ICALEPCS'13

14th International Conference on Accelerator & Large Experimental Physics Control Systems
October 6-11, 2013, San Francisco, California

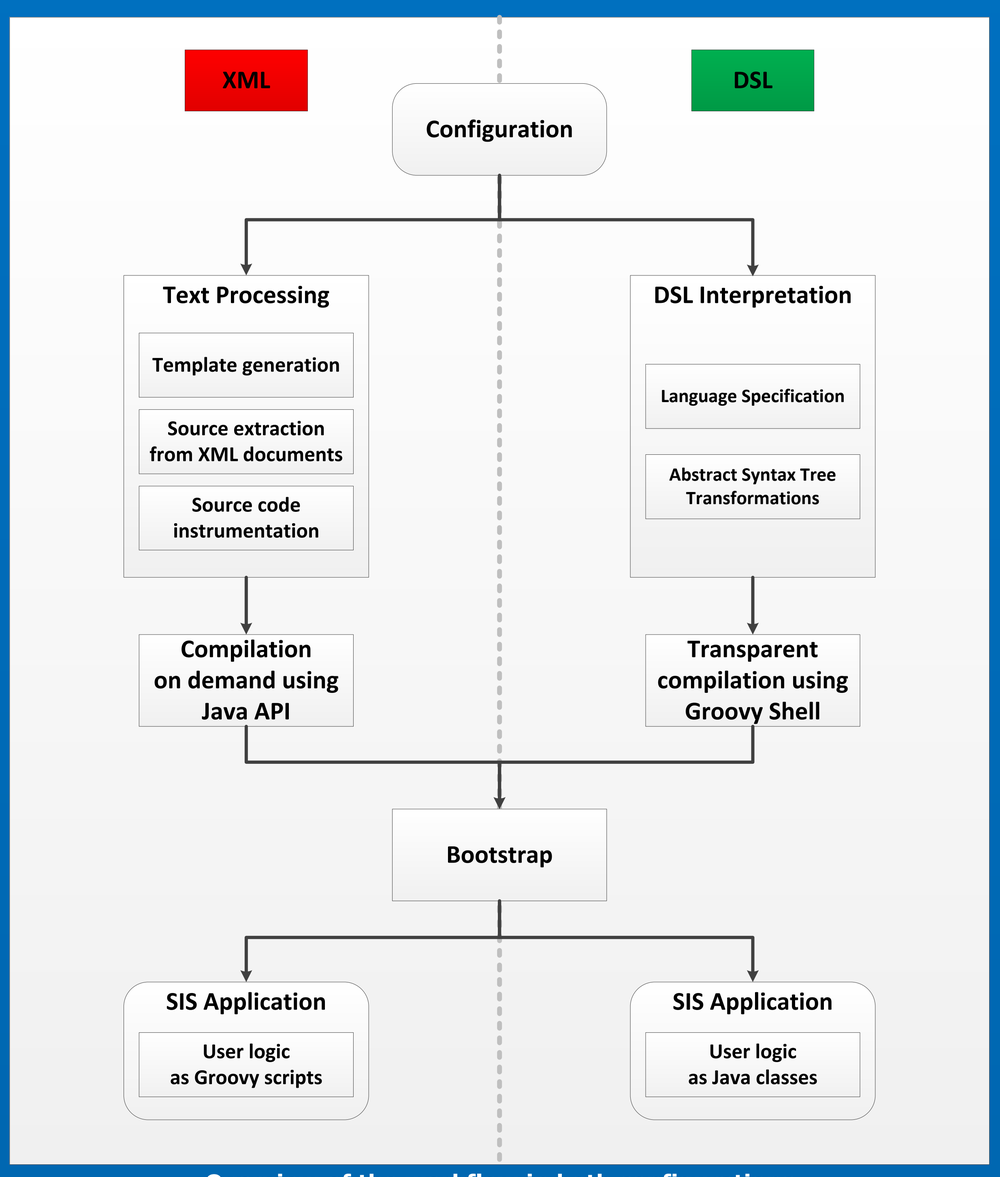
GROOVY AS DOMAIN-SPECIFIC LANGUAGE IN THE SOFTWARE INTERLOCK SYSTEM

J.Wozniak, M. Polnik, G.Kruk CERN, Geneva, Switzerland

Abstract

After 7 years in operation the Software Interlock System (SIS) has become an indispensable and mission-critical controls tool covering many operational areas from general machine protection to diagnostics. The growing number of running instances as much as the size of existing configurations have increased both the complexity and maintenance cost of running the SIS infrastructure. In response to those issues, a new ways of configuring the system have been investigated aiming at simplifying the configuration process by making it faster, more user friendly and understandable for wider audiences and domain experts alike. As one of the possible choices the Groovy scripting language has been considered as being particularly well suited for writing a custom Domain-Specific Language (DSL) due to its built-in language features like native syntax