



Software- und Organisations-Service GmbH

# **Job Scheduler**

# **Event Processing**

**Technical Information**  
**12. Oktober 2009**

## Contact Information

Software- und Organisations-Service GmbH  
Giesebrechtstr. 15  
10629 Berlin  
Germany

Telephone +49 30 864790-0

Telefax +49 30 8613335

Mail [info@sos-berlin.com](mailto:info@sos-berlin.com)

Web <http://www.sos-berlin.com>

Revised: 06.10.2009

---

1	Introduction.....	4
1.1	Examples.....	5
2	Event Processing.....	7
2.1	Event Generators.....	8
2.2	Event Processors.....	15
2.3	Testing for events in a job chain.....	17
3	Synchronization of Job Chains.....	18
4	Event Handlers.....	19
4.1.1	Script-Based Event Handlers.....	20
4.1.2	XSL Event Handlers.....	22
4.1.3	XML Event Handlers.....	23
5	Event Monitoring.....	33
6	Event Processing Configuration.....	37
7	Event Handling Routine Examples.....	39
7.1	Split and Merg.....	39
7.2	Job A and Job B on Job Scheduler 1, then Job C on Job Scheduler 2.....	40
7.3	Job A has not run by 5 o'clock.....	40
7.4	Notification when a file has not arrived before 17:00.....	41
8	Appendix.....	42

# 1 Introduction

In its standard configuration, the Job Scheduler can process job chains that start jobs in a predictable order, this serialization of jobs being suitable for many applications and situations. However, job chains are not able to represent the dynamic dependencies that occur, for example, when jobs are dependent on several other jobs or are to be started either by external events or by jobs that are started by other Job Schedulers.

A solution for this problem is available for the Job Scheduler, in which centrally processed events can be triggered. This makes it possible to evaluate events originating in different sources and to initiate job starts or orders depending on these events.

The Job Scheduler can use events to control the progress of jobs and orders in job chains: jobs and job chains being started when one or more particular events occur.

Events are particularly suited for the implementation of procedures in which there are several followers and / or several forerunners.

Three different methods are available for handling events:

1. Processing in event handlers: this solution offers the greatest freedom in formulating complex conditions. See --> *"XML Event Handlers"*
2. Monitoring of events and appropriate handling using shell scripts: this method allows individual shell scripts and events to be generated and the existence of events tested. See --> *"Self-Written Script-Based Event Handlers"*
3. Job chain synchronization using the Synchronizer Job: this method allows the processing of a number of job chains to be synchronized with a minimum of configuration.  
--> See *"Job Chain Synchronization"*.

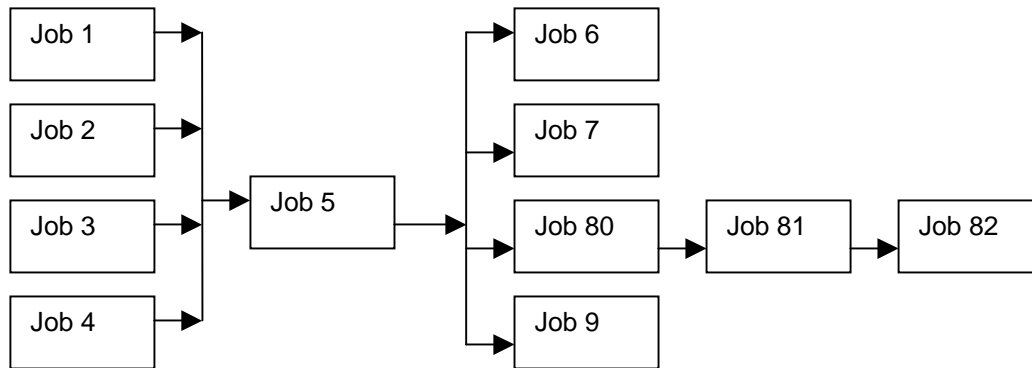
Example application: Consider the situation with two job chains - job chain A is used to process file orders and job chain B writes data to a database. Data for job chain B is written by a persistent order which is started at daily at 15.00. However, database entries can only be made once a particular file for job chain A has been received. On the other hand, it should be possible for the steps in both job chains up to the synchronization point to be carried out in any order and in parallel.

## 1.1 Examples

The following example shows a *Merge & Split* series:

The architecture shown in this example could be used, for example, when processing time-intensive jobs that can run in parallel. Parallel processing of jobs is often used to speed processing as a whole, with different jobs possibly being run on different Job Schedulers.

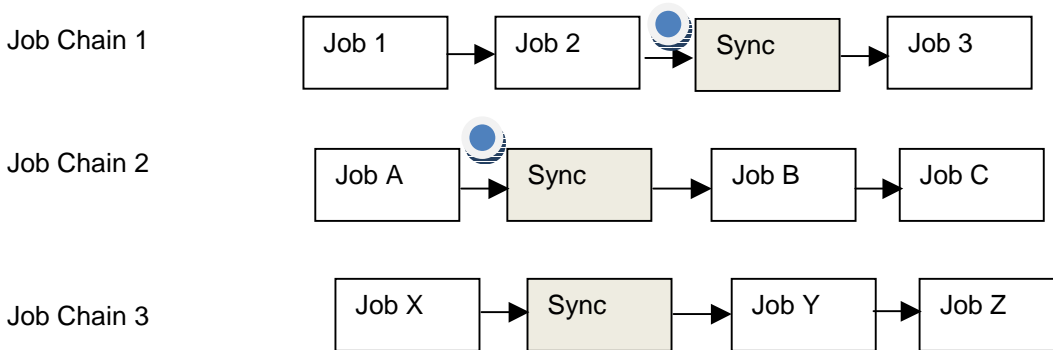
A further reason for such a processing architecture is to allow for delays to events: when jobs are serialized in job chains a delay in one event means that the whole chain would be delayed.




In this example, jobs “Job 1”, “Job 2”, “Job 3” and “Job 4” run at different times. However, “Job 5” will only be started when all four have been successfully completed. Finally, “Job 6”, “Job 7”, “Job 9” and the job chain comprising “Job 80”, “Job 81” and “Job 82” will be started. The process architecture just described will be used in the course of this documentation to illustrate the use of event processing.

## Job Chain Synchronization

Consider a number of job chains, which are to be synchronized at a particular point in each chain. In other words, the orders in a job chain will only be processed beyond the synchronization point once all the orders in the other job chains have been processed up to the synchronization point.



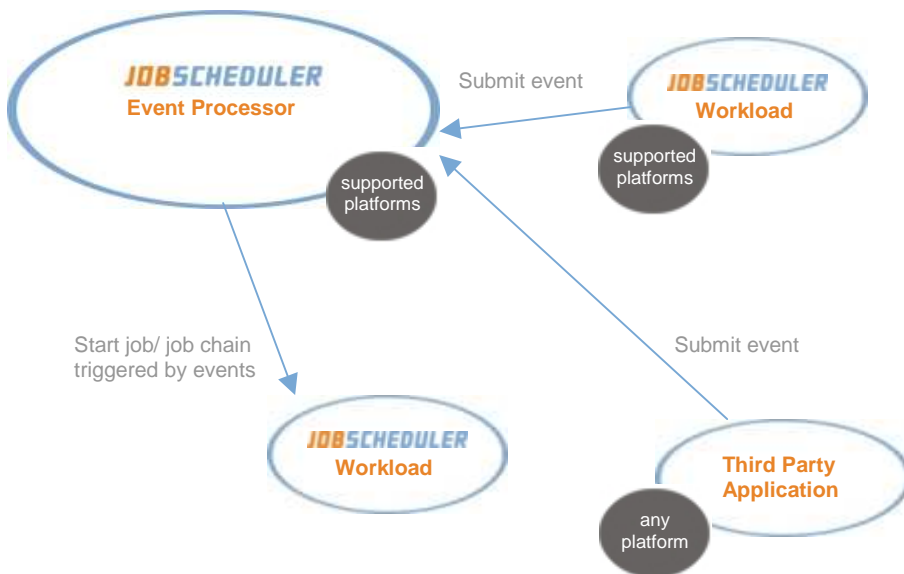
Both orders  will wait at the synchronization point until all job chains have an order. This means that job chains 1 and 2 will not be processed as long as job chain 3 does not have an order.

## 2 Event Processing

The expressions:

- event generators,
- event processors and
- event handler

will be explained in this chapter.



## 2.1 Event Generators

Events need to be created before they can be processed by the Job Scheduler. This can be carried out by one of a number of possible event sources (event generators):

- job monitors in a job chain,
- standard jobs that generate events,
- scripts or
- event handling routines.

All event generators share the principle that they send an order to the Job Scheduler Event Processor via TCP/IP. This processor then makes an entry for each order in the event database table as well as holding them in a global → Job Scheduler variable. The Event Generator Order contains a description of the event as well as specifying the action to be carried out. The `add` action is used to insert events in the database table and `remove` is used to remove all the events possessing a particular attribute (e.g. all events belonging to a particular class).

In principle, any software could be used to generate events, what is important is that the order is correctly generated and send to the relevant supervising Job Scheduler.

Events possess the following attributes:

Name	Description	Default
event-id	Must only be unique in conjunction with the event-class	
exit-code	The current exit code of the job script running (<script>)	the script exit code
event-class	Results in a unique identifier when used together with the event-id. Can, for example, be used to remove events (remove all events of a particular class)	
job-name	The name of the job currently running	the current job name
job-chain	The name of the current job chain	the current job chain
order-id	The identifier of the current order	the current order identifier (order id)
name	The event name	
creation-date	The timestamp from the moment at which the event was generated	now
expiry-date	The timestamp for the moment when the event expires. If this attribute is not set, then the event lifetime is set according to the value set in the Event Processor (default=24h).	00.00 on the following day

The event class must be specified when an event is generated – all other attributes are optional.

Events can be created using the following *event generators*:

## Event Generator Jobs

The standard *JobSchedulerSubmitEventJob* job can be used to set off an event. An event generated by this job will be forwarded to a Job Scheduler Supervisor1. If a Supervisor has not been configured then the event will be forwarded to the Job Scheduler that ran the event processor job. Both the order and job parameters of this generating job can be used to configure the event. This generating job can either be run in a job chain or 'stand-alone'.

