031016-16:01:45	Info	Releasing block: main
20031016-16:01:46	Start	Testcase: Scenario01
20031016-16:01:46	Start	Testcase: Scenario01.TestA
20031016-16:02:02	Stop	Testcase: Scenario01.TestA, ElapsedTime: 00:00:16
20031016-16:02:03	Start	Testcase: Scenario01.TestB
20031016-16:02:07	Pass	Testcase: Scenario01.TestB, Pass: 1, Fail: 0, Last Status: pass, Message: Scenario01
20031016-16:02:14	Fail	Testcase: Scenario01.TestB, Pass: 1, Fail: 1, Last Status: fail, Message: Scenario01.T
20031016-16:02:15	Stop	Testcase: Scenario01.TestB, ElapsedTime: 00:00:11
20031016-16:02:15	Start	Testcase: Scenario01.TestA
20031016-16:02:20	Stop	Testcase: Scenario01.TestA, ElapsedTime: 00:00:05
20031016-16:02:20	Stop	Testcase: Scenario01, ElapsedTime: 00:00:34
20031016-16:02:20	Status	Testcase: Scenario01.TestA, Pass: 2, Fail: 0, ElapsedTime: 00:00:21, NumStarts: 2
20031016-16:02:20	Status	Testcase: Scenario01.TestB, Pass: 1, Fail: 1, ElapsedTime: 00:00:11, NumStarts: 1
20031016-16:02:20	Status	Testcase Totals: Tests: 2, Pass: 3, Fail: 1
20031016-16:02:20	Stop	JobID: 7

STAX Variables

The following variables are set in Python during Job Execution by the STAX service and can be referenced by your job definition. However, do not change their values.

- **RC**
  - Description: the return code from a <stafcmd>, <process> element, or <timer> element
  - Assigned: each time a <process>, <stafcmd>, or <timer> element completes
  - Type: numeric. Note that the actual return code from a <process> element (if an error starting the process did not occur) a PyLong numeric type (e.g 0L, 25L). In other cases, the return code will be a PyInteger type.

- **STAFResult**
  - Description: the result from a <stafcmd>, <process>, or <job> element
  - Assigned: each time a <stafcmd>, <process>, or <job> element completes
  - Type: string

- **STAFRC**
  - Description: the alias for the imported com.ibm.staf.STAFResult Java class. It contains constant definitions for STAF return codes (e.g. STAFRC.Ok, STAFRC.DoesNotExist)
  - Assigned: at the beginning of the execution of a STAX job

- Type: Java class com.ibm.staf.STAFResult

- **STAXResult**
  - Description: the result from a function <call>, <call-with-list>, <call-with-map>, <process>, <job>, or <import> element
  - Assigned: each time a <call>, <call-with-list>, <call-with-map>, <process>, <job>, or <import> element completes
  - Type: anything (integer, string, variable, list, nothing, etc.)

- **STAXJobID**
  - Description: the Job ID of the STAX job
  - Assigned: at the beginning of the execution of a STAX job
  - Type: numeric

- **STAXSubJobID**
  - Description: the Job ID of the STAX sub-job (executed via a <job> element)
  - Assigned: at the beginning of the execution of a STAX sub-job
  - Type: numeric

- **STAXJobName**
  - Description: the name of the job specified on the EXECUTE request
  - Assigned: at the beginning of the execution of a STAX job
  - Type: string

- **STAXJobXMLFile**
  - Description: the fully qualified name of the file containing the XML document; this is the value of the FILE parameter, if specified, on the EXECUTE request; otherwise, if DATA <xml data> was specified on the EXECUTE request instead of FILE, it is set to "<inline data>"
  - Assigned: at the beginning of the execution of a STAX job
  - Type: string

- **STAXJobXMLMachine**
  - Description: the name of the machine where the file containing the XML document is located; this is the value of the MACHINE parameter, if specified, on the EXECUTE request; if not specified, then it is the name of the machine submitting the EXECUTE request
  - Assigned: at the beginning of the execution of a STAX job
  - Type: string

- **STAXJobSourceMachine**
  - Description: the name of the machine submitting the EXECUTE request
  - Assigned: at the beginning of the execution of a STAX job
  - Type: string

- **STAXJobSourceHandle**
  - Description: the handle of the process submitting the EXECUTE request
  - Assigned: at the beginning of the execution of a STAX job
  - Type: numeric

- **STAXJobSourceHandleName**
  - Description: the name of the handle of the process submitting the EXECUTE request
  - Assigned: at the beginning of the execution of a STAX job
  - Type: string

- **STAXJobScriptFileMachine**
  - Description: the name of the machine where the script files are located. This is the value of the SCRIPTFILEMACHINE parameter, if specified, on the EXECUTE request. If the SCRIPTFILEMACHINE parameter was not specified on an EXECUTE request, it defaults to the value specified for the the MACHINE option on the EXECUTE request. If the MACHINE option was not specified, it assumes the script file(s) are on the machine submitting the STAX EXECUTE request.
  - Assigned: at the beginning of the execution of a STAX job
  - Type: string

- **STAXJobScriptFiles**
  - Description: a list of names of script files; this is a list of the values of the SCRIPTFILE parameter(s), if specified, on the EXECUTE request; if no SCRIPTFILE parameters are specified, then it is an empty list
  - Assigned: at the beginning of the execution of a STAX job
  - Type: PyArray

- **STAXJobStartDate**
  - Description: the date the job started in format yyyymmdd
  - Assigned: at the beginning of the execution of a STAX job
  - Type: string

- **STAXJobStartTime**
  - Description: the time the job started in format hh:mm:ss
  - Assigned: at the beginning of the execution of a STAX job
  - Type: string

- **STAXJobStartFunctionName**
  - Description: the name of the starting function for the STAX job
  - Assigned: at the beginning of the execution of a STAX job
  - Type: string

- **STAXJobStartFunctionArgs**
  - Description: the arguments passed to the starting function for the STAX job (in string format)
  - Assigned: at the beginning of the execution of a STAX job
  - Type: string

- **STAXJobUserLog**
  - Description: a STAFLog wrapper object for the STAX Job User Log. You can use this object to log to the STAX Job User Log from Python code, instead of using the <log> element, which cannot be included in a <script> element. For example, within a <script> element, could do the following to log a message of level "info" in the STAX Job User Log:

    ```
    res = STAXJobUserLog.log('info', 'Here is an informational message')
    ```

    Note that the log method on this object requires two parameters: the log level string and the message. The log method returns a STAFResult object from which you can reference the return code and the result (e.g. res.rc and res.result).
  - Assigned: at the beginning of the execution of a STAX job
  - Type: STAFLog wrapper object

- **STAXThreadID**
  - Description: the Thread ID currently running
  - Assigned: at the creation of a new Thread. For example, a new thread is created when a job begins execution, when each task element contained in a <parallel> element is executed, and when each iteration of the task element

contained in a <paralleliterate> element is executed.
  - Type: numeric

- **STAXCurrentBlock**
  - Description: the name of the current block running in a thread
  - Assigned: when entering a block (e.g. upon encountering a <block name="..."> element and when entering the default 'main' block).
  - Type: string

- **STAXCurrentTestcase**
  - Description: the name of the current testcase running in a thread
  - Assigned: when entering a testcase (e.g. upon encountering a <testcase name="..."> element). If no testcase is currently running, then it is assigned None (a special object provided by Python which serves as an empty placeholder, much like null in Java).
  - Type: string

- **STAXProcessHandle**
  - Description: the process handle
  - Assigned: when the process has started executing. Only the <process-action> element can use this variable to access the handle for its process.
  - Type: string

- **STAXServiceName**
  - Description: the registered name of the STAX service executing the job
  - Assigned: at the beginning of the execution of a STAX job
  - Type: string

- **STAXServiceMachine**
  - Description: the effective machine name of the STAX service machine (the local machine)
  - Assigned: at the beginning of the execution of a STAX job
  - Type: string

- **STAXEventServiceName**
  - Description: the registered name of the Event service (to which the STAX service submits requests)
  - Assigned: at the beginning of the execution of a STAX job
  - Type: string

- **STAXEventServiceMachine**
  - Description: the machine name of the Event service machine
  - Assigned: at the beginning of the execution of a STAX job
  - Type: string

- **STAXLogTCElapsedTime**
  - Description: a flag indicating if "Log TC Elapsed Time" is enabled or disabled for the STAX job. 1 means enabled; 0 means disabled.
  - Assigned: once at the beginning of the execution of a STAX job
  - Type: numeric (0 or 1)

- **STAXLogTCNumStarts**
  - Description: a flag indicating if "Log TC Num Starts" is enabled or disabled for the STAX job. 1 means enabled; 0 means disabled.
  - Assigned: once at the beginning of the execution of a STAX job
  - Type: numeric (0 or 1)

- **STAXLogTCStartStop**
  - Description: a flag indicating if "Log TC Start/Stop" is enabled or disabled for the STAX job. 1 means enabled; 0 means disabled.
  - Assigned: once at the beginning of the execution of a STAX job
  - Type: numeric (0 or 1)

- **STAXArg**
  - Description: the arguments passed when calling a function that did not define its arguments (via <function-list-args>, <function-map-args>, etc.)
  - Assigned: each time a <call>, <call-with-list>, <call-with-map> element begins if the function being called did not define its arguments
  - Type: anything (integer, string, variable, list, nothing, etc.)

STAX Extensions

You can add extensions to the STAX service. For example, you can write a STAX service extension to define one or more new elements and you can write a STAX monitor extension to define a new plug-in view that can be displayed via the STAX Monitor. See the "STAX Extensions Developer's Guide" for more information on how to write extensions.

Registering STAX Service Extensions

This section discusses how to register STAX service extensions when configuring the STAX service. STAX service extensions allow you to extend the STAX DTD and define additional XML elements that can be used in STAX jobs.

STAX extensions are registered via the PARMS option when configuring the STAX service (either in the STAF.cfg file or dynamically using a SERVICE ADD request). Each STAX extension is provided in a jar file. You can specify the STAX extension jar files that you want to register using an EXTENSIONXMLFILE parameter or using EXTENSION parameters (or using an EXTENSIONFILE parameter, but this parameter has been deprecated).