constructs, command chain expressions, hierarchical structures with builders, closures or Abstract Syntax Tree (AST) transformations. This document explains best practices and lessons learned while introducing an accelerator physics domain oriented DSL language for the configuration of the Software Interlock System developed by the Data & Application Section at CERN.

Configuration Process Comparison

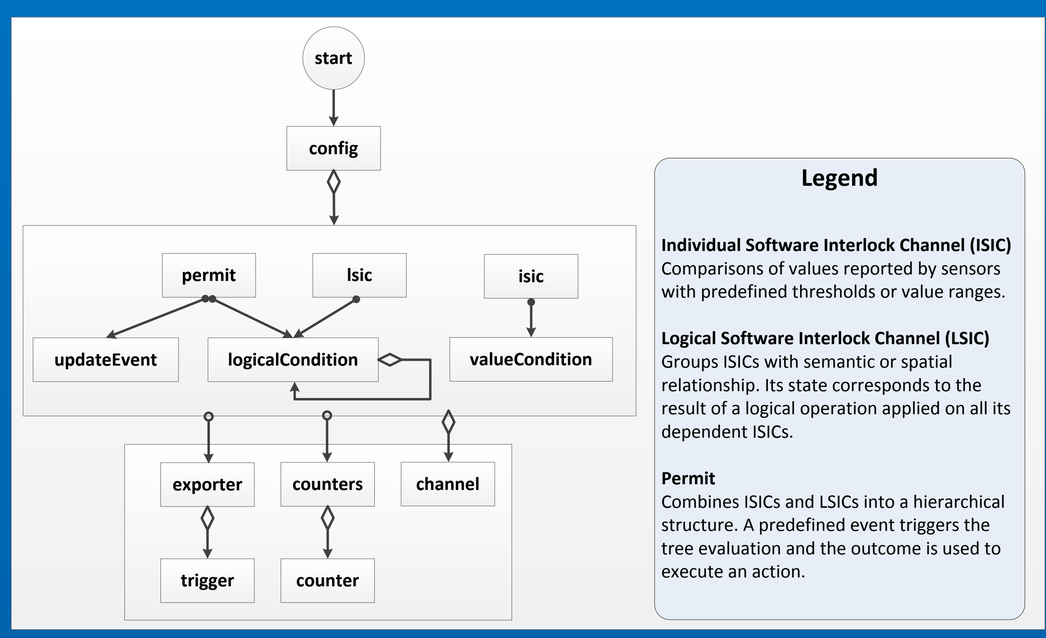


Overview of the workflow in both configurations

Being originally designed for describing documents, XML is not well suited to accommodate condition logic. Its definition, although possible in XML, makes the document verbose and difficult to read. Moreover, requirements for typical programming language constructs like include directives, variables and control statements have also become desired features in SIS. According to these needs a Velocity template language was introduced to pre-process and generate the final XML files. Overall the typical configuration as a mixture of Velocity statements, XML tags, Groovy scripts and references to Java classes was neither maintainable in the long term nor straightforward to handle.