→ See the job documentation in *./jobs/JobSchedulerSubmitEventJob.xml*

The following job generates an event with the `myClass` class and `myId` ID.

```
<job name = "JobSchedulerSubmitEventJob"
  title = "Submit Events"
  order = "yes" >

  <description>
    <include file = "jobs/JobSchedulerSubmitEventJob.xml" />
  </description>

  <params>
    <param name = "scheduler_event_class" value = "myClass" />
    <param name = "scheduler_event_id" value = "myId" />
  </params>

  <script language = "java"
    java_class = "sos.scheduler.job.JobSchedulerSubmitEventJob" />
</job>
```

Example: an event causing "It is now 17:00" to be printed. This is done by configuring a job that has 17.00 as its starting time.

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<job name="sample"
  title="Submit Event &quot;17:00 Uhr&quot;">
  <description>
    <include file="jobs/JobSchedulerSubmitEventJob.xml"/>
  </description>
  <params>
    <param name="scheduler_event_class"
      value="myClass"/>
    <param name="scheduler_event_id"
      value="myId1"/>
  </params>
  <script language="java"
    java_class="sos.scheduler.job.JobSchedulerSubmitEventJob"/>
  <run_time>
    <period single_start="17:00"/>
  </run_time>
</job>
```

This job can, for example, be configured as a step in a job chain in order to use the event to document the progress of an order. This use of an event is illustrated in the following example, which is based on a job chain consisting of three steps. The event is to be generated once step "100" has been completed.

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<job_chain name="job_chain1"
  orders_recoverable="yes"
  visible="yes">
  <job_chain_node state="100"
    job="anyJob"
    next_state="200"
    error_state="error"/>
  <job_chain_node state="200"
    job="firstStepIsReady"
    next_state="300"
    error_state="error"/>
  <job_chain_node state="300"
    job="anyOtherJob"
    next_state="success"
    error_state="error"/>
  <job_chain_node state="error"/>
  <job_chain_node state="success"/>
</job_chain>

<?xml version="1.0" encoding="ISO-8859-1"?>
<job name="firstStepIsReady"
  title="Submit Event &quot;Step 100 is ready&quot;"
  order="yes">
  <description>
    <include file="jobs/JobSchedulerSubmitEventJob.xml"/>
  </description>
  <params>
    <param name="scheduler_event_class"
      value="myClass"/>

    <param name="scheduler_event_id"
      value="myId2"/>
  </params>
  <script language="java"
    java_class="sos.scheduler.job.JobSchedulerSubmitEventJob"/>
  <run_time/>
</job>
```

## Event Generator Monitors

A monitor has the same functions as a job, but its implementation as a monitor allows its use with any other jobs. See the job documentation in *./jobs/JobSchedulerSubmitEventJob.xml*.

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<job title="Erzeugt einen Event mit einem Monitor"
      name="job2_event_by_monitoing">
  <params>
    <param name="scheduler_event_id"
           value="2"/>

    <param name="scheduler_event_class"
           value="monitorEvent"/>
  </params>
  <script language="javascript">
    <![CDATA[
function spooler_process(){
  spooler.log.info("Here is a event coming from a monitor");
  return false;
}
    ]]>
  </script>
  <monitor name="create_event"
          ordering="0">
    <script java_class="sos.scheduler.job.JobSchedulerSubmitEventMonitor"
           language="java"/>
  </monitor>
  <run_time/>
</job>
```

## Event Generator Scripts

The `jobscheduler_event.sh` and `jobscheduler_event.cmd` scripts are provided for use on Unix and Windows systems respectively. Calling either of these scripts causes an event to be generated. These scripts can, for example, be incorporated in existing shell scripts; called either as a separate step in a job chain or called as a standalone job.

**The following example is for a shell job that generates an event with the class “example”:**

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<job order="yes"
  name="file_job">
  <script language="shell">
    <![CDATA[
      @echo off
      @rem submit event to Supervisor Job Scheduler
      %SCHEDULER_HOME%/bin/jobscheduler_event.cmd -x %ERRORLEVEL% -e example
    ]]>
  </script>
  <run_time/>
</job>
```

**The following parameters can be specified in these scripts:**

`jobscheduler_event.sh` - submit event to Job Scheduler

-x	exit-code	Use \$? to specify the exit code of a previous Unix command.
-e	event-class	Specifies a common name for a set of events that enabling <i>event handlers</i> to process multiple events of the same class. For example, "daily_closing" could be an event class for jobs that should start once day-time business processes have drawn to a close.
-i	event-id	An identifier for an event. Allows event handlers to react to events having a particular ID. The ID is unique for an event class.
-c	job-chain	The name of the job chain. If empty, the name of the current job chain is used.
-o	order-id	If the current job is executed within a job chain then its Order ID can be used to identify the event. By default, the SCHEDULER_ORDER_ID environment variable is set by the Job Scheduler if this option is not specified.
-j	job-name	The name of the current job. By default, the SCHEDULER_ORDER_ID environment variable is set. This environment variable is automatically set by the Job Scheduler
-h	workload-job-scheduler-host	Specifies the local Job Scheduler Workload instance host name. The SCHEDULER_HOST environment variable is used by default.
-p	workload-job-scheduler-port	Specifies the local Job Scheduler Workload instance host name. The SCHEDULER_TCP_PORT environment variable is used per default.
-s	supervisor-job-scheduler-host	Specifies the Job Scheduler Supervisor instance host name. The SCHEDULER_SUPERVISOR_HOST environment variable is used by default. Workload Job Scheduler instances automatically register at a Supervisor in order to synchronize job configurations. The Supervisor instance receives events,

		executes the event handler and starts jobs and job chains.
-r	supervisor-job-scheduler-port	Specifies the port number on which the Job Scheduler Supervisor instance is operating. By default the SCHEDULER_SUPERVISOR_PORT environment variable is used.
-v	supervisor-job-chain	Specifies the name of the job chain in the Job Scheduler Supervisor instance that implements the event processor. The default value is scheduler_event_service.
-w	what	Events can be added ( <code>add</code> ), deleted ( <code>remove</code> ) and checked ( <code>check</code> ). Die Default-Aktion ist <code>add</code> . <code>check</code> is used to determine whether one or more particular events are present. Here at least one of the options <code>-i</code> <code>-e</code> <code>-x</code> or <code>-a</code> must be specified, in order to determine the event being sought after.
-t   -td	expiration-date	Specifies the point in time at which an event automatically expires: this parameter being implemented by the Event Processor.  A specific point in time can be specified in the ISO date format, i.e. <code>yyyy-MM-dd hh:mm:ss</code> .  The value <code>never</code> ensures that the event does not ever expire. The options <code>expiration-date</code> , <code>-cycle</code> and <code>-period</code> may not be used at the same time.
-tc	expiration-cycle	Specifies a time of the current or the next day at which the event expires.  The cycle is specified in the format <code>hh:mm[:ss]</code> . May not be used together with expiration date or period.
-tp	expiration-period	Specifies a period after which the event expires.  The period is specified in the format <code>hh:mm[:ss]</code> . May not be used together with expiration date or cycle.
-d	name=value	Allows additional parameters for event handlers to be specified. Parameters are created using name-value pairs, separated by an equals sign. Parameter names can be freely chosen and event handlers configured to take account of values handed over. This option can be specified as often as required, allowing the definition of any number of parameters.
-a	xpath-expression	All events corresponding to the XPath expression specified when this parameter is set. Complex expressions are possible and all the attributes of an event can be considered. This parameter allows complex queries to be made, that would not be possible with the <code>-e</code> <code>-i</code> and <code>-x</code> options.
allowed-exit-code	...	Enables the specification of a list of exit codes that mean that a job should be considered as having run successfully. This is useful if job scripts provide return values in the form of exit codes and these codes should not be considered as errors.

The `jobscheduler_event.sh` script forwards events to the event processor in the Job Scheduler Supervisor instance. These events are then processed by event handlers in the event processor and are then used to trigger jobs and job chains depending on the parameters in the script. Suitably configured, event handlers are able to take account of situations that require multiple events to be fired in order to trigger a job start.

Should a Job Scheduler Supervisor not be accessible then the event data will be stored in a local file: `./logs/scheduler.events`. These events will be dequeued by the `scheduler_dequeue_events` job, which is one of the standard automation jobs provided with the Job Scheduler distribution, as soon as the Supervisor Job Scheduler is available again.

### Examples:

Create an event with the minimum parameters required for an event class (*event id* and *job name*):

```
./jobscheduler_event.sh -x $? -e daily_closing -i my_id -j my_job
```

Specify a different host name and port for the Job Scheduler Supervisor and define a parameter `proc1` with the value 42:

```
./jobscheduler_event.sh -x $? -e daily_closing -i my_id -j my_job -s master -r 4444 -d proc1=42
```

### Environment

<code>SCHEDULER_HOME</code>	Installation directory of the Job Scheduler Workload instance.
<code>SCHEDULER_HOST</code>	Host name of the Job Scheduler Workload instance.
<code>SCHEDULER_TCP_PORT</code>	Port number of the Job Scheduler Workload instance.
<code>SCHEDULER_JOB_CHAIN</code>	Name of the job chain currently being executed, should the current job be part of a job chain.
<code>SCHEDULER_ORDER_ID</code>	Id of the order currently being executed, should the current job be part of a job chain.
<code>SCHEDULER_SUPERVISOR_HOST</code>	Hostname of the Job Scheduler Supervisor instance.
<code>SCHEDULER_SUPERVISOR_PORT</code>	Port number on which the Job Scheduler Supervisor instance operates.

### Diagnostics:

The following error messages may be written to `stderr`:

<code>ERROR-001: no host has been specified</code>	The host name for the Job Scheduler Supervisor instance has not been specified.
<code>ERROR-001: no port has been specified</code>	The port number for the Job Scheduler Supervisor instance has not been specified.
<code>ERROR-010: could not connect to host</code>	The specified host name or IP address is invalid or no Job Scheduler is running via the specified port.