Syntax

```
SERVICE <Name> LIBRARY JSTAF EXECUTE <STAX Jar File Name>
               [OPTION <Name[=Value]>]...
               [PARMS <"> [EVENTSERVICEMACHINE <EventMachine>]
                          [EVENTSERVICENAME <EventName>]
                          [NUMTHREADS <NumThreads>]
                          [PROCESSTIMEOUT <ProcessTimeout>]
                          [CLEARLOGS <Enabled | Disabled>]
                          [LOGTCELAPSEDTIME <Enabled | Disabled>]
                          [LOGTCNUMSTARTS <Enabled | Disabled>]
                          [LOGTCSTARTSTOP <Enabled | Disabled>]
                          [EXTENSIONXMLFILE <Extension XML File> |
                           EXTENSIONFILE <Extension Text File>]
                          [EXTENSION <Extension Jar File>...
                      <">]
```

`EXTENSIONXMLFILE` specifies the fully-qualified name of an extension XML file that defines all of the STAX extensions to be registered in an XML format. This XML file must conform to the stax-extensions DTD. Refer to the "Creating a STAX Extensions XML File" section for more information on how to create an extension xml file. We recommend using the EXTENSIONXMLFILE parameter to register STAX extensions as it provides the ability to specify parameters for extensions (if the extension supports parameters) and to include or exclude specific elements provided in the extension jar file. This option resolves STAF variables.

`EXTENSION` specifies the fully-qualified name of an extension jar file that defines a STAX extension to be registered. You may specify multiple EXTENSION parameters. You may optionally specify to only register some of the elements provided in the extension jar file by specifying one or more spaces after the jar file name, a #, one or spaces, and then a space separated list of the element names that to be registered. The format is:

```
<Jar File Name> [ # elementName1 elementName2 ...]
```

If no elements are specified when registering the extension, all of the elements with staf/stax/extension/<element> entries in the extension jar file's manifest file will be registered. This option resolves STAF variables.

`EXTENSIONFILE` specifies the fully-qualified name of a text file that contains entries where each line has the same format as described above for the EXTENSION parameter. This parameter has been deprecated as of STAX V1.5.0. Use the EXTENSIONXMLFILE parameter instead. This option resolves STAF variables.

Examples

Goal: Configure the STAX service and register all the STAX extensions specified in an extension xml file named extensions.xml in the services directory off the STAF root directory.

```
SERVICE STAX LIBRARY JSTAF EXECUTE {STAF/Config/STAFRoot}/services/STAX.jar \
        OPTION J2=-Xmx512 \
        PARMS "EXTENSIONXMLFILE {STAF/Config/STAFRoot}/services/extensions.xml"
```

Goal: Configure the STAX service using the EXTENSION option to specify extension jar files C:/STAXExt/ExtDelay.jar and C:/STAXExt/MyExt.jar.

```
SERVICE STAX LIBRARY JSTAF EXECUTE C:/STAF/services/STAX.jar \
        OPTION J2=-Xm512 \
        PARMS "EXTENSION C:/STAXExt/ExtDelay.jar EXTENSION C:/STAXExt/MyExt.jar"
```

Goal: Configure the STAX service using the EXTENSION option to specify extension jar file C:/STAXExt/ExtDelay.jar and to only register the ext-delay and ext-wait elements). Note that double quotes are needed around the value specified for EXTENSION because the value contains spaces. Note also that because these double quotes are within the double quotes specified for the PARMS option, they need to be escaped (using \).

```
SERVICE STAX LIBRARY JSTAF EXECUTE {STAF/Config/STAFRoot}/services/STAX.jar \
        PARMS "EXTENSION \"C:/STAXExt/ExtDelay.jar  # ext-delay ext-wait\""
```

Goal: Configure the STAX service using the EXTENSIONFILE option to specify the name of a text file, extensions.txt, located in the services directory off the STAF root directory. This text file contains the names of extension jar files to be registered.

```
SERVICE STAX LIBRARY JSTAF EXECUTE {STAF/Config/STAFRoot}/services/STAX.jar \
        PARMS "EXTENSIONFILE {STAF/Config/STAFRoot}/services/extensions.txt"
```

Creating a STAX Extensions XML File

A STAX extensions XML file defines the STAX extensions to be registered when configuring the STAX service. This XML file must comply with the STAX Extensions document type definition (DTD) shown in appendix "STAX Extensions DTD". Note that the stax-extensions DTD is provided as part of the STAX zip/tar file (called ext/stax-extensions.dtd).

This section describes how to create a STAX Extensions XML file.

The first line in a STAX Extensions XML file should start with an XML declaration. This indicates the document is written in XML and specifies the XML version, the language encoding for the document, and indicates that the document refers to an external DTD (standalone="no").

The second line in a STAX Extensions XML file should be the document type declaration. This is used to indicate the DTD used for the document. It defines the name of the root element (stax-extensions), and the DTD to be used. STAX checks the syntax of XML documents using a validating XML parser to verify that the document complies with the DTD. Note that DTDs are all about specifying the structure and syntax of XML documents (not their content).

So, the first two lines in a STAX Extensions XML file should look like:

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

stax-extensions

A STAX Extensions XML file must contain a root element which contains all other elements in the document. The root element of a STAX Extensions XML file is **stax-extensions**.

The **stax-extensions** element consists of one or more **extension** elements,

extension

The **extension** element is used to specify a STAX extension to register. All **extension** elements are contained within the root **stax-extensions** element.

The extension element has one attribute:

- **jarfile** - specifies the fully-qualified name of the STAX extension jar file to be registered. This attribute is required. This attribute will resolve STAF variables.

The extension element can optionally contain the following sub-elements:

- **parameter** - specifies the name and value for a parameter to this STAX extension. Any number of parameter elements may be specified for an extension. See the extension provider's documentation for each extension to see what parameters it supports, if any.

  The parameter element has two attributes:

  - **name** - specifies the name of the parameter. It is required.
  - **value** - specifies the value for the parameter. It is required.

- **include-element** - specifies the name of an element supported by the STAX extension jar file to be included. Any number of include-elements may be specified for an extension. However, if an include-element is specified, no exclude-elements can be specified. If one or more include-elements are specified, then only those elements named by each include-element are registered. Any other elements supported by the STAX extension jar file are excluded (not registered).

The include-element has one attribute:

  - **name** - specifies the name of the element to include. It is required.

- **exclude-element** - specifies the name of an element supported by the STAX extension jar file to be excluded (not registered). Any number of exclude-elements may be specified for an extension. However, if an exclude-element is specified, no include-elements can be specified. If one or more exclude-elements are specified, then only those elements supported by the STAX extension jar file that are not named by an exclude-element are registered.

  The exclude-element has one attribute:

  - **name** - specifies the name of the element to exclude. It is required.

Note:

- If no **include-element** or **exclude-element** elements are specified, all of the elements supported by the STAX extension jar file will be registered.

Usage:

The following STAX Extensions XML file defines three STAX extensions. In this example, all three extension elements are empty elements.

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

  <extension jarfile="C:/STAF/services/ExtDelay.jar"/>

  <extension jarfile="C:/STAF/services/ExtMessageText.jar"/>

  <extension jarfile="C:/STAF/services/EmailExt.jar"/>

</stax-extensions>
```

The following STAX Extensions XML file defines four STAX extensions. All three extension files are located in the services directory off the root of the STAF directory.

- The first extension jar file, named ExtMessageText.jar, has no parameters, no include-elements, and no exclude-elements.
- The second extension jar file, named ExtDelay.jar, has one parameter specified named delay and specifies to include all elements supported by this extension jar file except for the ext-wait and ext-sleep elements.
- The third extension jar file, named EmailExt.jar specifies to include just one element named email so that any other elements supported by this extension jar file are excluded (not registered).

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

<stax-extensions>

   <extension jarfile="{STAF/Config/STAFRoot}/services/ExtMessageText.jar"/>
```

```
  <extension jarfile="{STAF/Config/STAFRoot}/services/ExtDelay.jar">
    <parameter name="delay" value="5"/>
    <exclude-element name="ext-wait"/>
    <exclude-element name="ext-sleep"/>
  </extension>

  <extension jarfile="{STAF/Config/STAFRoot}/services/EmailExt.jar">
    <include-element name="email"/>
  </extension>

</stax-extensions>
```

Registering STAX Monitor Extensions

STAX monitor extensions define a new plug-in views that can be displayed via the STAX Monitor. For example, you may want to register a STAX monitor extension that displays a new extension element in the "Active Job Elements" panel. Or you may want to register a STAX monitor extension that displays a new tab in the "Active Job Elements" or "Info" panel.

As of STAX V1.5.0, any STAX monitor extensions that are registered with the STAX service will be automatically made available to the STAX Monitor. You should only specify local extension jar files that are not registered with the STAX service or that contain monitor extensions that you want to override (e.g. with a later version of the extension). See the "Registering STAX Service Extensions" section for more information on how to register STAX extensions with the STAX service.

To view monitor extensions that are registered with the STAX Monitor, from the main "STAX Job Monitor" panel, click on the **File->Properties->Extensions** tab. See the "Extensions" tab section for more information. You can also display the monitor extensions that are registered by specifying the -extensions option when starting the STAX Monitor.

To view, add, or delete a local extension jar file, from the main "STAX Job Monitor" panel, click on the **File->Properties->Extension Jars** tab. See the "Extension Jars" tab section for more information.

Note: The "View" menu for the "STAX Job Monitor" panel does not currently support the ability to de-select registered monitor extensions.

Generating STAX Function Documentation using a XSL StyleSheet

A sample XSL Stylesheet, FunctionList.xsl, is provided to aid in generating documentation about your STAX Functions. It describes how to transform information provided in the <function-prolog element (or the deprecated <function-description element), the <function-epilog> element and the function argument elements into readable documentation. It can be used in conjunction with an XSLT stylesheet processor, such as Xalan, to transform function information provided in a STAX XML document to an HTML document.

The Xalan-Java XSLT processor is available for download from http://xml.apache.org/xalan-j/index.html. Other XSLT processors are also available.

The FunctionList.xsl file can be found in the samples directory from the extracted STAX zip/tar file you downloaded. You may want to customize this stylesheet to generate function documentation in the form that best suites your purposes.

Here is the HTML document generated from the sample1.xml document provided with STAX. This documentation was generated using the Xalan-Java XSLT Processor and the FunctionList.xsl stylesheet. Note that we had to comment out the DOCTYPE line in sample1.xml to avoid an error that STAX.DTD is not available (unless you create it using the STAX GET DTD request and redirect its output to a file).

