GROOVY AS DOMAIN-SPECIFIC LANGUAGE IN THE SOFTWARE INTERLOCK SYSTEM

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

Abstract

After over 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, 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.

THE SIS PROJECT

The Software Interlock System (SIS) is part of the overall Machine Protection systems' group of applications. It helps protect the machine by surveying the state of a set of devices. At each evaluation, it performs a number of checks (conditions) and dumps or inhibits the beam production if an abnormal situation is discovered. The basic checks, called Individual Software Interlock Channels (ISICs), usually compare the reading from a device with a predefined threshold or range of values (like *temperature < 50 deg. Celsius*). Several ISICs with some logical or geographical relationship can be grouped into a so-called Logical Software Interlock Channels (LSICs). The state of an LSIC corresponds to the result of a logical operation applied on the state of all of its dependent ISICs. The logical operation may be any combination of AND, OR and NOT operators. The state of each LSIC is either TRUE or FALSE. All the ISICs and LSICs together with a root node, called Permit, form the structure of a logical tree. The evaluation of the tree is fired by a predefined periodic event, and the outcome is used to act on external systems. SIS was designed to protect the machines against repetitive faulty conditions thus limiting damage caused by radiation or other harmful states, thus extending the equipment lifetime and making the machine diagnostics much easier. The interested reader can refer to [1] for more information about the SIS itself.

DOMAIN-SPECIFIC LANGUAGES

A DSL is a type of computer language or specification language used in a domain (banking, physics, medicine, controls, etc.) to solve or describe a particular domain problem or area of interest. Using DSLs instead of general-purpose languages allows a particular type of problem or solution to be expressed in terms closer to the language used by the end users of the system that are not necessarily computer scientists or programmers [2].

MOTIVATION

The growing number of SIS instances as well as the complexity of each configuration has induced an increased cost of maintenance. The original choice of XML for the configuration of SIS had to be extended by other techniques in order to overcome its limitations. Being originally designed as a document description language, XML is not well suited to accommodate conditional logic. The definition of a Boolean condition, although still possible in XML, makes the document verbose and difficult to read. For that reason SIS allows to define complex conditions as Java classes or Groovy scripts. Typical language constructs like file includes, variables, loops or if/else statements were also allowed in the SIS configuration files, as the configuration usually consists of a repetitive number of similar conditions multiplied by the number of devices of a given class (like power-converters). These features were implemented using the Velocity template language that was used to pre-process and generate the final XML files. In summary, the current typical configuration of a SIS instance is a mixture of Velocity statements, XML tags, Groovy scripts and references to Java classes. This mixture of languages makes it difficult to read, understand and maintain the configuration of an SIS instance. Furthermore, the configuration file is a static entity that gets processed in runtime postponing error detection to the very last moment, at system startup. A comparable DSL [4] solution could possibly improve this situation by using a compiled language with IDE facilities like support for syntax highlighting and code completion. A well-designed DSL could unify all the previously mentioned requirements within a unique language having

ISBN 978-3-95450-139-7

both the domain specific syntax and the host language syntax available at the same time. A number of different possible options have been taken into account like external versus internal DSLs [2], and Groovy versus Scala languages [6, 7]. Finally we selected Groovy for its syntax and binary compatibility with Java, its relatively shallow learning curve, its built-in DSL features and previous experience with the language.

IMPLEMENTATION

The implementation of DSL solutions does not differ from well-established practices in computer languages processing. It can be implemented using an interpreter or applying a multi-phase compilation process. The first, simpler approach is sufficient in most applications that leverage a DSL for configuration. However, some scenarios may require executing the user-defined logic at the speed that only a compiled language can provide. These separate goals of interpretation and compilation are reflected in the market. Although Java can execute its source code provided at runtime using the Java 7.0 Compiler API, it is not the right tool. Dynamic languages were designed to allow for the extension and modification of a program in run time. Groovy is a dynamic language that provides a set of built-in solutions for defining DSLs, and allows the programmer to build systems that combine the efficiency of precompiled code with the flexibility of interpreted code in one, unified tool.

GROOVY FEATURES

Groovy supports seamless integration with Java. A Groovy source code file can reference or extend Java classes. It is even possible to cross-reference both Java and Groovy code. This is particularly important in applications that require high level of customization by allowing runtime configuration, and running client-side logic. Successful examples are software build systems or development frameworks [3, 5].