### Event Generator Event Handler (see Section 2.3)

New events can be generated by event handlers. To do this the event handler executes the `<add_event>` element.

```
<commands>
  <add_event>
    <event event_id="7" event_class="sample" event_title="Neues Ereignis" />
  </add_event>
</commands>
```

## 2.2 Event Processors

Event Processors carry out two tasks:

1. Recording the event and saving it in the database as well as in a Job Scheduler variable;
2. Execution of event handling routines (Event Handlers) and the evaluation of conditions.

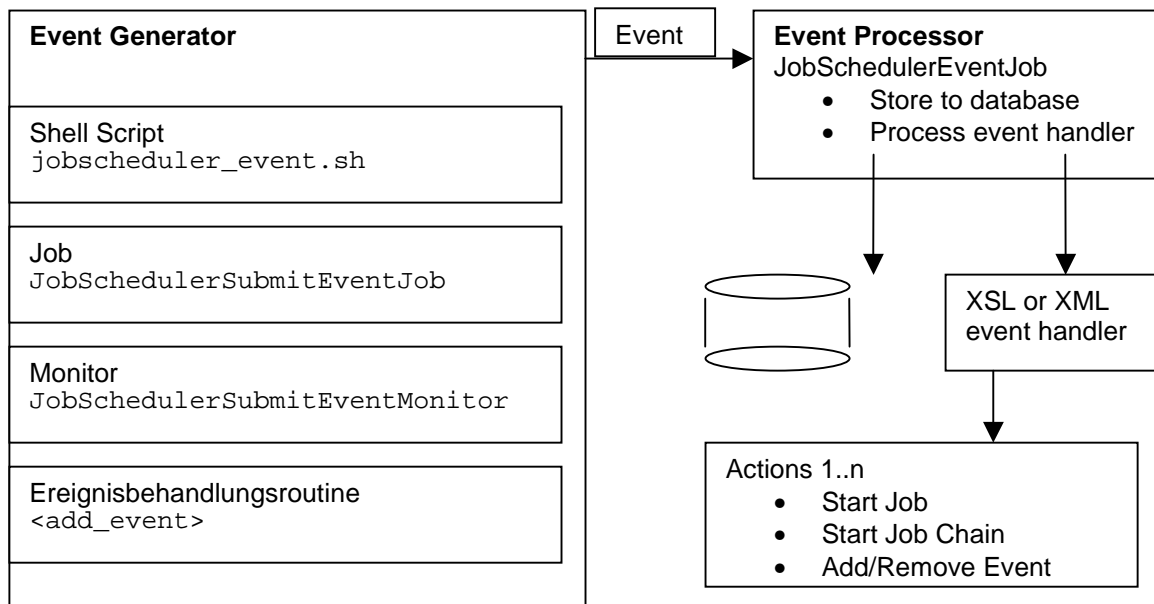
Event handlers consist of:

- a *set of rules*, to test whether an event has occurred and
- a list of *actions* that are to be carried out when the event conditions are met. The following actions are possible:
  - starting a job,
  - starting a job chain,
  - generating a new event,
  - deleting events.

Event handlers can either be in XML format or in XSL in the form of stylesheets. Event handling routines in XML format are supported by the Job Scheduler *Job Editor*, which can be used to write and edit routines.

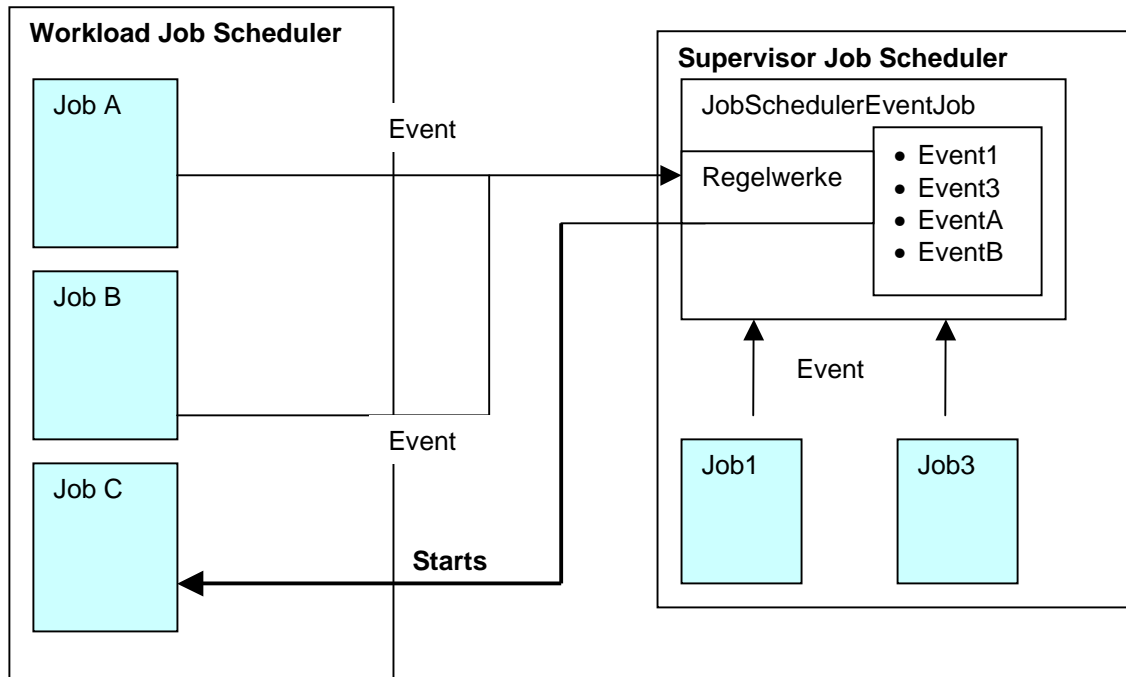
→ See Chapter 2.3

*Event Generators* can also generate events. These are collected in the *Event Processor* and one or more event handlers will be executed by the Event Processor each time an event occurs. The conditions used to determine whether one or more events have taken place are tested whilst the Event Handler is being executed. The actions specified for the event will be carried out if these tests are positive.



The *Event Processor* consists of a job chain made up of the `scheduler_event_service` job and the `scheduler_event_service` job chain. The *event processor* definition can be found in the `./config/scheduler_event.xml` file.

The job chain contains the `scheduler_event` job. The parameterization of this job is described in the `scheduler/jobs/JobSchedulerEventJob.xml` documentation.



### Summary: example system architecture with two Job Schedulers

In this example, Jobs A and B are started on a Workload Job Scheduler, with each of these jobs causing an event. These events are processed by the Supervisor Job Scheduler Event Processor. Within the Event Processor, the event processing Rules Set is processed by the Event Handler. Jobs 1 and 3 are run by the Supervisor Job Scheduler and also cause events. The event processor would then start, for example, Job C when a rule set returns TRUE.

Events have a limited lifetime, which can be set when they are generated: the default setting being that events expire at 00:00 – i.e. at midnight. As long as an event has not expired, it is possible for it to be explicitly deleted by an event handling routine. All events are saved in a database and remain available after the Job Scheduler has been restarted.

## 2.3 Testing for events in a job chain

Sometimes it is necessary to test within a job chain for the presence of one or more events and to make further processing of the chain dependent on the results of these tests. The *JobSchedulerExistsEventJob* job is provided for this purpose.

This job tests whether particular events exist. It does this by processing orders that contain specification(s) of the event(s). Depending on whether or not the events exist, the order will either be set to the `next_state` or to the `error_state`.

The specification of the events to be tested is saved in the `scheduler_event_spec` parameter. This parameter contains an XPath expression, which will be applied to the XML representation of the events. If the XPath expression returns a result, then the order will be set to the `next_state`. If the XPath expression does not return a result, then the order will be set to the `error_state`.

An example for the use of this job can be found in the Appendix.

The documentation for this job can be found in the Job Scheduler installation directory under *./jobs/JobSchedulerExistsEventJob.xml*.

### Examples:

```
//event[@event_class='foo']
```

Successful, if an event with the "foo" event class is found

```
//events[event[@event_class='foo'] and events[event[@event_class='bar']]]
```

Successful, if an event with the "foo" event class is found along with one with the "bar" class

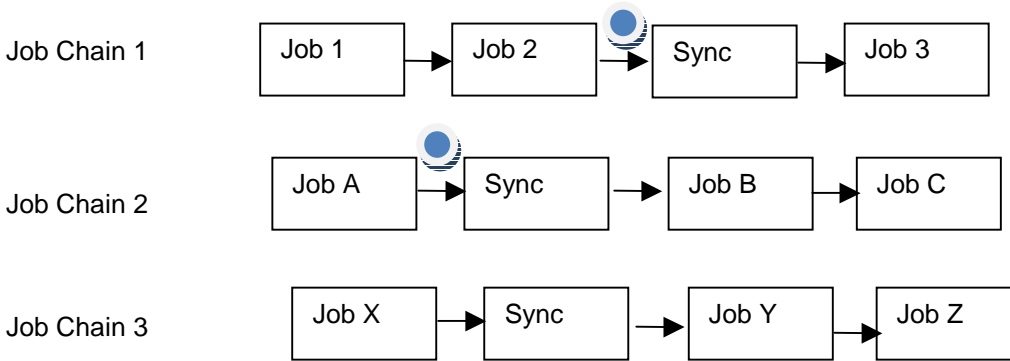
```
//events[not(event[@event_class='foo'])]
```

Successful, if no event with the "foo" event class is found

### 3 Synchronization of Job Chains

The synchronization of job chains is a standard task that in a simplified form can be configured without the use of events:

Consider a number of job chains, that are to be synchronized at a particular point in each chain. In other words, the orders in a job chain will only be processed beyond the synchronization point once all the orders in the other job chains have been processed up to the synchronization point.



Both orders will wait at the synchronization point until all job chains have an order. This means that job chains 1 and 2 will not be processed as long as job chain 3 does not received an order.