STAX Function Definitions

MonitorTest

For each machine specified by the MachList argument, function RunProcesses is called and run in parallel. This is done in a continous loop until the time specified by the duration argument is reached.

This function takes an argument map			
Name	**Description**	**Required**	**Default**
duration	Timer duration to run the test. e.g. '5m', '1h', '90s', etc.	No	'2m'
MachList	List of machines where the test will be run	No	['local', 'local']
STAXJarFile	Fully-qualified name of STAX jar file on each machine where the test will be run	No	STAFJarFile

---

RunProcesses

This function runs multiple processes. Each process runs a Java program called TestProcess (which is included in the STAXMon.jar file) and passes it different parameters which effect how long it runs until it completes and whether it is successful or not. The parameters for TestProcess are number of loops, seconds to wait between loops, and RC to return at the end of the process.

This function takes an argument map			
Name	**Description**	**Required**	**Default**
machName	Location (machine name) to run the process	Yes	N/A
blockNum	Number used in conjunction with the machine name to get a unique block name (in case running multiple times on the same machine)	Yes	N/A
loopNum	Current loop number	Yes	N/A

RunSTAFCommands

This function runs several STAF Commands using following STAF services: DELAY, MISC, and SERVICE.

This function takes an argument map			
Name	**Description**	**Required**	**Default**
machName	Location (machine name) to run the process	Yes	N/A
blockNum	Number used in conjunction with the machine name to get a unique block name (in case running multiple times on the same machine)	Yes	N/A

PASS-if-0

This function checks if a value is 0. If 0, it sets the testcase status result to 'pass'; otherwise, it sets it to 'fail' and sends a message to the STAXMonitor.

This function takes a single argument	
Name	value
Description	Value (usually RC or STAXResult variable) to compare with 0
Required	Yes
Default	N/A

Debugging

If you have a question or are experiencing a problem, first check out the STAF/STAX FAQ at http://staf.sourceforge.net/current/STAFFAQ.htm to see if it provides an answer to your question or problem.

Note that since STAX is a Java service, errors that occur running the STAX service may be logged to its JVM log. If you're experiencing a problem configuring the STAX service, or an RC 6 submitting a request to the STAX service, or a problem running a STAX job, such as the job hanging, check its JVM log to see if any additional information about the problem is logged, such as a Java exception. The JVM logs are stored in the {STAF/Config/STAFRoot}/data/JSTAF/<JVMName> directory on the system where the STAX service is registered, where <JVMName> is either STAFJVM1 if you're using the default JVM name or it's the value specified for the JVMName option when the STAX service was registered. The current JVM log is named JVMLog.1.

If the JVM log contains an OutOfMemory error, any Java services using this JVM will have to be removed and added (registered) in order to start accepting requests. You may want to look at increasing the JVM's maximum heap size as the Java service(s) using this JVM may require more memory than can be allocated. Refer to the STAF User's Guide, section "4.4.3 JSTAF service proxy library", for more information on how to do this. If the JVM was killed, any Java services using this JVM will have to be removed and added (registered) in order to start accepting requests.

Support Information

If you have a question or are experiencing a problem, first check out the STAF/STAX FAQ at http://staf.sourceforge.net/current/STAFFAQ.htm to see if it provides an answer to your question or problem.

Please report bugs or request features via the STAF SourceForge website at:

http://staf.sourceforge.net

You may also post questions, problems, comments and suggestions via this website.

It is our goal to retain backward compatibility in future versions of STAX.

Known Problems

Here are some of the known problems and issues with this version of STAX. You may view a complete list of bugs and requested features via the STAF SourceForge website at:

http://staf.sourceforge.net

These problems will be resolved in a future version of STAX.

1. Each time the STAX Service is started, the Job ID is reset to 1. Thus, STAX log files can contain information for multiple jobs. That is, log file STAX_LOG_1 can contain information for multiple jobs whose Job ID was 1. If you do not want this to happen, each time you stop the STAX Service, you can backup the STAX log files (if desired) and enable "Clear Logs" when registering the STAX service or when executing a STAX job.

   Note, multiple jobs in the same log are separated by messages with a level of "start" and "stop" with the message text beginning with "JobID: ". So, even when multiple jobs appear in the same log file, it is still possible to determine where one job ends and another one starts.

2. More detailed tracing of a job for debugging purposes will be provided in a future version of STAX.

3. More information about how to override and then restore the default signal handlers will be provided in a future version of STAX.

History of Changes

See the History file provided in the STAX zip/tar file you downloaded for a history of the changes that have been made to the STAX service.

Appendix A: Comparison of STAX with GenWL

STAX allows you to automate workflow in your test environment as does the Generic WorkLoad Processor (GenWL). The intention is to replace GenWL with STAX. STAX provides all the functions of GenWL and more.

Appendix B: STAX XML Document Examples

Some examples of STAX XML documents are provided in the samples and libraries directories as part of the STAX zip/tar file. The samples directory contains examples of STAX jobs. The libraries directory contain STAX XML documents that contain common functions that you may want to import and call from STAX XML documents that you write.

STAX Utility Functions

A STAX xml file called STAXUtil.xml is provided in the libraries directory. It contains some common STAX utility functions that you may want to import using the <import> element and call from STAX XML documents that you write. This allows you to reuse common functions and makes integration of tests written by different parties easier. See the STAXUtil.html file that is also provided in the libraries directory for more information about these functions, such as the arguments to pass in, what is returned, and usage examples. Note that this HTML file was generated using a XSLT processor which converted information in the STAXUtil.xml file into readable HTML documentation.

STAXUtil.xml contains the following functions:

- **STAF**

  Submits a request to STAF. It's a shortcut for the <stafcmd> element.

- **STAFProcess**

  Submits a STAF request to start a process in a separate shell (using the default shell command). It's a shortcut for the <process> element when you only need to specify the command to be started in a separate shell.

- **STAFProcessUsing**

  Submits a STAF request to start a process using a map to define values for the process element's sub-elements. It's a shortcut for the <process> element when additional options need to be specified.

- **STAXUtilLogAndMsg**

  Logs a message and sends the message to the STAX Monitor. It's a shortcut for specifying the <message> and <log> elements for the same message.

- **STAXUtilWaitForSTAF**

  Waits for STAF to become available (that is, for the STAFProc daemon to be running) on one or more machines. A maximum wait time can be specified, overriding the default maximum wait time of 5 minutes. If one or more machines are not available, and the maximum wait time has not been exceeded, delays 5 seconds and then retries. This function can be useful after rebooting one or more systems.

- **STAXUtilCopyFiles**

  Copies files from a directory on a machine to a directory on the same or different machine. You can specify which files to copy from a directory using the name pattern, extension pattern, case sensitivity, and/or regular expression arguments.

  The regular expression allows you to define complicated pattern matching rules to determine which files to copy.

Note that this function only copies files; no subdirectories will be copied.

For performance reasons, the files are copied in groups of up to 5 in parallel. If the toDirectory does not exist, it will be created. If any of the files being copied already exist on the toMachine, they will be replaced.

- **STAXUtilListDirectory**

  Lists files in a directory on a machine. You can specify which files to list using the name pattern, extension pattern, case sensitivity, and/or regular expression arguments.

  The regular expression allows you to define complicated pattern matching rules to determine which files to copy.

- **STAXUtilCheckSuccess**

  Checks if a result indicates success or failure. If the result evaluates to a true value:

  1. If a pass message is provided, it is logged in the STAX User Log and, optionally, sent to the STAX Monitor.
  2. A testcase status of pass is recorded if the recordStatus evaluates to a true expression.

  Otherwise, if the result evaluates to a false value:

  1. If a failure message is provided, it is logged in the STAX User Log and, optionally, sent to the STAX Monitor.
  2. A testcase status of fail is recorded if the recordStatus evaluates to a true expression.

- **STAXUtilImportSTAFVars**

  Imports STAF variables on the specified machines, creating STAX variables from them.

  If only one machine is specified, for each STAF variable name that is specified, a STAX variable with the specified name is created with the resolved contents of the STAF variable for the specified machine assigned to it.

  If a list of machines is specified, for each STAF variable name that is specified, a STAX map is created with the specified name which contains an entry for each machine (key) with a value of the resolved contents of the STAF variable for that machine assigned.

- **STAXUtilImportSTAFConfigVars**

  Imports STAF Configuration variables (such as STAF/Config/OS/Name and STAF/Config/Sep/File) on the specified machine, creating a STAX variable map containing their values and returning the map.

- **STAXUtilExportSTAFVars**

  Exports STAX variables, creating STAF variables from them on the specified machines.

  For each STAX variable name that is specified, a STAF variable with the specified name is created for the specified machines.

- **STAXUtilQueryAllTests**

  For each STAX variable name that is specified, a STAF variable with the specified name is created for the specified machines. Query the results for all testcases in the currently running job, accumulating the total number of testcases, passes, and fails recorded so far as well as a map of all the testcases and their passes, fails, elapsed times, and number of starts.

- **STAXUtilQueryTest**

  Query the results for a single testcase in the currently running job.

Sample STAX Jobs

Here are a couple of examples of a STAX XML document's contents. The following example is the sample1.xml file provided in the samples directory as part of the STAX zip/tar file.