The aforementioned issues in SIS framework were mitigated after adopting the DSL approach for defining constraints that must be satisfied for running systems safely. There are no restrictions concerning the programming logic used in conditions. They can perform computations, access system components or check the status of devices using features implemented in the DSL making the process of providing user defined logic more flexible and robust. Moreover, the workflow of the configuration handling startup has been simplified as well. Groovy Shell transparently compiles the configuration to a single script class and to a set of inner classes containing user defined conditions. Assuming there were no compilation errors, the script is executed to create a tree-like object representation of the configuration. This intermediate structure will be used to instantiate and wire up the application components.

Domain Model



SIS DSL Language Model

The SIS system helps protecting the machine by surveying the state of the devices. It continuously evaluates user-defined conditions and dumps or inhibits the beam production if an abnormal situation is detected. The SIS is designed to protect the machines against repetitive faulty conditions limiting radiation, extending the equipment lifetime and making the machine diagnostics much easier.

DSL Language Features

XML

```
#set($virtualDev =["BTY.VSISQDE209",
  "BTY.VSISQFO210","BTY.VSISDHZ211"])
#set($hardwareDev = ["BTY.QDE209",
  "BTY.QFO210","BTY.DHZ211"])
#macro( isoChannel $name $virtualParam )
<Isic id="$name">
  <ValueCondition param="$${name}" operator="<"
  refValue="100"/>
  <Exporter beanId="timingExporter">
     <Trigger event="SKIP_IF_MASKED"/>
  </Exporter>
</Isic>
#end
#foreach($device in $hardwareDev)
  #set( $virtualParam = $virtualDev
  [$foreach.index])
  #isoChannel( $device $virtualParam $device )
#end
<Permit id="ISO_GPS_PERMIT">
  <LogicalCondition operator="AND">
     #foreach($device in $hardwareDev)
          <Test refid="$device"/>
     #end
  </LogicalCondition>
  <Exporter beanId="timingExporter">
     <Trigger event="ON_EVAL"/>
  </Exporter>
  <UpdateEvent>
     <![CDATA[
          return isTriggerId("tgmTelegram")
     ]]>
  </UpdateEvent>
</Permit>
```

DSL

```
def virtualDev = ["BTY.VSISQDE209",
  "BTY.VSISQFO210","BTY.VSISDHZ211"]
def hardwareDev = ["BTY.QDE209",
  "BTY.QFO210","BTY.DHZ211"]
def isic = {String name, String virtualParam ->
    isic(id:name) {
        valueCondition {
            return $(name) < 100
        }
        exporter(beanId:"timingExporter") {
            trigger(event:"SKIP_IF_MASKED")
        }
    }
}
for(int i=0; i < virtualDev.size(); ++i) {
    isic(virtualDev[i],hardwareDev[i])
}
permit(id:"ISO_GPS_PERMIT") {
  logicalCondition {
      return channel(virtualDev[0]) &
  channel(virtualDev[1]) &
  channel(virtualDev[2])
    }
    exporter(beanId:"timingExporter") {
       trigger(event:"ON_EVAL")
    }
  updateEvent {
       return
  "tgmTelegram".equals(it.getTriggerId())
  }
}
```

Example configuration in XML and DSL

The main advantages in the new configuration approach leveraging a DSL are significant improvements in terms of productivity and support for domain-oriented features:

- IDE assistance
- User-defined conditions
- Self-documenting configurations
- Service locator
- Access to device sensors

CONCLUSIONS

Taking the DSL approach for the SIS configuration proved itself to be the right choice in practice. Its interoperability with Java on the binary level is a great advantage opening ways for the implementation of the DSL in a mixed Java & Groovy mode. Also its build-in features targeting directly the DSL construction make the design of such language much easier. The corresponding files are much smaller and more readable comparing to their XML counterparts. At the same time the configuration is more concise with all its entities represented as Groovy code constructs. Overall it improves significantly the level of user satisfaction and maintainability of the system as a whole.