The number of orders that are synchronized is specified in the Sync-Job (default=1). This means that, for example, if a new order arrives in Job Chain 2 (meaning that there are now two orders in this job chain), and then an order in Job Chain 3, then only one order will be processed in each job chain. In this case, the first order in Job Chain 2 will be processed along with the order in Job Chain 3. The second order in Job Chain 2 will remain at the synchronization point until a further order arrives in Job Chain 3.

More information about this procedure can be read under :

[http://jobscheduler.sourceforge.net/osource\\_scheduler\\_howto\\_split\\_merge\\_en.htm](http://jobscheduler.sourceforge.net/osource_scheduler_howto_split_merge_en.htm).

Configuration of the Sync-Job:

```

<?xml version="1.0" encoding="ISO-8859-1"?>
<job order="yes"
  stop_on_error="no">
  <!--
  <params>
    <param name="ChainA_required_orders"
      value="2" />

    <param name="ChainB_required_orders"
      value="3" />
  </params>
  -->
  <script java_class="sos.scheduler.job.JobSchedulerSynchronizeJobChains"
    language="java" />
  <delay_order_after_setback setback_count="1"
    is_maximum="no"
    delay="00:02" />
  <delay_order_after_setback setback_count="200"
    is_maximum="yes"
    delay="00:02" />

  <run_time />
</job>
  
```

## 4 Event Handlers

All event handlers basically function by searching for and determining whether or not events exist. All existing events are held in a global Job Scheduler variable. The content of the Job Scheduler variables – in this case the list of the events currently existing – can be found in the event processor log file.

Further information can be written in the *scheduler/logs/task.scheduler\_event\_service.log* file when the debug-level in the *./config/factory.ini* file is set to a value > 2. This is done in the [spooler] section of the file and a typical value would be debug3. Settings made in this section of the *scheduler/config/factory.ini* file are applicable to all jobs. When, however, the debug level is set in the [scheduler\_event\_service] section, the settings made will only apply to jobs being run by the Event Processor.

With a debug3 level, the current task XML structure will be written in the *event processor* log file (*scheduler/logs/task.scheduler\_event\_service.log*). A typical XML structure would be:

```
<events current_date="2008-05-16 11:14:01"
        expiration_date="2008-05-16 23:14:01">

<event created="2008-05-16 11:13:29"
        event_class="example"
        event_id="0"
        exit_code="0"
        expires="2008-05-16 23:13:30"
        job_chain=""
        job_name="simple_shell_job"
        order_id=""
        remote_scheduler_host="wilma"
        remote_scheduler_port="4444"
        scheduler_id="scheduler.supervisor"/>

<event created="2008-05-16 11:14:01"
        event_class="example"
        event_id="0"
        exit_code="0"
        expires="2008-05-16 23:14:01"
        job_chain="txt_chain"
        job_name="file_job"
        order_id="files/in/sample2.txt"
        remote_scheduler_host="wilma"
        remote_scheduler_port="4444" scheduler_id="scheduler.supervisor"/>
</events>
```

The *Event Processor* first of all determines and executes all the event handlers. It recognizes two types of event handlers:

- XML Event Handlers
- XSL Event Handlers

XML event handlers are written and modified using the Job Scheduler Job Editor and define the event processing rules and the corresponding actions.

XSL event handlers are processed using an XSLT style sheet transformation. This transformation results in commands for one or more Job Schedulers.

All event handlers are stored in a central directory, the name of which is stored in the event processor *event\_handler\_filepath* job parameter.

### 4.1.1 Script-Based Event Handlers

Events can be monitored using shell scripts. This means that it is possible to implement checks in shell scripts and use the possibilities provided by shell scripts to handle the events. In this case, the event processor is only used to record and save the events. Either the `jobscheduler_event.cmd` or the `jobscheduler_event.sh` shell scripts are used to determine whether or not a particular event is present. In this case there are two possible ways of checking whether the event exists:

1. Direct testing of the `-e` and `-i` attributes for events,
2. testing with XPath expressions.

The `-w` option is set to `check`. In addition, the attributes that are to be sought for can be set. The script writes the number of events found to `stdout`. In addition, the `%JOB_SCHEDULER_COUNT_EVENTS%` environment variable is set on Windows systems.

#### Example queries:

How many "foo" class events have occurred?

```
call c:\scheduler\bin\jobscheduler_event.cmd -h localhost -p 4454 -w check -e foo
```

How many "foo" class events have occurred (with XPath)?

```
call c:\scheduler\bin\jobscheduler_event.cmd -h localhost -p 4454 -w check
-a "//events/event[@event_class='foo']"
```

Has an event occurred with both the "foo" and the "bar" classes?

```
call c:\scheduler\bin\jobscheduler_event.cmd -h localhost -p 4454 -w check
-a "//events[event/@event_class='foo' and event/@event_class='bar']"
```

Has no "foo" class event occurred?

```
call c:\scheduler\bin\jobscheduler_event.cmd -h localhost -p 4454 -w check
-a "//events[not(event/@event_class='foo')]"
```

#### An example split & merge script

In this example, one script is used to start three further scripts in parallel (split). It is only after all three scripts have been completed that processing will be continued (erge). The three scripts running in parallel each signal their completion by generating an event.

##### 1. The main script

The implementation of the main script is defined in a job. The scripts called by this script do not have to be present on the Job Scheduler as jobs.

```
@echo off
@echo This is Script A.
@rem ----- Begin split
@echo Split is now taking place.
start b.cmd
start c.cmd
start d.cmd
@rem ----- End split

@rem ----- Begin merge
:merge
call c:\scheduler\bin\jobscheduler_event.cmd -h localhost -p 4444 -w check -e test >nul
@echo ++%JOB_SCHEDULER_COUNT_EVENTS%+ class test events found.
@rem Sleep 10 seconds
ping -n 11 localhost > nul
@rem Number of events present determined
if %JOB_SCHEDULER_COUNT_EVENTS% lss 3 goto merge

@rem ----- End merge
```

```
@echo Merge has been completed.
@rem Delete events
call c:\scheduler\bin\jobscheduler_event.cmd -h localhost -p 4444 -w remove -e test
```

## 2. Script B (similar to scripts C and D)

```
@echo off
@echo This is Script B.
... (processing)
@rem Generate result
call c:\scheduler\bin\jobscheduler_event.cmd -h localhost -p 4444 -w add -i B -e test
```

The same example could also, for example, have been configured for 4 jobs A, B, C & D instead of the three described above. All scripts are defined as jobs. In this case the implementation of Job A would look like:

```
<job>
  <script language="shell"><![CDATA[
    @echo off
    @echo This is Script A.
    @rem ----- Begin split
    @echo Split is now taking place.
    call c:\sos\scheduler\bin\jobscheduler.cmd command "<start_job job='test/jobB'/">
    call c:\sos\scheduler\bin\jobscheduler.cmd command "<start_job job='test/jobC'/">
    call c:\sos\scheduler\bin\jobscheduler.cmd command "<start_job job='test/jobD'/">
    @rem ----- End split

    @rem ----- Begin merge
    :merge
    call c:\scheduler\bin\jobscheduler_event.cmd -w check -e test > nul
    @echo ++%JOB_SCHEDULER_COUNT_EVENTS%+ class test events found.
    ping -n 11 localhost > nul
    if %JOB_SCHEDULER_COUNT_EVENTS% lss 3 goto merge

    @rem ----- End merge
    @echo Merge has been completed.
    call c:\scheduler\bin\jobscheduler_event.cmd -w remove -e test
  ]></script>
  <run_time/>
</job>
```

The implementation of Job B and correspondingly Jobs C and D would look like:

```
<job>
  <script language="shell"><![CDATA[
    @echo off
    @echo This is Script B.
    call c:\scheduler\bin\jobscheduler_event.cmd -w add -i B -e test
  ]></script>
  <run_time/>
</job>
```

### 4.1.2 XSL Event Handlers

Event handlers will be executed when their names correspond with the regular expression defined in the *event\_handler\_filespec* job parameter.

In addition, the Event Processor job processes job stylesheets for particular jobs, job chains or event classes according to the following name convention:

- *[job\_name].\*.job.xml* will be executed for events belonging to the *[job\_name]* job;
- *[job\_chain\_name].\*.job\_chain.xml* will be executed for events belonging to a job in the *[job\_chain\_name]* job chain;
- *[event\_class].\*.event\_class.xml* will be executed for events belonging to the *[event\_class]* class.

XSL event handlers must be manually written using a text editor. The event handler in the example below executes the list of all known events (see above).

The following example shows an event handler that searches for three events: an event from the *simple\_shell\_job* job and an event from each of the two job chains listed.

```
<xsl:template match="events[
  event[@job_name='simple_shell_job'] and
  event[@job_chain='txt_chain'] and
  event[@job_chain='pdf_chain'] ]">
```

Once these events have been found, two commands are sent to the Supervisor Job Scheduler. The first command starts the *samples/events/done\_job* job in the Workload Job Scheduler:

```
<xsl:call-template name="run_job">
  <xsl:with-param name="job">samples/events/done_job</xsl:with-param>
  <xsl:with-param name="host">localhost</xsl:with-param>
  <xsl:with-param name="port">4444</xsl:with-param>
</xsl:call-template>
```

The second command deletes the events, in order to ensure that they are not processed a second time.

```
<remove_event><event event_class="example"/></remove_event>
```

### 4.1.3 XML Event Handlers

The **Event Handler** directory is searched for events in the following order:

- `[job_name].*.job.actions.xml` will be executed for events belonging to the `[job_name]` job;
- `[job_chain_name].*.job_chain.actions.xml` will be executed for events belonging to a job in the `[job_chain_name]` job chain;
- `[event_class].*.event_class.actions.xml` will be executed for events belonging to the `[event_class]` class
- `[name].actions.xml` will be executed for all events.

If an event is created by the `job1` job and a file with the name `job1.job.actions.xml` exists, then this file will be executed.

XML event handling routines can be written and modified using the Job Scheduler Job Editor. In order to create new event handling routines, click on the “New” menu item and on “Event Handler”.