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

<!--
    sample1.xml - Sample of a job definition file for STAX

    Job Description:

    This job executes various STAF commands and processes and
    sends messages to the STAX Job Monitor.
-->

<stax>

  <!--
    Change the values specified for MachList and/or duration
    and/or STAXJarFile as desired by changing the values passed to
    function MonitorTest via the <defaultcall> element in this file
    or override them by specifying the arguments parameter when
    submitting the job for execution.  Examples of values for the
    arguments parameter:
      { 'duration': '10m', 'MachList': ['machA', machB', 'machC'] }
      { 'MachList': ['myMachine'] }
      { 'duration': '1h' }
      { 'STAXJarFile': '/usr/local/staf/services/STAXMon.jar' }
   -->

  <script>
    STAXServicesDir = '{STAF/Config/STAFRoot}{STAF/Config/Sep/File}services'
    STAXJarFile = '%s{STAF/Config/Sep/File}STAXMon.jar' % STAXServicesDir
  </script>

  <defaultcall function="MonitorTest">
    { 'MachList': ['local', 'local'], 'duration': '2m', 'STAXJarFile': STAXJarFile }
  </defaultcall>

  <function name="MonitorTest" scope="local" requires="RunProcesses">

    <function-prolog>
      For each machine specified by the MachList argument, function
      RunProcesses is called and run in parallel.  This is done in a
      continous loop until the time specified by the duration argument
      is reached.
    </function-prolog>

    <function-map-args>

      <function-optional-arg name="duration" default="'2m'">
        Timer duration to run the test.  e.g. '5m', '1h', '90s', etc.
      </function-optional-arg>

      <function-optional-arg name="MachList" default="['local', 'local']">
        List of machines where the test will be run
      </function-optional-arg>

      <function-optional-arg name="STAXJarFile" default="STAFJarFile">
        Fully-qualified name of STAX jar file on each machine where the test will be
run
      </function-optional-arg>

    </function-map-args>

    <testcase name = "'Timer'">

      <sequence>
        <script>
          import time
          starttime = time.time(); # record starting time
        </script>

        <message>
          'duration=%s, MachList=%s' % (duration, MachList)
        </message>

        <!-- Resolve the STAXJarFile which may contain STAF variables -->
        <stafcmd>
          <location>'local'</location>
          <service>'var'</service>
          <request>'resolve %s' % STAXJarFile</request>
        </stafcmd>

        <if expr="RC == 0">
          <sequence>
            <script>STAXJarFile = STAFResult</script>
            <message>'STAXJarFile=%s' % (STAXJarFile)</message>
          </sequence>
          <else>
            <sequence>
              <message>
                'Error resolving STAXJarFile: RC=%s, STAFResult=%s, \
                 STAXJarFile=%s' % (RC, STAFResult, STAXJarFile)
              </message>
              <message>'Terminating job'</message>
              <terminate block="'main'"/>
            </sequence>
          </else>
        </if>

        <!-- Loop continuously for the specified duration -->
        <timer duration="duration">
          <loop var="loopNum">
            <paralleliterate var="machName" in="MachList" indexvar="i">
              <block name="'%s_%d' % (machName, i)">
                <testcase name="machName">
                  <call-with-map function="'RunProcesses'">
                    <call-map-arg name="'machName'">machName</call-map-arg>
                    <call-map-arg name="'loopNum'">loopNum</call-map-arg>
                    <call-map-arg name="'blockNum'">i</call-map-arg>
                  </call-with-map>
                </testcase>
              </block>
            </paralleliterate>
          </loop>
        </timer>

        <script>
          stoptime = time.time()             # record ending time
          elapsedSecs = stoptime - starttime # difference yields time elapsed in
seconds
        </script>

        <message>'Test complete - ran for %d seconds' % elapsedSecs</message>

        <if expr="RC == 1">
          <tcstatus result="'pass'">
            'Timer ran for %d seconds' % elapsedSecs
          </tcstatus>
          <else>
            <tcstatus result="'fail'">
             'Timer only ran for %d seconds. RC=%d' % (elapsedSecs, RC)
            </tcstatus>
          </else>
        </if>

      </sequence>

    </testcase>

  </function>

  <function name="RunProcesses" scope="local"
                                requires="PASS-if-0 RunSTAFCommands">

    <function-prolog>
      This function runs multiple processes.  Each process runs a Java
      program called TestProcess (which is included in the STAXMon.jar file)
      and passes it different parameters which effect how long it runs
      until it completes and whether it is successful or not.
      The parameters for TestProcess are number of loops, seconds to wait
      between loops, and RC to return at the end of the process.
    </function-prolog>

    <function-map-args>
```

```
      <function-required-arg name="machName">
        Location (machine name) to run the process
      </function-required-arg>

      <function-required-arg name="blockNum">
        Number used in conjunction with the machine name to get a unique
        block name (in case running multiple times on the same machine)
      </function-required-arg>

      <function-required-arg name="loopNum">
        Current loop number
      </function-required-arg>

    </function-map-args>

    <sequence>

      <message>
        'Starting run #%d on %s' % (loopNum, machName)
      </message>

      <script>
        className = 'com.ibm.staf.service.stax.TestProcess'
      </script>

      <process name="'TestProcess'">
        <location>machName</location>
        <command>'java'</command>
        <parms>'%s 5 6 0' % className</parms>
        <title>'First title example'</title>
        <env>'CLASSPATH=%s{STAF/Config/Sep/Path}{STAF/Env/ClassPath}' %
STAXJarFile</env>
        <console use="'same'"/>
      </process>

      <call function="'PASS-if-0'">RC</call>

      <message>
        'Process RC=%d on machine %s' % (RC, machName)
      </message>

      <call function="'RunSTAFCommands'">
        { 'machName': machName, 'blockNum': blockNum }
      </call>

      <call function="'PASS-if-0'">STAXResult</call>

      <process name="'TestProcess'">
        <location>machName</location>
        <command>'java'</command>
        <parms>'%s 3 4 99' % className</parms>
        <env>'CLASSPATH=%s{STAF/Config/Sep/Path}{STAF/Env/ClassPath}' %
STAXJarFile</env>
        <console use="'same'"/>
      </process>
      <call function="'PASS-if-0'">RC</call>

      <message>
        'Process RC=%d on machine %s' % (RC, machName)
      </message>

      <process name="'TestProcess'">
        <location>machName</location>
        <command>'java'</command>
        <parms>'%s 5 5 100' % className</parms>
        <title>'Second title example with many Process elements'</title>
        <workload>'STAX Monitor Workload'</workload>
        <vars>['firstName=Dave','middleInitial=M.','lastName=Bender']</vars>
        <vars>['pet=cat','petName=Fluffy']</vars>
        <env>'CLASSPATH=%s{STAF/Config/Sep/Path}{STAF/Env/ClassPath}' %
STAXJarFile</env>
        <env>'JAVA_APP=javaw.exe'</env>
        <useprocessvars/>
        <disabledauth action="'ignore'"/>
        <console use="'same'"/>
      </process>

      <call function="'PASS-if-0'">RC</call>

      <message>
        'Process RC=%d on machine %s' % (RC, machName)
      </message>

      <message>
        'Finished run #%d on machine %s' % (loopNum, machName)
      </message>

    </sequence>

  </function>


  <function name="RunSTAFCommands" scope="local">

    <function-prolog>
      This function runs several STAF Commands using following
      STAF services: DELAY, MISC, and SERVICE.
    </function-prolog>

    <function-map-args>

      <function-required-arg name="machName">
        Location (machine name) to run the process
      </function-required-arg>

      <function-required-arg name="blockNum">
        Number used in conjunction with the machine name to get a unique
        block name (in case running multiple times on the same machine)
      </function-required-arg>

    </function-map-args>
    <block name="'STAFCommandBlock%d' % blockNum">

      <sequence>

        <script>from random import random;r=random();r=r*10000</script>

        <message>'Delaying %d ms on machine %s' % (r,machName)</message>

        <stafcmd name="'STAF Command: RANDOM DELAY'">
          <location>machName</location>
          <service>'delay'</service>
          <request>'delay %d' % r</request>
        </stafcmd>

        <if expr="RC != 0">
          <return>RC</return>
        </if>

        <stafcmd name="'STAF Command: MISC VERSION'">
          <location>machName</location>
          <service>'misc'</service>
          <request>'version'</request>
        </stafcmd>

        <if expr="RC != 0">
          <return>RC</return>
          <else>
            <message>
              'Machine %s has STAF Version %s' % (machName,STAFResult)
            </message>
          </else>
        </if>

        <stafcmd name="'STAF Command: SERVICE LIST'">
          <location>machName</location>
          <service>'service'</service>
          <request>'list'</request>
        </stafcmd>

        <if expr="RC != 0">
          <return>RC</return>
          <else>
            <message>
              'Machine %s has STAF services:\n%s' % (machName,STAFResult)
            </message>
          </else>
        </if>

        <return>0</return>

      </sequence>

    </block>

  </function>
```

```
  <function name="PASS-if-0" scope="local">

    <function-prolog>
      This function checks if a value is 0.  If 0, it sets the
      testcase status result to 'pass'; otherwise, it sets it
      to 'fail' and sends a message to the STAXMonitor.
    </function-prolog>

    <function-single-arg>
      <function-required-arg name="value">
        Value (usually RC or STAXResult variable) to compare with 0
      </function-required-arg>
    </function-single-arg>

    <if expr="value == 0">
      <tcstatus result="'pass'"/>
      <else>
        <tcstatus result="'fail'">
          'value=%d. Expected 0.' % value
        </tcstatus>
      </else>
    </if>
  </function>