XML event handling routines consist of a *rule set* and a list of *commands*. The commands will be executed once the rule set is evaluated as being TRUE. Commands for the each Job Scheduler instance can be brought together in groups.

#### Example:

In this example, two command groups are described: in the first group, two commands for the `localhost:4454` Job Scheduler are defined and; in the second group one command for the `server:4444` Job Scheduler :

```
<commands>
  <command name="command_1" scheduler_host="localhost" scheduler_port="4454">
    <start_job job="job0" at="now" />
    <add_order job_chain="job_chain1" replace="yes" />
  </command>
  <command name="command_2" scheduler_host="server" scheduler_port="4444">
    <start_job job="job33" at="now" />
  </command>
</commands>
```

#### Rule Sets

The rule set consists of one or more event groups. An event group consists of a list of events combined together with a Boolean expression. The event groups are evaluated individually and the results of the evaluation combined using the Boolean expression. This determines whether or not the commands will be executed.

**Shortened notation:** When *all* the elements of a group are to be combined using OR or AND operators, then it is sufficient when the logic parameter is set to *or* or *and* as shown in the example below. This also applies for combination of groups.

**Example:**

Group 1: Event A and Event B  
Logic of group Test is A OR AND B

Group 2: Event C and Event D  
Logic of group Sample is C AND D

The groups are linked together using Test AND Sample

This means that, for example, when A is present, B missing and both C and D present, then the commands will be executed.

```
<events logic=Test and Sample>
  <event_group group="Test" logic="or">
    <event event_id="A"
      event_class="sample"
      event_title="Job1 ist gelaufen"
      job_name="job1" />
    <event event_id="B"
      event_class="sample"
      event_title="Job2 ist gelaufen"
      job_name="job2" />
  </event_group>
  <event_group group="Sample">
    <event event_id="C"
      event_class="sample"
      event_title="Job3 ist gelaufen"
      job_name="job3" />
    <event event_id="D"
      event_class="sample"
      event_title="Job4 ist gelaufen"
      job_name="job4" />
  </event_group>
</events>
```

**Example:** If the time is 17:00 or later and a particular step in a job chain has been completed, then the *dailyPrint* job should be started.

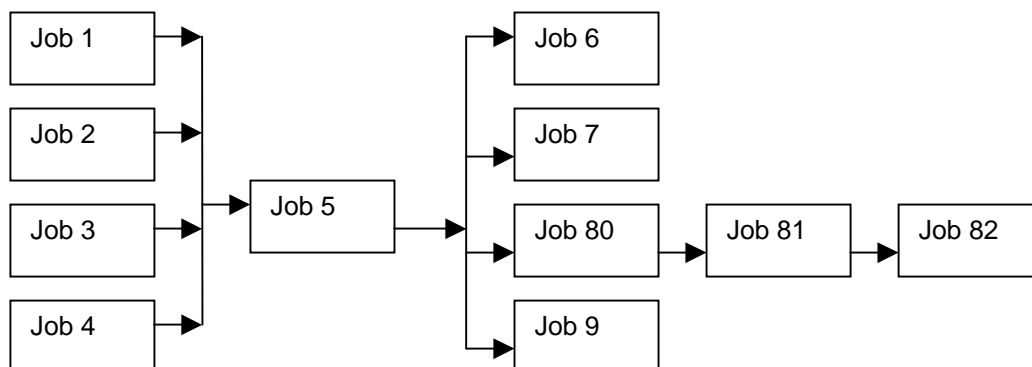
→Note that the configuration of the event generator for this example was described in Section 2.1

```
<?xml version="1.0" encoding="UTF-8"?>
<actions>
  <action name="SampleAction">
    <events>
      <event_group group="beforePrinting" logic="and">
        <event event_id="myId1" event_class="myClass" event_title="17:00 reached" />
        <event event_id="myId2" event_class="myClass" event_title="Step100 is ready" />
      </event_group>
    </events>
    <commands>
      <command name="command_1" scheduler_host="localhost" scheduler_port="4444">
        <start_job job="dailyPrint" />
      </command>
    </commands>
  </action>
</actions>
```

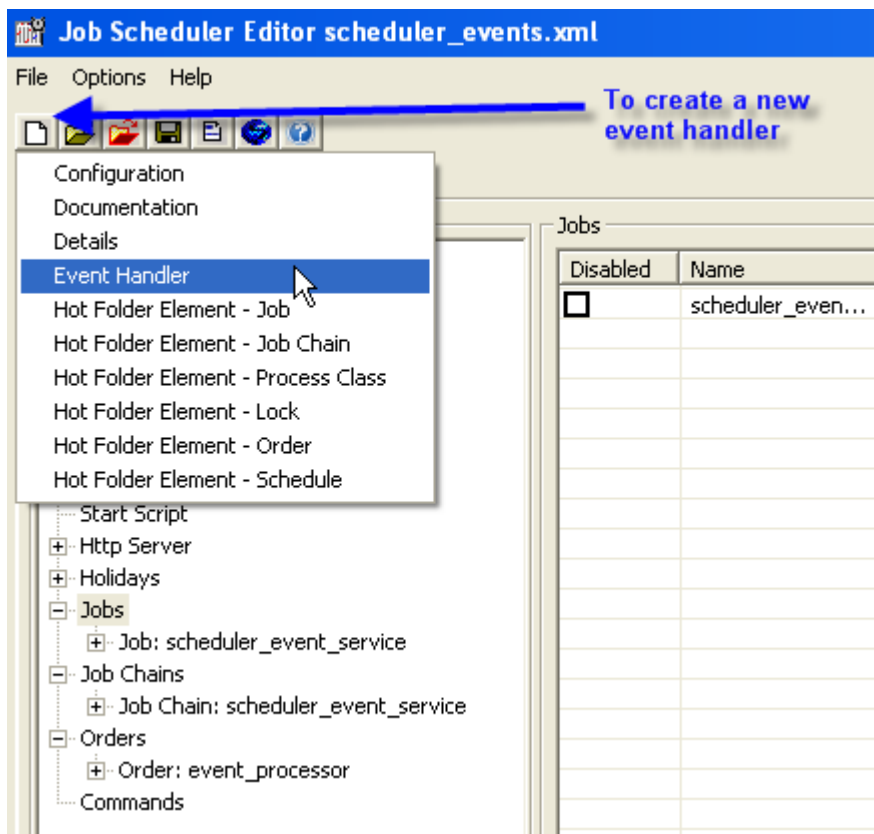
## Example for the scenario described in Chapter 1

In this example, jobs "Job 1", "Job 2", "Job 3" and "Job 4" run at different times. However, "Job 5" will only be started when all four have been successfully completed. Finally, "Job 6", "Job 7", "Job 9" and the job chain comprising "Job 80", "Job 81" and "Job 82" will be started.

```
<?xml version="1.0" encoding="UTF-8"?>
<actions>
  <action name="BeforeJob5">
    <events logic="and">
      <event_group group="pre" logic="and" event_class="pre">
        <event event_id="1" event_title="Job 1 is ready" job_name="job1" />
        <event event_id="2" event_title="Job 2 is ready" job_name="job2" />
        <event event_id="3" event_title="Job 3 is ready" job_name="job3" />
        <event event_id="4" event_title="Job 4 is ready" job_name="job4" />
      </event_group>
    </events>
    <commands>
      <command name="command_1" scheduler_host="localhost" scheduler_port="4444">
        <start_job job="job5" />
      </command>
    </commands>
  </action>
  <action name="AfterJob5">
    <events>
      <event_group group="post" event_class="post" logic="and">
        <event event_id="5" event_title="Job5 is ready" job_name="job5" />
      </event_group>
    </events>
    <commands>
      <command name="Start Job6" scheduler_host="localhost" scheduler_port="4444">
        <start_job job="job6" />
      </command>
      <command name="Start Job7" scheduler_host="localhost" scheduler_port="4444">
        <start_job job="job7" />
      </command>
      <command name="Start Job9" scheduler_host="localhost" scheduler_port="4444">
        <start_job job="job9" />
      </command>
      <command name="Start Jobchain" scheduler_host="localhost" scheduler_port="4444">
        <add_order job_chain="job_chain_sample" replace="yes" id="4711" />
      </command>
    </commands>
  </action>
</actions>
```



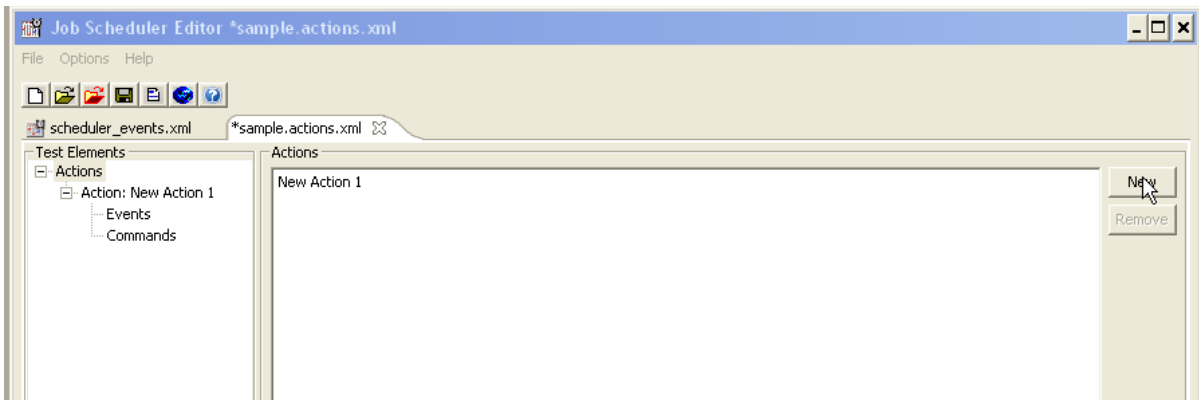
The commands and rule set can be conveniently worked on using the Job Scheduler Job Editor:



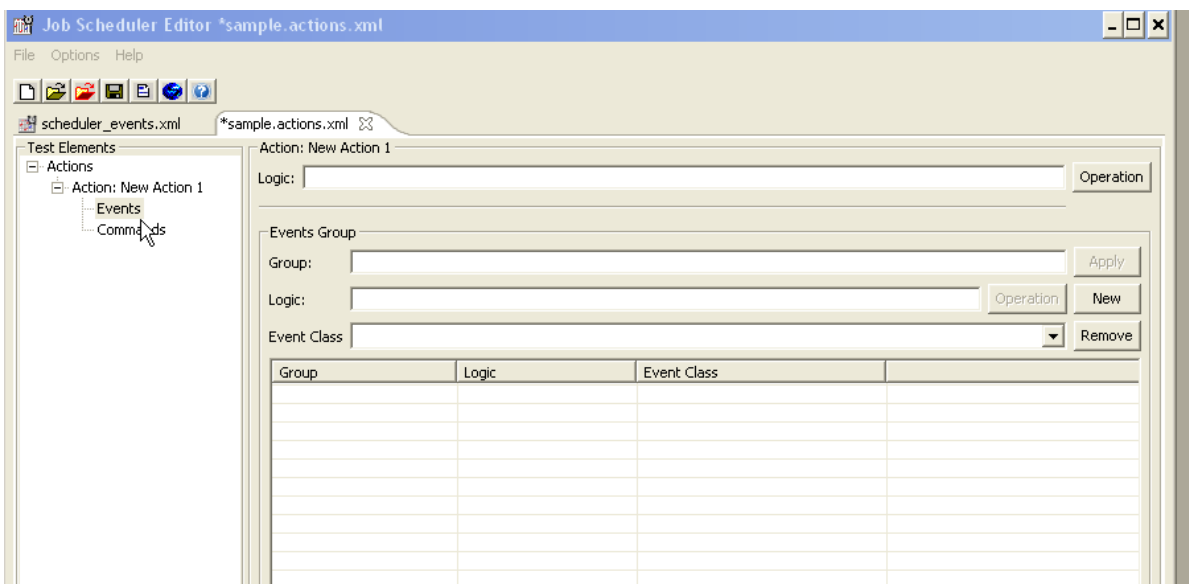
Event handling routines are made up of one or more *actions*. Actions, in turn, are made up from event groups and commands with the event groups being evaluated according to the (event logic) conditions specified. Similarly, groups will be evaluated according to the corresponding group logic conditions. If all groups evaluate to TRUE, then the relevant commands will be executed. If only one group has been specified, then this must evaluate TRUE. If only no condition has been specified, then one of the groups must evaluate TRUE (default *or*).

**Example:** one group and two events. When event 1 or event 2 is present then the commands will be executed.

### Step 1: add a new action



### Step 2: add a new event group



A group has a name and a logic parameter and comprises one or more events. An event class can be specified for all the events of a group. The default logic operator is "OR". The logical operators possible are:

- *or*: one of the events must be present
- *and*: all events must be present
- conditions that combine the results of a group – for example: *event1 and not event2*

Within the logic parameter, events are referenced by their names. If the name of an event is empty, then a name will be generated for it by the Event Processor using `<class>.<id>`.

**Step 3: specify events**

\*sample.actions.xml

Action: New Action 1 Group: Test

Event Title: Job2 ist gelaufen Apply

Event Class: sample New

Event Id: 2

Job Name: jobd2

Job Chain:

Order Id:

Comment:

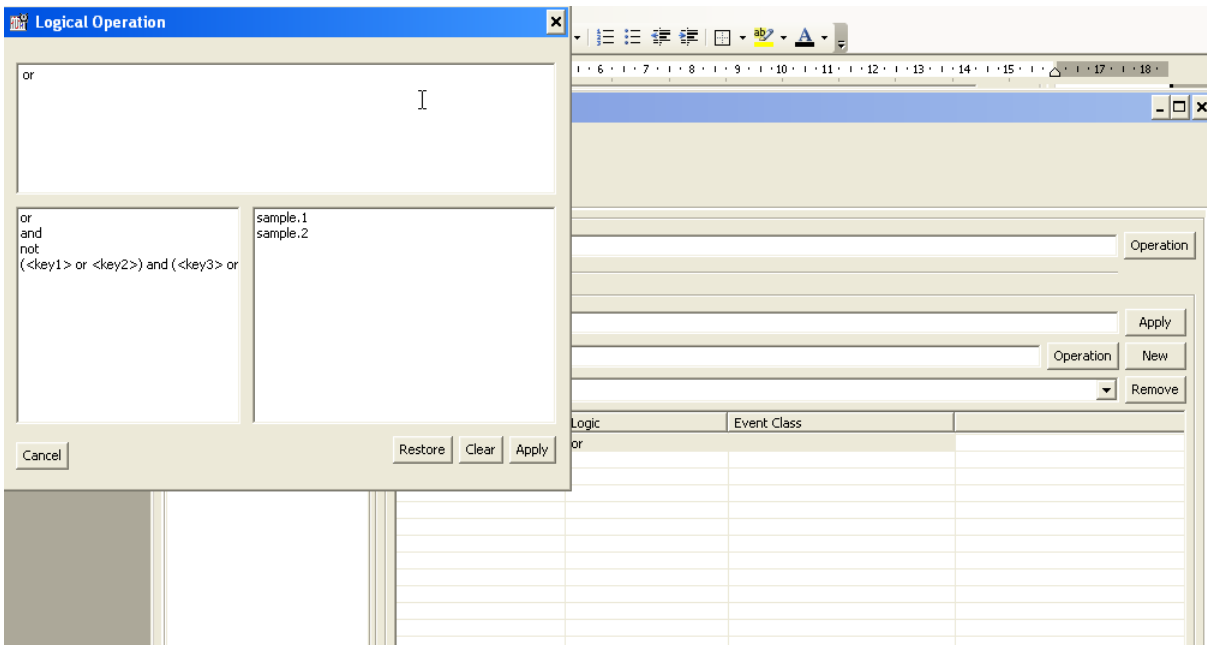
Event Id	Event Title	Event Class	Jobname	Job...	Order Id	Comment	
1	Job1 ist ...	sample	job1				Remove

An event has the following parameters:

- A **title**: is shown in the console event monitor.
- A **name**: is used to address the event in the logic (e.g. `event1` and not `event2`). If only `or` or `and` are specified in the logic, then it is not necessary to specify a name. However, it is recommended that a name is defined as, for example, this will be shown in the console event monitor.
- A **class**: is used to test whether an event is to be found in the list of current events. Can be used if the `name` of an event is empty and an internal name is generated using `class.id`.
- An **ID**: is used to test whether an event is to be found in the list of current events. Can be used if the `name` of an event is empty and an internal name is generated using `class.id`.
- A **job name**: is used to test whether an event is to be found in the list of current events.
- A **job chain name** is used to test whether an event is to be found in the list of current events.
- A **current order identifier** (order id) is used to test whether an event is to be found in the list of current events.

**Step 4: specify logic**

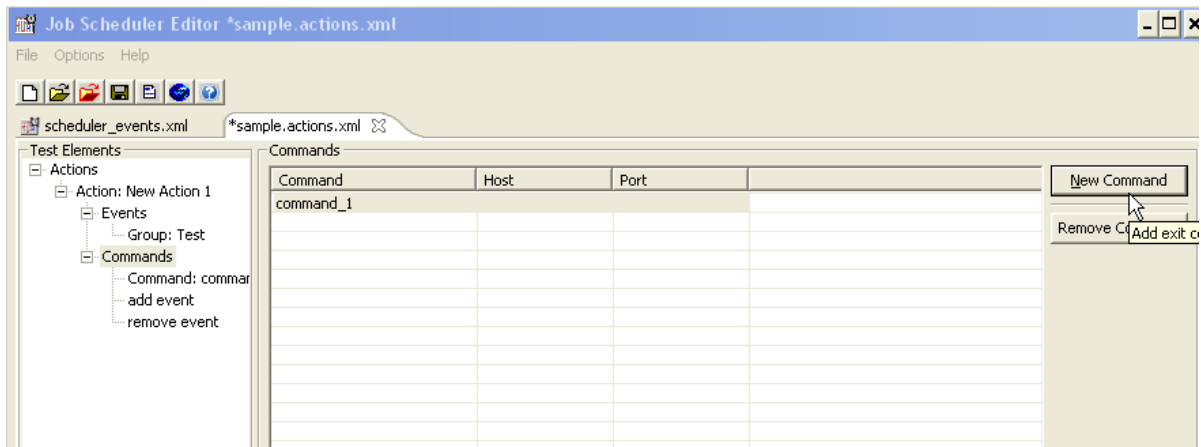
The logic can either be specified using a wizard or directly entered in the input field. In the screen shot below "or" has been selected from the list of possible operators



**Step 5: add any additional groups**

More groups can be included using the Group Logic Field. The default logic operator for groups is "OR".

## Step 6: add commands



A host and a port need to be specified for all commands within a block. The commands are then sent to this Job Scheduler.

Commands are added using "New Command". •The following commands are available:

- **start job**: starts a job. The start time is to be specified.
- **order**: starts a job chain,
  - **job chain**: the job chain for the order
  - **order ID**: a unique ID for the order
  - **start time**: orders can be given delayed starts
  - **priority**: the order priority
  - **title**: free text
  - **status**: the state with which the order should start in the job chain.
  - **end state**: the job chain state which the order should run to.
  - **replace**: should the order overwrite an existing order of the same ID?

Command: command\_1

Job chain:

Order Id:

Start at:  -  -   :  :

Priority:

Title:

State:

End State:

Replace:

- **add event:** creates a new event
- **remove event:** deletes events. For example, all the events belonging to a class or job could be deleted.

The screenshot shows a web application window titled 'sample.actions.xml'. The main area contains a form for defining an event group. The form has the following fields:

- Action: Group: (text input)
- Event Class: (dropdown menu)
- Event Id: (text input)
- Job Name: (text input)
- Job Chain: (text input)
- Order Id: (text input)

Buttons for 'Apply' and 'New' are located to the right of the form. Below the form is a table with the following columns: Event Id, Event Title, Event Class, Jobname, Job..., Order Id, Comment. A 'Remove' button is located to the right of the table.