</stax>
```

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<!DOCTYPE stax SYSTEM "stax.dtd">
<!--
   j13auto.xml

   XML Sample for STAX (STAf eXecution)

   Job Description:

    This job executes a set of automated tests on each machine
    defined in the "TestMachines" List. Each machine executes
    in parallel.  The setup and tests run serially on each machine.
    On each machine, the "Setup" Function is executed, and then the
    "RunVariations" Function is executed first for the "Java2D API
    variations" list and then for the "Print API variations" List.

    This job imports functions from STAXUtil.xml.
-->

<stax>

  <defaultcall function="j13auto"/>

  <script>
    # Make sure the importDir is where you put STAXUtil.xml
    importDir = 'C:/staf/services/libraries'
    TestCaseServer = 'AL1B4'
    TestCaseDir = 'D:/jdk13/tests/api'
    TestCaseFiles = ['Java2DAPI.jar','PrintAPI.jar']
    TestMachines = ['AL2C4','AM3D4','AA3B4','CE1A4','AH2D4']
    Java2DAPI = ['AlphaComposite001',     'AlphaComposite002',
                 'AlphaComposite003',     'GraphicsEnvironment001',
                 'GraphicsEnvironment002','ICC_ProfileRGB001',
                 'ICC_ProfileRGB002',     'ICC_ProfileRGB003',
                 'ICC_ProfileRGB004',     'ICC_ProfileRGB005',
                 'ICC_ProfileRGB006',     'GlyphMetrics001',
                 'GlyphMetrics002',       'DirectColorModel001',
                 'DirectColorModel002',   'DirectColorModel003',
                 'DirectColorModel004',   'DirectColorModel005']
    PrintAPI =  ['PageFormat001',     'PageFormat002',
                 'PrinterJob001',     'PrinterJob002',
                 'PrinterJob003',     'PrinterJob004',
                 'PageAttributes001', 'PageAttributes002',
                 'PageAttributes003']
  </script>

  <function name="j13auto">

    <sequence>

      <import machine="'local'" file="'%s/STAXUtil.xml' % importDir"/>

      <paralleliterate var="machName" in="TestMachines">

        <sequence>

          <call function="'Setup'"/>

          <testcase name="Java2D">
            <sequence>
              <script>variationList = Java2DAPI[:]</script>
              <script>jarName = '%s/%s' % (TestCaseDir, TestCaseFiles[0])</script>
              <call function="'RunVariations'"/>
            </sequence>
          </testcase>

          <testcase name="Print">
            <sequence>
              <script>variationList = PrintAPI[:]</script>
              <script>jarName = '%s/%s' % (TestCaseDir, TestCaseFiles[1])</script>
              <call function="'RunVariations'"/>
            </sequence>
          </testcase>

        </sequence>

      </paralleliterate>

    </sequence>

  </function>
```

```
  <function name="Setup">

    <iterate var="file" in="TestCaseFiles">

      <stafcmd>
        <location>TestCaseServer</location>
        <service>'FS'</service>
        <request>
          'COPY FILE %s/%s TOMACHINE %s' % (TestCaseDir, file, machName)
        </request>
      </stafcmd>

    </iterate>

  </function>

  <function name="RunVariations">

    <testcase name="machName">

      <iterate var="variationName" in="variationList">

        <sequence>

          <process>
            <location>machName</location>
            <command>'java'</command>
            <parms>'-jar ' + jarName + ' ' + variationName</parms>
          </process>

          <call> function="'STAXUtilCheckSuccess'">
            { 'result': RC == 0,
               'failMsg': 'Process failed. RC=%s Result=%s' % (RC, STAFResult),
               'sendToMonitor': 1, 'recordStatus': 1 }
          </call>

        </sequence>

      </iterate>

    </testcase>

  </function>

</stax>
```

If you performed a STAX EXECUTE request specifying a file containing the above sample XML file and then ran the following STAX request to query the job's testcase information right before the job completed:

```
  LIST JOB 15 TESTCASES
```

you would see output similar to the following (depending on how many testcase variations passed or failed on each machine) if "Log TC Elapsed Time" and "Log TC Num Starts" are disabled:

```
Java2D.AL2C4;18;0
Java2D.AM3D4;16;2
Java2D.AA3B4;5;13
Java2D.CE1C4;18;0
Java2D.AH2D4;18;0
Print.AL2C4;9;0
Print.AM3D4;9;0
Print.AA3B4;7;2
Print.CE1C4;8;1
Print.AH2D4;9;0
```

Appendix C: STAX Service Error Code Reference

In addition to the common STAF return codes, the following STAX service return codes are defined:

Error Code	**Meaning**	**Comment**
4001	Error submitting execute request	Additional information about the error is put into the STAF Result. An example of additional information that may be provided is: Caught com.ibm.staf.service.stax.STAXXMLParseException: Line 78: The element type "sequence" must be terminated by the matching end-tag "</sequence>". In this case, it indicates an error in your XML file that you must correct.
4002	Block not held	Requested to release a block that is not held.
4003	Block already held	Requested to hold a block that is already held.

Appendix D: STAX Document Type Definition (DTD)