The event handler is saved as an XML file.

```
<?xml version="1.0" encoding="UTF-8"?>
<actions>
  <action name="New Action 1">
    <events>
      <event_group group="Test" logic="or">
        <event event_id="1"
          event_class="sample"
          event_title="Job1 has been completed"
          job_name="job1" />
        <event event_id="2"
          event_class="sample"
          event_title="Job2 has been completed"
          job_name="jobd2" />
      </event_group>
    </events>
    <commands>
      <command name="command_1" scheduler_host="localhost" scheduler_port="4454">
        <start_job job="job0" at="now" />
        <add_order job_chain="job_chain1" replace="yes" />
      </command>
    </commands>
  </action>
</actions>
```

## Evaluation of the rule set

Evaluation of the rule set tests whether the events specified in the logic of an event group can be found in the list of currently existing events. The TRUE / FALSE values determined are then combined using the group Boolean operators.

An event is seen as being present in the events list once the attributes specified in the event handler have been tested. Note, however, that the *title*, *expires* and *creation* attributes are not considered here.

### Example:

This event group evaluates to TRUE when event1 is present and event2 not present.

- event1 is present, when an event with *event\_class="sample"*, *event\_id="1"* and *job\_name="job1"* is found
- event2 is present, when an event with *event\_id="2"* is found.

```
<event_group group="Test" logic="event1 and not event2">
  <event event_id="1"
    event_name="event1"
    event_class="sample"
    event_title="Job1 has been completed"
    job_name="job1" />
  <event event_id="2"
    event_name="event2"
    event_title="Job2 has been completed"/>
</event_group>
```

## 5 Event Monitoring

Events can be monitored using the Job Scheduler Console, which is available with the commercial Job Scheduler license.

The Console is set up using a configuration file in which the Job Schedulers to be monitored are specified, with consecutively numbered host / port pair being defined for each Job Scheduler being monitored.

To be able to use this plug-in together with the Event Handler, it is necessary to specify the Event Handler directory in the Console plug-in configuration file. This is done in the `plugin_action_show_dialog` section of the file.

The Console is started using the `scheduler/bin/console` command.

A sample configuration:

```
[Console.Supervisor]
host1=kyrill.sos
port1=9000

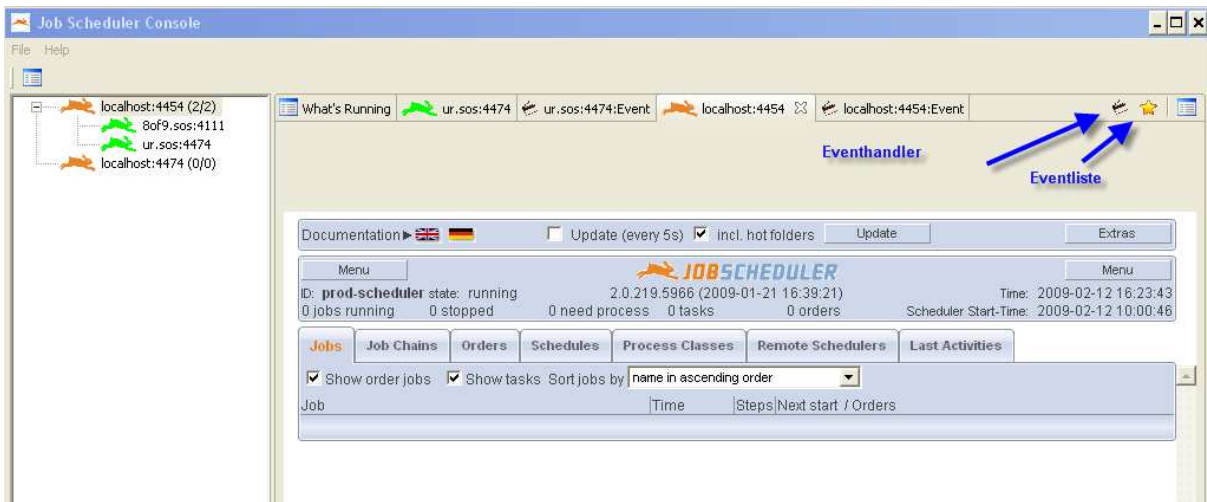
host2=kyrill.sos
port2=9001

host3=kyrill.sos
port3=9002

host4=localhost
port4=4474

host5=localhost
port5=4454


[plugin_action_show_dialog]
configuration_directory=c:/scheduler/config/events
```

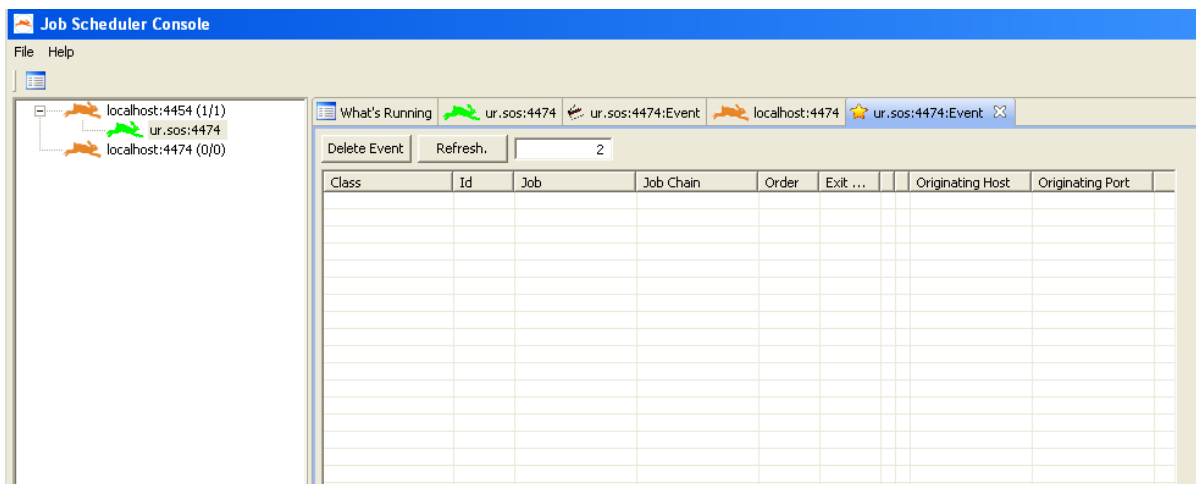


The following views of Job Scheduler events are available in the Console:


- Events list

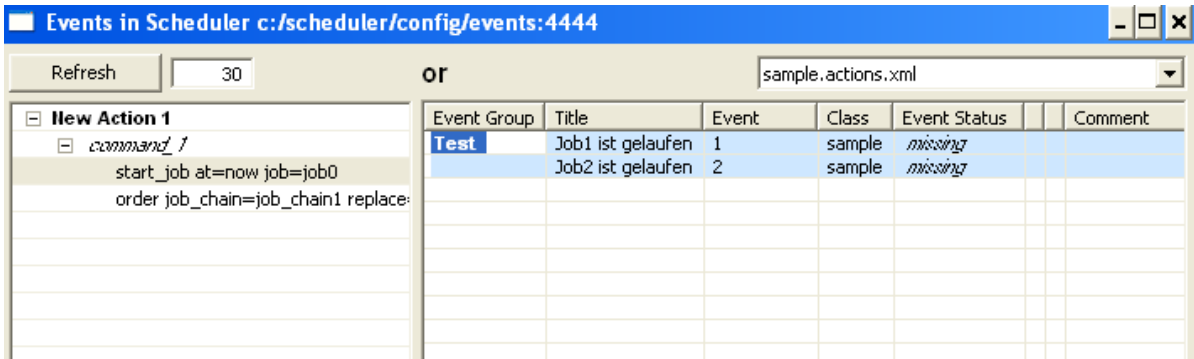
Events present are shown in the events list. The "Delete Event" button (see the screen shot below) is used to delete events that have been marked in the list.

The list of events is shown when the  button is clicked, as shown in the screen shot above.



- Event Handler List with current event status

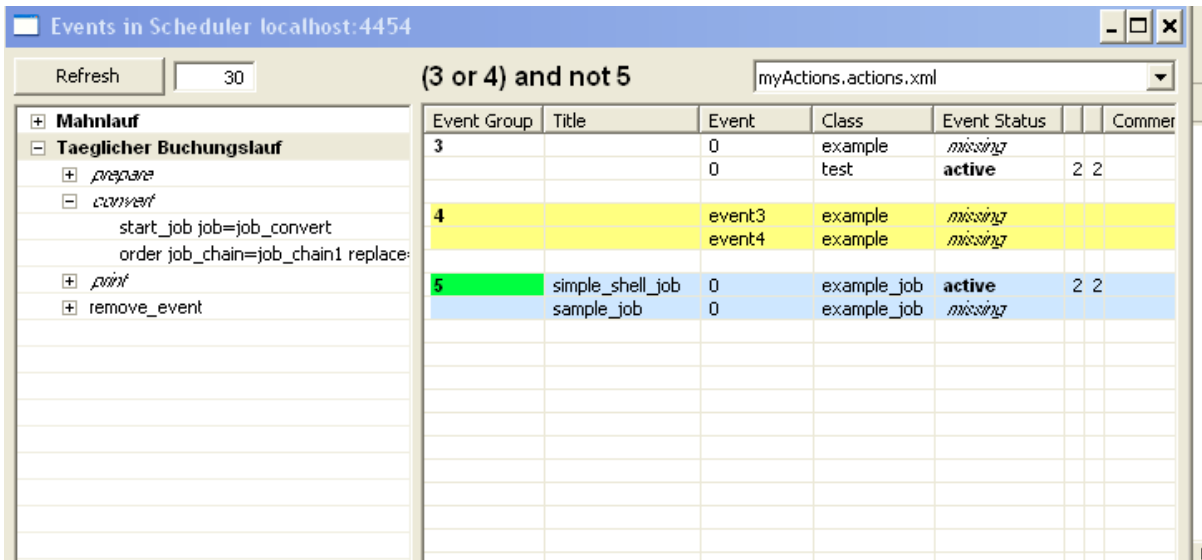
The list of event handlers is shown when the  button, as shown in the screen shot on the previous page, is clicked.