This section contains the DTD for the STAX service (without any extensions). You can display the DTD that the STAX service is using (including any extensions) by issuing the GET DTD request.

```
<!--
   STAf eXecution (STAX) Document Type Definition (DTD)

   Generated Date: 20031021-17:54:56

   This DTD module is identified by the SYSTEM identifier:

     SYSTEM 'stax.dtd'

-->

<!-- Parameter entities referenced in Element declarations -->
```

```
<!ENTITY % stax-elems 'function | script | signalhandler'>

<!ENTITY % task       'timer | log | parallel |
                       call | stafcmd | script |
                       tcstatus | message | iterate |
                       sequence | import | raise |
                       job | nop | process |
                       try | break | testcase |
                       paralleliterate | continue | throw |
                       release | signalhandler | rethrow |
                       block | hold | terminate |
                       return | if | call-with-list |
                       loop | call-with-map'>

<!--================= STAX Job Definition ========================== -->
<!--
     The root element STAX contains all other elements.  It consists
     of an optional defaultcall element and any number of function,
     script, and/or signalhandler elements.
-->
<!ELEMENT stax        ((%stax-elems;)*, defaultcall?, (%stax-elems;)*)>

<!--================= The Default Call Function Element ============ -->
<!--
     The defaultcall element defines the function to call by default
     to start the job.  This can be overridden by the 'FUNCTION'
     parameter when submitting the job to be executed.
     The function attribute's value is a literal.
-->
<!ELEMENT defaultcall  (#PCDATA)>
<!ATTLIST defaultcall
          function     IDREF    #REQUIRED
>

<!--================= Continue Element ============================= -->
<!--
     The continue element can be used to continue to the top of a loop
     or iterate element.
-->
<!ELEMENT continue   EMPTY>

<!--================= Break Element ================================ -->
<!--
     The break element can be used to break out of a loop or iterate
     element.
-->
<!ELEMENT break      EMPTY>

<!--================= The Call-With-Map Element ==================== -->
<!--
     Perform a function with the referenced name with any number of
     arguments in the form of a map of named arguments.  The function
     and name attribute values as well as the argument value are
     evaluated via Python.
```

```
-->
<!ELEMENT call-with-map       (call-map-arg*)>
<!ATTLIST call-with-map
          function   CDATA    #REQUIRED
>

<!ELEMENT call-map-arg        (#PCDATA)>
<!ATTLIST call-map-arg
          name       CDATA    #REQUIRED
>

<!--================= The No Operation Element ===================== -->
<!--
     No operation action.
-->
<!ELEMENT nop        EMPTY>

<!--================= The Function Element ========================= -->
<!--
     The function element defines a named task which can be called.
     The name and scope attribute values are literals.
     If desired, the function can be described using a function-prolog
     element (or the deprecated function-description element) and/or a
     function-epilog element.  Also, if desired, the function element
     can define the arguments that can be passed to the function.
-->
<!ELEMENT function     ((function-prolog | function-description)?,
                        (function-epilog)?,
                        (function-no-args | function-single-arg |
                         function-list-args | function-map-args)?,
                        (%task;))>
<!ATTLIST function
          name         ID       #REQUIRED
          requires     IDREFS   #IMPLIED
          scope        (local | global) "global"
>

<!ELEMENT function-prolog       (#PCDATA)>

<!ELEMENT function-epilog       (#PCDATA)>

<!ELEMENT function-description  (#PCDATA)>

<!ELEMENT function-no-args      EMPTY>

<!ELEMENT function-single-arg   (function-required-arg |
                                 function-optional-arg)>

<!ELEMENT function-list-args    (((function-required-arg+,
                                   function-optional-arg*) |
                                  (function-required-arg*,
                                   function-optional-arg+)),
                                 (function-other-args)?)>

<!ELEMENT function-map-args     ((function-required-arg |
                                   function-optional-arg)+,
                                  (function-other-args)?)>

<!ELEMENT function-required-arg (#PCDATA)>
<!ATTLIST function-required-arg
          name         CDATA    #REQUIRED
>

<!ELEMENT function-optional-arg (#PCDATA)>
<!ATTLIST function-optional-arg
          name         CDATA    #REQUIRED
          default      CDATA    "None"
>

<!ELEMENT function-other-args   (#PCDATA)>
<!ATTLIST function-other-args
          name         CDATA    #REQUIRED
>

<!--================= The Iterate Element ========================= -->
<!--
     The iterate element iterates through a list of items, performing
     its contained task while substituting each item in the list.
     The iterated tasks are performed in sequence.
-->
<!ELEMENT iterate  (%task;)>
<!-- var      is the name of the variable which will contain the
              current item in the list or tuple being iterated.
              It is a literal.
     in       is the list or tuple to be iterated.  It is evaluated
              via Python and must evaluate to be a list or tuple.
     indexvar is the name of a variable which will contain the index of
              the current item in the list or tuple being iterated.
              It is a literal.  The value for the first index is 0.
-->
<!ATTLIST iterate
          var        CDATA    #REQUIRED
          in         CDATA    #REQUIRED
          indexvar   CDATA    #IMPLIED
>

<!--================= The Log Element ============================== -->
<!--
     Writes a message and its log level to a STAX Job User Log file.
     The message must evaluate via Python to a string.

     The log level specified defaults to 'info'.  If specified, it
     must evaluate via Python to a string containing be one of the
     following STAF Log Service Log levels:
       fatal, warning, info, trace, trace2, trace3, debug, debug2,
       debug3, start, stop, pass, fail, status, user1, user2, user3,
       user4, user5, user6, user7, user8
     If an if attribute is specified and it evaluates via Python to
     false, then the log element is ignored.
-->
```

```
<!ELEMENT log         (#PCDATA)>
<!ATTLIST log
          level       CDATA       "'info'"
          if          CDATA       "1"
>

<!--================= The Parallel Iterate Element ================ -->
<!--
     The parallel iterate element iterates through a list of items,
     performing its contained task while substituting each item in
     the list.  The iterated tasks are performed in parallel.
-->
<!ELEMENT paralleliterate  (%task;)>
<!-- var      is the name of a variable which will contain the current
              item in the list or tuple being iterated.
              It is a literal.
     in       is the list or tuple to be iterated.  It is evaluated
              via Python and must evaluate to be a list or tuple.
     indexvar is the name of a variable which will contain the index of
              the current item in the list or tuple being iterated.
              It is a literal.  The value of the first index is 0.
-->
<!ATTLIST paralleliterate
          var        CDATA    #REQUIRED
          in         CDATA    #REQUIRED
          indexvar   CDATA    #IMPLIED
>

<!--================= The Throw Element ============================ -->
<!--
     The throw element specifies an exception to throw.
     The exception attribute value and any additional information
     is evaluated via Python.
-->
<!ELEMENT throw      (#PCDATA)>
<!ATTLIST throw
          exception  CDATA        #REQUIRED
>

<!--================= The Block Element ============================ -->
<!--
     Defines a task block that can be held, released, or terminated.
     Used in conjunction with the hold/terminate/release elements to
     define a task block that can be held, terminated, or released.
     The name attribute value is evaluated via Python.
-->
<!ELEMENT block      (%task;)>
<!ATTLIST block
          name       CDATA    #REQUIRED
>

<!--================= The Try / Catch Block Elements =============== -->
<!--
     The try element allows you to perform an action and to catch
     exceptions that are thrown.
-->
<!ELEMENT try        ((%task;), catch+)>
<!--
     The catch element performs a task when the specified exception is
     caught.  The var attribute specifies the name of the variable to
     receive the data specified within the throw element.  The typevar
     attribute specifies the name of the variable to receive the type
     of the exception.

-->
<!ELEMENT catch      (%task;)>
<!ATTLIST catch
          exception  CDATA        #REQUIRED
          var        CDATA        #IMPLIED
          typevar    CDATA        #IMPLIED
>

<!--================= The Terminate Element ========================= -->
<!--
     The terminate element specifies to terminate a block in the job.
-->
<!ELEMENT terminate  EMPTY>
<!ATTLIST terminate
          block      CDATA    #IMPLIED
>

<!--================= The Sequence Element ========================== -->
<!--
     The sequence element performs one or more tasks in sequence.
-->
<!ELEMENT sequence   (%task;)+>

<!--================= The Timer Element ============================= -->
<!--
     The timer element runs a task for a specified duration.
     If the task is still running at the end of the specified duration,
     then the RC variable is set to 1, else if the task ended before
     the specified duration, the RC variable is set to 0, else if the
     timer could not start due to an invalid duration, the RC variable
     is set to -1.
-->
<!ELEMENT timer     (%task;)>
<!-- duration is the maximum length of time to run the task.
       Time can be expressed in milliseconds, seconds, minutes,
       hours, days, weeks, or years.  It is evaluated via Python.
         Examples:  duration='50'    (50 milliseconds)
                    duration='90s'   (90 seconds)
                    duration='5m'    ( 5 minutes)
                    duration='36h'   (36 hours)
                    duration='3d'    ( 3 days)
                    duration='1w'    ( 1 week)
                    duration='1y'    ( 1 year)
-->
```

```
<!ATTLIST timer
          duration   CDATA        #REQUIRED
>

<!--================= The Message Element ========================== -->
<!--
     Generates an event and makes the message value available to the
     STAX Job Monitor.  The message must evaluate via Python to a string.
     If an if attribute is specified and it evaluates via Python to
     false, the message element is ignored.
-->
<!ELEMENT message     (#PCDATA)>
<!ATTLIST message
          if          CDATA       "1"
>

<!--================= The Signal Handler Element =================== -->
<!--
     The signalhandler element defines how to handle a specified signal.
     The signal attribute value is evaluated via Python.
-->
<!ELEMENT signalhandler (%task;)>
<!ATTLIST signalhandler
          signal     CDATA        #REQUIRED
>

<!--================= The Testcase Element ========================= -->
<!--
     Defines a testcase.  Used in conjunction with the tcstatus
     element to mark the status for a testcase.
     The name attribute value is evaluated via Python.
-->
<!ELEMENT testcase   (%task;)>
<!ATTLIST testcase
          name       CDATA    #REQUIRED
          mode       CDATA    "'default'"
>

<!--================= The Import Elements ============================ -->
<!--
     Allows importing of functions from another STAX XML job file.
-->
<!ELEMENT import       (import-include?, import-exclude?)?>
<!ATTLIST import
          machine   CDATA    #REQUIRED
          file      CDATA    #REQUIRED
          mode      CDATA    "'error'"
>

<!ELEMENT import-include          (#PCDATA)>
<!ELEMENT import-exclude          (#PCDATA)>

<!--================= The Hold Element ============================= -->
<!--
     The hold element specifies to hold a block in the job.
```

```
-->
<!ELEMENT hold       EMPTY>
<!ATTLIST hold
          block      CDATA    #IMPLIED
>

<!--================= The Conditional Element (if-then-else) ======= -->
<!--
     Allows you to write an if or a case construct with zero or more
     elseifs and one or no else statements.

     The expr attribute value is evaluated via Python and must evaluate
     to a boolean value.
-->
<!ELEMENT if         ((%task;), elseif*, else?)>
<!ATTLIST if
          expr       CDATA   #REQUIRED
>
<!ELEMENT elseif     (%task;)>
<!ATTLIST elseif
          expr       CDATA   #REQUIRED
>
<!ELEMENT else       (%task;)>

<!--================= The STAF Command Element ===================== -->
<!--
     Specifies a STAF command to be executed.
     Its name and all of its element values are evaluated via Python.
-->
<!ELEMENT stafcmd    (location, service, request)>
<!ATTLIST stafcmd
          name       CDATA   #IMPLIED
>
<!ELEMENT service    (#PCDATA)>
<!ELEMENT request    (#PCDATA)>

<!--================= The STAF Process Element ===================== -->
<!--
     Specifies a STAF process to be started.
     All of its non-empty element values are evaluated via Python.
-->
<!ENTITY % procgroup1 '((parms?, workdir?) | (workdir?, parms?))'>
<!ENTITY % procgroup2 '((title?, workload?) | (workload?, title?))'>
<!ENTITY % procgroup1a '((parms?, workload?) | (workload?, parms?))'>
<!ENTITY % procgroup2a '((title?, workdir?) | (workdir?, title?))'>
<!ENTITY % procgroup3 '(((vars | var | envs | env)*, useprocessvars?) |
                        (useprocessvars?, (vars | var | envs | env)*))'>
<!ENTITY % procgroup4 '(((username, password?)?, disabledauth?) |
                        ((disabledauth?, (username, password?)?)))'>
<!ENTITY % procgroup5 '((stdin?, stdout?, stderr?) |
                        (stdout?, stderr?, stdin?) |
                        (stderr?, stdin?, stdout?) |
                        (stdin?, stderr?, stdout?) |
                        (stdout?, stdin?, stderr?) |
                        (stderr?, stdout?, stdin?))'>
<!ENTITY % returnfileinfo '(returnfiles | returnfile)*'>
<!ENTITY % procgroup5a '((%returnfileinfo;, returnstdout?, returnstderr?) |
                        (returnstdout?, returnstderr?, %returnfileinfo;) |
                        (returnstderr?, %returnfileinfo;, returnstdout?) |
                        (%returnfileinfo;, returnstderr?, returnstdout?) |
                        (returnstdout?, %returnfileinfo;, returnstderr?) |
                        (returnstderr?, returnstdout?, %returnfileinfo;))'>
<!ENTITY % procgroup6 '((stopusing?, console?, statichandlename?) |
                        (stopusing?, statichandlename?, console?) |
                        (console?, stopusing?, statichandlename?) |
                        (console?, statichandlename?, stopusing?) |
                        (statichandlename?, stopusing?, console?) |
                        (statichandlename?, console?, stopusing?))'>
<!ELEMENT process    (location, command,
                      ((%procgroup1;, %procgroup2;) |
                       (%procgroup2;, %procgroup1;) |
                       (%procgroup1a;, %procgroup2a;) |
                       (%procgroup2a;, %procgroup1a;)),
                      %procgroup3;,
                      ((%procgroup4;, %procgroup5;, %procgroup5a;, %procgroup6;) |
                       (%procgroup4;, %procgroup6;, %procgroup5;, %procgroup5a;) |
                       (%procgroup5;, %procgroup5a;, %procgroup4;, %procgroup6;) |
                       (%procgroup5;, %procgroup5a;, %procgroup6;, %procgroup4;) |
                       (%procgroup6;, %procgroup4;, %procgroup5;, %procgroup5a;) |
                       (%procgroup6;, %procgroup5;, %procgroup5a;, %procgroup4;)),
                      other?, process-action?)>
<!ATTLIST process
          name        CDATA   #IMPLIED
>

<!--
     The process element must contain a location element and a
     command element.
-->
<!ELEMENT location            (#PCDATA)>
<!ELEMENT command             (#PCDATA)>
<!ATTLIST command
          mode      CDATA     "'default'"
          shell     CDATA     #IMPLIED
>

<!--
     The parms element specifies any parameters that you wish to
     pass to the command.
     The value is evaluated via Python to a string.
-->
<!ELEMENT parms               (#PCDATA)>
<!ATTLIST parms
          if        CDATA     "1"
>

<!--
     The workdir element specifies the directory from which the
     command should be executed.  If you do not specify this
     element, the command will be started from whatever directory
     STAFProc is currently in.
     The value is evaluated via Python to a string.
-->
<!ELEMENT workdir             (#PCDATA)>
<!ATTLIST workdir
          if        CDATA     "1"
>

<!--
     The title element specifies the program title of the process.
     Unless overridden by the process, the title will be the text
     that is displayed on the title bar of the application.
     The value is evaluated via Python to a string.
-->
<!ELEMENT title               (#PCDATA)>
<!ATTLIST title
          if        CDATA     "1"
>

<!--
     The workload element specifies the name of the workload for
     which this process is a member.  This may be useful in
     conjunction with other process elements.
     The value is evaluated via Python to a string.
-->
<!ELEMENT workload            (#PCDATA)>
<!ATTLIST workload
          if        CDATA     "1"
>

<!--
     The vars (and var) elements specify STAF variables that go into the
     process specific STAF variable pool.
     The value must evaluate via Python to a string or a list of
     strings. Multiple vars elements may be specified for a process.
     The format for each variable is:
       'varname=value'
     So, a list containing 3 variables could look like:
       ['var1=value1', 'var2=value2', 'var3=value3']
     Specifying only one variable could look like either:
       ['var1=value1']      or
       'var1=value1'
-->
<!ELEMENT vars                (#PCDATA)>
<!ATTLIST vars
          if        CDATA     "1"
>

<!ELEMENT var                 (#PCDATA)>
<!ATTLIST var
          if        CDATA     "1"
>
```

```
<!--
     The envs (and env) elements specify environment variables that will
     be set for the process.  Environment variables may be mixed case,
     however most programs assume environment variable names will
     be uppercase, so, in most cases, ensure that your environment
     variable names are uppercase.
     The value must evaluate via Python to a string or a list of
     strings. Multiple envs elements may be specified for a process.
     The format for each variable is:
       'varname=value'
     So, a list containing 3 variables could look like:
       ['ENV_VAR_1=value1', 'ENV_VAR_2=value2', 'ENV_VAR_3=value3']
     Specifying only one variable could look like either:
       ['ENV_VAR_1=value1']      or
       'ENV_VAR_1=value1'
-->
<!ELEMENT envs                (#PCDATA)>
<!ATTLIST envs
          if        CDATA     "1"
>

<!ELEMENT env                 (#PCDATA)>
<!ATTLIST env
          if        CDATA     "1"
>
<!--
     The useprocessvars element specifies that STAF variable
     references should try to be resolved from the STAF variable
     pool associated with the process being started first.
     If the STAF variable is not found in this pool, the STAF
     global variable pool should then be searched.
-->
<!ELEMENT useprocessvars      EMPTY>
<!ATTLIST useprocessvars
          if        CDATA     "1"
>

<!--
     The username element specifies the username under which
     the process should be started.
     The value is evaluated via Python to a string.
-->
<!ELEMENT username            (#PCDATA)>
<!ATTLIST username
          if        CDATA     "1"
>

<!--
     The password element specifies the password with which to
     authenticate the user specified with the username element.
     The value is evaluated via Python to a string.
-->
<!ELEMENT password            (#PCDATA)>
```

```
<!ATTLIST password
          if        CDATA     "1"
>

<!-- The disabledauth element specifies the action to take if a
     username/password is specified but authentication has been disabled.

     action  Must evaluate via Python to a string containing either:
             - 'error' specifies that an error should be returned.
             - 'ignore'  specifies that any username/password specified
               is ignored if authentication is desabled.
             This action overrides any default specified in the STAF
             configuration file.
-->
<!ELEMENT disabledauth        EMPTY>
<!ATTLIST disabledauth
          if        CDATA     "1"
          action    CDATA     #REQUIRED
>

<!--
     The stdin element specifies the name of the file from which
     standard input will be read.  The value is evaluated via
     Python to a string.
-->
<!ELEMENT stdin               (#PCDATA)>
<!ATTLIST stdin
          if        CDATA     "1"
>

<!--
     The stdout element specifies the name of the file to which
     standard output will be redirected.
     The value is evaluated via Python to a string.
-->
<!ELEMENT stdout              (#PCDATA)>
<!--  mode  specifies what to do if the file already exists.
            The value must evaluate via Python to one of the
            following:
            'replace' - specifies that the file will be replaced.
            'append'  - specifies that the process' standard
                        output will be appended to the file.
-->
<!ATTLIST stdout
          if        CDATA     "1"
          mode      CDATA     "'replace'"
>

<!--
     The stderr element specifies the file to which standard error will
     be redirected. The mode and filename are evaluated via Python to a
     string.
-->
<!ELEMENT stderr              (#PCDATA)>
<!-- mode   specifies what to do if the file already exists or to
```

```
            redirect standard error to the same file as standard output.
            The value must evaluate via Python to one of the following:
            'replace' - specifies that the file will be replaced.
            'append'  - specifies that the process's standard error will
                        be appended to the file.
            'stdout'  - specifies to redirect standard error to the
                        same file to which standard output is redirected.
                        If a file name is specified, it is ignored.
-->
<!ATTLIST stderr
          if        CDATA     "1"
          mode      CDATA     "'replace'"
>

<!--
     The returnstdout element specifies to return in STAXResult
     the contents of the file where standard output was redirected
     when the process completes.
-->
<!ELEMENT returnstdout        EMPTY>
<!ATTLIST returnstdout
          if        CDATA     "1"
>

<!--
     The returnstderr element specifies to return in STAXResult
     the contents of the file where standard error was redirected
     when the process completes.
-->
<!ELEMENT returnstderr        EMPTY>
<!ATTLIST returnstderr
          if        CDATA     "1"
>

<!--
     The returnfiles (and returnfile) elements specify that the
     contents of the specified file(s) should be returned in
     STAXResult when the process completes.  The value must evaluate
     via Python to a string or a list of strings. Multiple returnfile(s)
     elements may be specified for a process.
-->
<!ELEMENT returnfiles         (#PCDATA)>
<!ATTLIST returnfiles
          if        CDATA     "1"
>

<!ELEMENT returnfile          (#PCDATA)>
<!ATTLIST returnfile
          if        CDATA     "1"
>

<!--
     The stopusing element allows you to specify the method by
     which this process will be STOPed, if not overridden on the
     STOP command.
     The value is evaluated via Python to a string.
-->
<!ELEMENT stopusing          (#PCDATA)>
<!ATTLIST stopusing
          if        CDATA     "1"
>

<!--
     The console element allows you to specify if the process should
     get a new console window or share the STAFProc console.

     use    Must evaluate via Python to a string containing either:
            - 'new' specifies that the process should get a new console
              window.  This option only has effect on Win32.  This is
              the default for Win32 and OS/2 systems.
            - 'same' specifies that the process should share the
              STAFProc console.  This option only has effect on Win32.
              This is the default for Unix systems.
-->
<!ELEMENT console            EMPTY>
<!ATTLIST console
          if        CDATA     "1"
          use       CDATA     #REQUIRED
>

<!--
     Specifies that a static handle should be created for this process.
     The value is evaluated via Python to a string.  It will be the
     registered name of the static handle.  Using this option will also
     cause the environment variable STAF_STATIC_HANDLE to be set
     appropriately for the process.
-->
<!ELEMENT statichandlename   (#PCDATA)>
<!ATTLIST statichandlename
          if        CDATA     "1"
>

<!--
     The process-action element specifies a task to be executed
     when a process has started.
-->
<!ELEMENT process-action     (%task;)>
<!ATTLIST process-action
          if        CDATA     "1"
>

<!--
     The other element specifies any other STAF parameters that
     may arise in the future.  It is used to pass additional data
     to the STAF PROCESS START request.
     The value is evaluated via Python to a string.
-->
<!ELEMENT other              (#PCDATA)>
<!ATTLIST other
          if        CDATA     "1"
```

```
>

<!--================== The STAX Job Element ===================== -->
<!--
     Specifies a STAX sub-job to be executed.  This element is equivalent
     to a STAX EXECUTE request.

     The name attribute specifies the name of the job. The job name
     defaults to the value of the function name called to start the job.
     Its name and all of its element values are evaluated via Python.
     The job element must contain a location element and either a
     file or data element.  This attribute is equivalent to the
     JOBNAME option for a STAX EXECUTE command.

     The clearlogs attribute specifies to delete the STAX Job and Job
     User logs before the job is executed to ensure that only one job's
     contents are in the log.  This attribute is equivalent to the
     CLEARLOGS option for a STAX EXECUTE command.  The default is the
     same option that was specified for the parent job.  Valid values
     include 'parent', 'default', 'enabled', and 'disabled'.

     The monitor attribute specifies whether to automatically monitor the
     subjob.  Note that 'Automatically monitor recommended sub-jobs' must
     be selected in the STAX Job Monitor properties in order for it to be
     used.  The default value for the monitor attribute is 0, a false
     value.

     The logtcelapsedtime attribute specifies to log the elapsed time
     for a testcase in the summary record in the STAX Job log and on a
     LIST TESTCASES request.  This attribute is equivalent to the
     LOGTCELAPSEDTIME option for a STAX EXECUTE command.  The default is
     the same option that was specified for the parent job.  Valid values
     include 'parent', 'default', 'enabled', and 'disabled'.

     The logtcnumstarts attribute specifies to log the number of starts
     for a testcase in the summary record in the STAX Job log and on a
     LIST TESTCASES request.  This attribute is equivalent to the
     LOGNUMSTARTS option for a STAX EXECUTE command.  The default is
     the same option that was specified for the parent job.  Valid values
     include 'parent', 'default', 'enabled', and 'disabled'.

     The logtcstartstop attribute specifies to log start/stop records
     for testcases in the STAX Job log.  This attribute is equivalent to
     the LOGTCSTARTSTOP option for a STAX EXECUTE command.  The default
     is the same option that was specified for the parent job.  Valid
     values include 'parent', 'default', 'enabled', and 'disabled'.

     The job element must contain either a job-file or job-data element.

     The job element has the following optional elements:
       job-function, job-function-args, job-scriptfile(s), and job-script

     Each of these optional elements may specify an if attribute.
     The if attribute must evaluate via Python to a true or false value.
     If it does not evaluate to a true value, the element is ignored.
     The default value for the if attribute is 1, a true value.
     Note that in Python, true means any nonzero number or nonempty
     object; false means not true, such as a zero number, an empty
     object, or None. Comparisons and equality tests return 1 or 0
     (true or false).
-->
<!ELEMENT job        ((job-file | job-data),
                      job-function?, job-function-args?,
                      (job-scriptfile | job-scriptfiles)?,
                      job-script*, job-action?)>
<!ATTLIST job
          name              CDATA   #IMPLIED
          clearlogs         CDATA   "'parent'"
          monitor           CDATA   #IMPLIED
          logtcelapsedtime  CDATA   "'parent'"
          logtcnumstarts    CDATA   "'parent'"
          logtcstartstop    CDATA   "'parent'"
>

<!--
     The job-file element specifies the fully qualified name of a file
     containing the XML document for the STAX job to be executed.
     The job-file element is equivalent to the FILE option for a STAX
     EXECUTE command.

     The machine attribute specifies the name of the machine where the
     xml file is located.  If not specified, it defaults to Python
     variable STAXJobXMLMachine.  The machine attribute is equivalent
     to the MACHINE option for a STAX EXECUTE command.
  -->
<!ELEMENT job-file           (#PCDATA)>
<!ATTLIST job-file
          machine    CDATA   "STAXJobXMLMachine"
>

<!--
     The job-data element specifies a string containing the XML document
     for the job to be executed.  This element is equivalent to the
     DATA option for a STAX EXECUTE command.

     The eval attribute specifies whether the data is be evaluated by
     Python in the parent job.  For example, if the job-data information
     is dynamically generated and assigned to a Python variable, rather
     than just containing the literal XML information, then you would
     need to set the eval attribute to true (e.g. eval="1").
     The default for the eval attribute is false ("0").
  -->
<!ELEMENT job-data           (#PCDATA)>
<!ATTLIST job-data
          eval       CDATA   "0"
>

<!--
     The job-function element specifies the name of the function element
     to call to start the job, overriding the defaultcall element, if any,
     specified in the XML document. The <function name> must be the name of
     a function element specified in the XML document. This element is
     equivalent to the FUNCTION option for a STAX EXECUTE command.
-->
<!ELEMENT job-function       (#PCDATA)>
<!ATTLIST job-function
          if         CDATA   "1"
>

<!--
     The job-function-args element specifies arguments to pass to the
     function element called to start the job, overriding the arguments,
     if any, specified for the defaultcall element in the XML document.
     This element is equivalent to the ARGS option for a STAX EXECUTE
     command.

     The eval attribute specifies whether the data is to be evaluated
     by Python in the parent job.  The default for the eval attribute
     is false ("0").
-->
<!ELEMENT job-function-args  (#PCDATA)>
<!ATTLIST job-function-args
          if         CDATA   "1"
          eval       CDATA   "0"
>

<!--
     The job-script element specifies Python code to be executed.
     This element is equivalent to the SCRIPT option for a STAX
     EXECUTE command.  Multiple job-script elements may be specified.

     The eval attribute specifies whether the data is to be evaluated
     by Python in the parent job.  The default for the eval attribute
     is false ("0").
-->
<!ELEMENT job-script         (#PCDATA)>
<!ATTLIST job-script
          if         CDATA   "1"
          eval       CDATA   "0"
>

<!--
     The job-scriptfile element (equivalent to the job-scriptfiles
     element) specifies the fully qualified name of a file containing
     Python code to be executed, or a list of file names containing
     Python code to be executed. The value must evaluate via Python to
     a string or a list of strings. This element is equivalent to the
     SCRIPTFILE option for a STAX EXECUTE command.

     Specifying only one scriptfile could look like either:
       ['C:/stax/scriptfiles/scriptfile1.py']      or
       'C:/stax/scriptfiles/scriptfiel1.py'
     Specifying a list containing 3 scriptfiles could look like:
       ['C:/stax/scriptfiles/scriptfile1.py',
        'C:/stax/scriptfiles/scriptfile2.py',
        C:/stax/scriptfiles/scriptfile2.py' ]

     The machine attribute specifies the name of the machine where the
     SCRIPTFILE(s) are located. If not specified, it defaults to Python
     variable STAXJobScriptFileMachine.  This attribute is equivalent
     to the SCRIPTFILEMACHINE option for a STAX EXECUTE command.
-->
<!ELEMENT job-scriptfile     (#PCDATA)>
<!ATTLIST job-scriptfile
          if         CDATA   "1"
          machine    CDATA   "STAXJobScriptFileMachine"
>

<!ELEMENT job-scriptfiles    (#PCDATA)>
<!ATTLIST job-scriptfiles
          if         CDATA   "1"
          machine    CDATA   "STAXJobScriptFileMachine"
>

<!--
     The job-action element specifies a task to be executed after the
     sub-job has started. This task will be executed in parallel with
     the sub-job via a new STAX-Thread. The task will be able to use the
     STAXSubJobID variable to obtain the sub-job ID in order to interact
     with the job. If the job completes before the task completes, the
     job will remain in a non-complete state until the task completes.
     If the job cannot be started, the job-action task is not executed.
-->
<!ELEMENT job-action         (%task;)>
<!ATTLIST job-action
          if        CDATA    "1"
>

<!--================= The Call-With-List Element =================== -->
<!--
     Perform a function with the referenced name with any number of
     arguments in the form of a list.  The function attribute value
     and argument values are evaluated via Python.
-->
<!ELEMENT call-with-list      (call-list-arg*)>
<!ATTLIST call-with-list
          function   CDATA    #REQUIRED
>

<!ELEMENT call-list-arg       (#PCDATA)>

<!--================= The Testcase Status Element ================== -->
<!--
     Marks status result ('pass' or 'fail') for a testcase and
     allows additional information to be specified.  The status
     result and the additional info is evaluated via Python.
-->
<!ELEMENT tcstatus   (#PCDATA)>
<!ATTLIST tcstatus
          result     CDATA  #REQUIRED
```

```
>

<!--================= The Return Element =========================== -->
<!--
     Specifies a value to return from a function.
-->
<!ELEMENT return     (#PCDATA)>

<!--================= The Release Element ========================== -->
<!--
     The release element specifies to release a block in the job.
-->
<!ELEMENT release    EMPTY>
<!ATTLIST release
          block      CDATA    #IMPLIED
>

<!--================= The Rethrow Element ========================= -->
<!--
     The rethrow element specifies to rethrow the current exception.
-->
<!ELEMENT rethrow      EMPTY>

<!--================= The Raise Element ============================ -->
<!--
     A raise signal element raises a specified signal.
     Signals can also be raised by the STAX execution engine.
     The signal attribute value is evaluated via Python.
-->
<!ELEMENT raise      EMPTY>
<!ATTLIST raise
          signal     CDATA        #REQUIRED
>

<!--================= The Call Element ============================= -->
<!--
     Perform a function with the referenced name.
     The function attribute value is evaluated via Python.
     Arguments can be specified as data to the call element.
     Arguments are evaluated via Python.
-->
<!ELEMENT call       (#PCDATA)>
<!ATTLIST call
          function   CDATA    #REQUIRED
>

<!--================= The Loop Element ============================= -->
<!--
     The loop element performs a task a specified number of times,
     allowing specification of an upper and lower bound with an
     increment value and where the index counter is available to
     sub-tasks.  Also, while and/or until expressions can be
     specified.
-->
<!ELEMENT loop       (%task;)>
```

```
<!-- var      is the name of a variable which will contain the loop
              index variable.  It is a literal.
     from     is the starting value of the loop index variable.
              It must evaluate to an integer value via Python.
     to       is the maximum value of the loop index variable
              It must evaluate to an integer value via Python.
     by       is the increment value for the loop index variable
              It must evaluate to an integer value via Python.
     while    is an expression that must evaluate to a boolean value
              and is performed at the top of each loop.  If it
              evaluates to false, it breaks out of the loop.
     until    is an expression that must evaluate to a boolean value
              and is performed at the bottom of each loop.  If it
              evaluates to false, it breaks out of the loop.
-->
<!ATTLIST loop
          var         CDATA    #IMPLIED
          from        CDATA    '1'
          to          CDATA    #IMPLIED
          by          CDATA    '1'
          while       CDATA    #IMPLIED
          until       CDATA    #IMPLIED
>

<!--================= The Script Element ============================ -->
<!--
     Specifies Python code to be executed.
-->
<!ELEMENT script     (#PCDATA)>

<!--================= The Parallel Element ========================== -->
<!--
     The parallel element performs one or more tasks in parallel.
-->
<!ELEMENT parallel   (%task;)+>
```