The required event handling routine must first be selected from the list box at the top right of the Console, as shown in the screenshot above, before the routine actions are shown. The commands which the routine should send to the Job Scheduler are listed in the left hand window: the events required, divided into event groups, are shown in the right window. The logic with which events in an event group are linked is shown when the name of the event group is clicked (see the screen shot in the example below).

Events in the right hand window are marked in color as described in the example below.

**Example:**



The yellow background color means:  
 "all the events in the group must be present (logic=AND)".

The light blue background color means:  
 "one of the events in the group must be present (logic=OR)".

The green background color means: "the event is present".

Actions in the right hand window are shown in red when the rule set returns TRUE.

## 6 Event Processing Configuration

### 1 Event Processor Installation

The Event Processor must be installed on a Supervisor Job Scheduler.

The `scheduler_event_service` file can be found in the `config` directory of the Job Scheduler installation and defines `scheduler_event_service` job chain. This job chain implements the Event Processor.

The configuration should be defined as a `<base file="..." />` so that it is made known to the Supervisor Job Scheduler.

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<config>
  <!-- -->
  <base file="scheduler_events.xml" />
  <process_classes>
    <process_class max_processes="10" />
  </process_classes>
</config>
```

The file can be downloaded from the [http://www.sos-berlin.com/download/scheduler/samples/events\\_supervisor.zip](http://www.sos-berlin.com/download/scheduler/samples/events_supervisor.zip)

link.

#### Configuration:

The `"expiration_period"` parameter should be set to the standard expiry time required for all events: e.g.

```
<param name="expiration_period" value="00:00" />
```

### 2 Define at least one event handler

Event handlers are written and modified using the Job Scheduler Job Editor. The default directory for saving Event Handlers is `scheduler/config/events`. Note that this directory can be changed by modifying the

```
<param name="event_handler_filespec" value="scheduler_events.xml" />
```

parameter in the Event Processor.

The following example illustrates a very simple Event Handler, in which Job B starts after Job A has been completed:

```
<?xml version="1.0" encoding="UTF-8"?>
<actions>
  <action name="sample">
    <events>
      <event_group group="Main">
        <event event_id="0" event_title="Job A" job_name="job_a" />
      </event_group>
    </events>
    <commands>
      <command name="start_Job_B" scheduler_host="localhost" scheduler_port="4444">
        <start_job job="job_b" />
      </command>
    </commands>
  </action>
</actions>
```

### 3 Configure an event generator for every event listed in the event handler

There are several possible methods to generate a "Job A is running" event:

- if it is a shell job then the `jobscheduler_event.cmd` shell script can be called;
- if it is a Java job then the monitor can be used.

Example showing the use of a shell script:

```
<job>
  <script language="shell">
    <![CDATA[
      # any commands
      # submit event to Supervisor Job Scheduler
      ${SCHEDULER_HOME}/bin/jobscheduler_event.sh -x $? -e "sample"
    ]]>
  </script>
  <run_time/>
</job>
```

## 7 Event Handling Routine Examples

### 7.1 Split and Merge

This example consists of two actions: the "split" action, which is followed by the "merge" action.

The "split" action starts *job\_B\_1* and *job\_B\_2* once *job\_A* has been completed. In addition, the event will be deleted after it has been processed.

The "merge" action waits on the *job\_B\_1 completed* and *job\_B\_2 completed* events as well as the conditions for *job\_B\_4*, which is manually started. The action then starts *job\_C* and removes the events involved.

```
<actions>
  <action name="Split">
    <events>
      <event_group group="A">
        <event event_class="example_split" event_id="0"
          event_title="Job A" job_name="job_A"/>
      </event_group>
    </events>

    <commands>
      <command name="Start B1" scheduler_host="localhost" scheduler_port="4474">
        <start_job job="samples/splitAndMerge/job_B_1"/>
      </command>

      <command name="Start B2" scheduler_host="localhost" scheduler_port="4474">
        <start_job job="samples/splitAndMerge/job_B_2"/>
      </command>
      <remove_event debug="example_split">
        <event event_class="example_split" event_id="0" job_name="job_A"/>
      </remove_event>
    </commands>
  </action>

  <action name="And Merge">
    <events>
      <event_group logic="and" group="A">
        <event event_class="example_andmerge" event_id="0"
          event_title="Job B_1" job_name="job_B_1"/>
        <event event_class="example_andmerge" event_id="0"
          event_title="Job B_2" job_name="job_B_2"
          comment="Call 01728388383 if not active before 18:00"/>
        <event event_class="example_andmerge" event_id="0" event_title="Job B_4"
          job_name="job_B_4" comment="Is started manually"/>
      </event_group>
    </events>

    <commands>
      <command name="Start C" scheduler_host="localhost" scheduler_port="4474">
        <start_job job="samples/splitAndMerge/job_C"/>
      </command>
      <remove_event debug="example_andmerge">
        <event event_class="example_andmerge" event_id="0"/>
      </remove_event>
    </commands>
  </action>
</actions>
```

## 7.2 Job A and Job B on Job Scheduler 1, then Job C on Job Scheduler 2

*Job A* and *job B* run on Job Scheduler 1 and each generates an event. Once both events are present, *Job\_C* is started on Job Scheduler 2.

```
<actions>
  <action name="New Action 1">
    <events>
      <event_group group="Main">
        <event event_id="1" event_class="sample" event_title="Job A" job_name="job_A" />
        <event event_id="2" event_class="sample" event_title="Job B" job_name="job_B" />
      </event_group>
    </events>
    <commands>
      <command name="command_1" scheduler_host="Scheduler2" scheduler_port="4444">
        <start_job job="job_C" />
      </command>
    </commands>
  </action>
</actions>
```

## 7.3 Job A has not run by 5 o'clock

In this example, the situation where *job\_A* monitors a directory is considered: The function of *job\_1700* is exclusively that of running at 17:00 and sending an event.

*Job\_B* is started, when *job\_1700* has run and *job\_A* has not been completed:

```
<actions>
  <action name="Missing File">
    <events logic="A">
      <event_group logic="example_missing.time and Not example_missing.file" group="A">
        <event event_class="example_missing" event_id="file"
          event_title="File is detected" job_name="job_A" />
        <event event_class="example_missing" event_id="time"
          event_title="Its Teatime now" job_name="job_1700" />
      </event_group>
    </events>
    <commands>
      <command name="Start B1" scheduler_host="localhost" scheduler_port="4474">
        <start_job job="job_B1" />
      </command>
      <remove_event>
        <event event_class="example_missing" event_id="0" />
      </remove_event>
    </commands>
  </action>
</actions>
```

## 7.4 Notification when a file has not arrived before 17:00

### Generate the "It is 17:00" event

```
<?xml version="1.0" encoding="UTF-8859"?>
<job name="sample" title="Submit &quot;17:00&quot; Event;">
  <description>
    <include file="jobs/JobSchedulerSubmitEventJob.xml"/>
  </description>
  <params>
    <param name="scheduler_event_class" value="file_arrived"/>
    <param name="scheduler_event_id" value="1"/>
  </params>
  <script language="java"
    java_class="sos.scheduler.job.JobSchedulerSubmitEventJob"/>
  <run_time>
    <period single_start="17:00"/>
  </run_time>
</job>
```

### Generate the "File has arrived" event

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<job name="sample" title="Submit Event &quot;File has arrived&quot;">
  <description>
    <include file="jobs/JobSchedulerSubmitEventJob.xml"/>
  </description>
  <params>
    <param name="scheduler_event_class" value="file_arrived"/>
    <param name="scheduler_event_id" value="2"/>
  </params>
  <script language="java"
    java_class="sos.scheduler.job.JobSchedulerSubmitEventJob"/>
</job>
```

### Event handler

```
<?xml version="1.0" encoding="UTF-8"?>
<actions>
  <action name="FileIsMissing">
    <events>
      <event_group group="File Missing" logic="Its17_00 and not File"
        event_class="file_arrived">
        <event event_name="File" event_class="file_arrived"
          event_title="File has arrived" event_id="2" />
        <event event_name="Its17_00"
          event_title="A specific time has been reached" event_id="1" />
      </event_group>
    </events>
    <commands>
      <command name="notify" scheduler_host="localhost" scheduler_port="4444">
        <start_job job="job_notify">
          <params>
            <param name="to" value="admin@host.de" />
            <param name="subject" value="File is missing" />
          </params>
        </start_job>
      </command>
      <remove_event>
        <event event_class="file_arrived" />
      </remove_event>
    </commands>
  </action>
</actions>
```

## 8 Appendix

### Event Processor configuration

The job chain contains the *scheduler\_event* job. The parameterization of this job is described in the *scheduler/jobs/JobSchedulerEventJob.xml* documentation.

```
<config>
  <jobs>

    <job name="scheduler_event_service"
        title="Process Events"
        order="yes"
        stop_on_error="no"
        timeout="120">
      <description>
        <include file="jobs/JobSchedulerEventJob.xml"/>
      </description>

      <params>
        <param name="event_handler_filepath"
            value="./config/events"/>
        <param name="event_handler_filespec"
            value="scheduler_events.xsl"/>
        <param name="expiration_period"
            value="12:00"/>
      </params>

      <script java_class="sos.scheduler.job.JobSchedulerEventJob"
            language="java"/>

      <run_time/>
    </job>
  </jobs>
  <job_chains>
    <job_chain name="scheduler_event_service"
        orders_recoverable="no"
        visible="yes">
      <job_chain_node state="start"
          job="scheduler_event_service"
          next_state="end"
          error_state="error"/>

      <job_chain_node state="end"/>

      <job_chain_node state="error"/>
    </job_chain>
  </job_chains>
  <commands>
    <add_order id="event_processor"
        job_chain="scheduler_event_service">
      <params>
        <param name="action"
            value="process"/>
        <param name="event_handler_filespec"
            value=".xsl$"/>
      </params>

      <run_time let_run="yes"
          repeat="300"/>
    </add_order>
  </commands>
</config>
```