Appendix E: STAX Extensions Document Type Definition (DTD)

This section contains the DTD for defining STAX Extensions to be registered for the STAX service.

```
<!--
   STAX Extensions Document Type Definition (DTD)

   This DTD module is identified by the SYSTEM identifier:

     SYSTEM 'stax-extensions.dtd'

   This DTD is used for files specified using the EXTENSIONXMLFILE
   parameter when registering the STAX service with extensions.

-->

<!--================== STAX Extension File Definition =============== -->
<!--
     The root element extensions contains all other elements.  It
     consists of one or more extension elements.
-->
<!ELEMENT stax-extensions    (extension+)>

<!--================== The Extension Element ========================= -->
<!--
     Specifies a STAX extension.  It can consist of 0 or more
     parameter elements, followed by 0 or more include-element or
     0 or more exclude-element elements.
-->
<!ELEMENT extension          (parameter*,
                              (include-element* | exclude-element*))>
<!ATTLIST extension
          jarfile            CDATA   #REQUIRED
>

<!--================== The Parameter Element ========================= -->
<!--
     Specifies a parameter for a STAX extension.
-->
<!ELEMENT parameter          EMPTY>
<!ATTLIST parameter
          name               CDATA   #REQUIRED
          value              CDATA   #REQUIRED
>

<!--================== The Include Element ========================= -->
<!--
     Specifies to only register this element for a STAX extension
     instead of registering all elements specified in the extension
     jar file's manifest file.
-->
<!ELEMENT include-element    EMPTY>
<!ATTLIST include-element
          name               CDATA   #REQUIRED
>

<!--================== The Exclude Element ========================= -->
<!--
     Specifies to excluce registering this element for a STAX extension
     instead of registering all elements specified in the extension
     jar file's manifest file.
-->
<!ELEMENT exclude-element    EMPTY>
<!ATTLIST exclude-element
          name               CDATA   #REQUIRED
>
```

Appendix F: References

- 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 the http://www.xml.org/xml/resources_focus_beginnerguide.shtml website for an XML Beginner's Guide.
- For more information about the XML Parser for Java used by STAX to validate and parse XML documents:
  - See the http://w3.xml.ibm.com/xml4j website.
  - Otherwise, see the http://xml.apache.org/xerces2-j website.

Appendix G: Jython and CPython Differences

Although in most cases Jython behavior is identical to the C-language implementation of Python (CPython), there are still cases where the two implementations differ. If you are already a CPython programmer, or are hoping to use CPython code under Jython, you need to be aware of these differences. Also, there is a time lag between a new CPython release and the corresponding Jython release. STAX uses Jython 2.1 which is based on Python 2.1. Jython 2.1 cannot execute Python code that uses functions that were provided in later versions of Python, such as Python 2.2.

Most Python modules that are written in Python work fine in Jython. A few types of modules will not run under Jython such as:

- Modules that contain functionality not included in a JVM

  Some standard CPython modules depend on operating system calls that are not available under Java. The most notable of these is *os*, which actually does run in Jython, but is missing much of its functionality.

- Modules that are implemented in C

  A number of common CPython modules are implemented in C rather than Python, either for a speed boost or because the module is a C wrapper around an external C library. The C modules, or any modules that depend on them, will not run in Jython.

See the "Jython Essentials" book, written by Samuele Pedroni and Noel Rappin, for more information about the differences between Jython and CPython.

Appendix H: Licenses and Acknowledgements

Jython

Jython is an implementation of the high-level, dynamic, object-oriented language Python written in 100% Pure Java, and seamlessly integrated with the Java platform. It thus allows you to run Python on any Java platform.

Acknowledgement

This product includes software developed by the Jython Developers (http://www.jython.org/).

Licence

XML Parser for Java (Xerces)

XML Parser for Java is a validating XML parser and processor written in 100% pure Java; it is a library for parsing and generating XML documents. This parser easily enables an application to read and write XML data.

Acknowledgement

This product includes software developed by the Apache Software Foundation (http://www.apache.org/).

License