In terms of usability features, syntax highlighting and IDE assistance are two of the most notable features for replacing XML with Groovy. Additionally, Groovy provides a built-in framework for defining internal DSLs and the ability to customize the compilation process. Embedded DSLs in Groovy are defined in a similar way to controllers in web application frameworks. The language designer is responsible for providing a set of keywords associated with methods that should be called when the token is encountered. The names of the keywords and methods do not have to correspond to each other. For transforming the logic expressed in textual form into its object code representation, Groovy provides the builder entity. It processes the input file line by line and executes predefined actions associated with a particular token. Apart from the semantics specified by the DSL keywords, files may contain Groovy source code blocks known as Closures [6]. This feature can be leveraged to create highly customizable applications open for user-defined logic. Closures are extensively used for defining update events and conditions in the SIS framework. The expressiveness of the DSL language is not solely determined by the size of the supported keyword set, which depends on the domain complexity, but also on the flexibility of its operators. The Java specification does not support operator overloading which forces developers to use methods instead. Operator overriding is allowed in Groovy, thus improving the readability of DSL code by reducing the overhead of the method calling syntax. On the other hand, the total number of operators is limited to the ones defined by the language.

The SIS DSL provides easy access to application components and the status of the accelerator devices. Some applications may require functionalities concerning validation checks or code instrumentation that go beyond the scope of the language grammar. Defining additional processing logic weaved during the compilation phase may cover such advanced scenarios. For this purpose, the Groovy compiler supports Abstract Syntax Tree (AST) transformations. In particular, the SIS project intercepts the semantic analysis stage of the compilation process to associate compiled scripts containing user-defined logic with their original source code for future reference in the presentation layer. It allows the users to consult the script content of the conditions in the graphical interface of the system.

DSL WORKFLOW IN SIS

The SIS framework provides a DSL solution for defining the safety constraints governing a system, and for wiring up the application components. The configuration workflow is as follows. The users of the SIS framework define the configuration of the system in a set of files written in both DSL and Groovy. The configuration files describe the connections between application components and the conditions that must be satisfied for running the system safely. There are no restrictions concerning the programming logic used in the conditions: they can perform computations, access system components or check the status of devices using features provided in DSL. When the application starts, the configuration is compiled to a single script class and individual conditions are compiled to separate inner classes. Assuming there were no errors in the compilation process, the script is executed to create a tree-like object representation of the configuration. This intermediate structure will be used to instantiate, configure and wire the components of the SIS application.

EXAMPLE

The following two snippets present the configuration of the same, simple application written in the SIS framework. The protected system is considered to be safe if all the values observed by its sensors are below fixed thresholds. When this condition is broken, an action implemented by the provided exporter is executed.

ISBN 978-3-95450-139-7

Configuration in XML

```
#set($virtualDev =["BZZ.VSISDEV1",
    "BZZ.VSISDEV2","BZZ.VSISDEV3"])
#set($hardwareDev = ["BZZ.DEV1",
    "BZZ.DEV2","BZZ.DEV3"])
#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>
```

Counterpart in DSL

```
def virtualDev = ["BZZ.VSISDEV1",
    "BZZ.VSISDEV2","BZZ.VSISDEV3"]
def hardwareDev = ["BZZ.DEV1,
    "BZZ.DEV2","BZZ.DEV3"]
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())
    }
}
```

The first configuration is written in XML, interwoven with Velocity directives, which reduce the total number of lines of code at the expense of its readability. This approach had to parse the XML document to extract the user-defined logic. On the other hand, the equivalent DSL configuration is more compact due to the conciseness of the Groovy language, and is less error-prone because of the IDE support. The Groovy shell directly interprets the user-defined logic.

CONCLUSIONS

Adopting Groovy as a host language came with its good and bad points. The main disadvantage lies in the existing but still poor IDE support where the tools have not reached the maturity state yet. However, we hope for the situation to improve over time. Another important point is a possible lack of type safety as Groovy is used in a scripting, interpreted mode. Despite those minor flaws it stays a valid technical choice. Its interoperability with Java on the binary level is a great advantage opening ways for the implementation of the DSL in a mixed Java and Groovy mode. Also the previously mentioned built-in features targeting directly the DSL construction make the design of such language much easier.

Taking the DSL approach for the SIS configuration proved itself to be the right choice in practice. 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.

REFERENCES

[1] J. Wozniak, V. Baggiolini, D. Garcia Quintas, J. Wenninger, “Software Interlocks System”, Icalepcs 09’.

[2] M. Fowler, R. Pearsons, “Domain-Specific Languages”, Addison-Wesley 10’.

[3] B. Muscho, “Gradle in Action”, Manning Publications 13’.

[4] M. Mernik, J. Heering, and A. M. Sloane, "When and how to develop domain-specific languages." ACM computing surveys (CSUR) 05’.

[5] M. Seifeddine, “Introduction to Groovy and Grails”. Department of Computer Science, Lund University, 09’.

[6] F. Dearle,“Groovy for Domain-Specific Languages”, Packt Publishing, 10’.

[7] M. Bannet,R. Borgen, K. Havelund, M. Ingham and D. Wagner, “Prototyping a Domain-Specific Language for Monitor and Control Systems”, Journal of Aerospace, Computing, Information and Communication, vol. 7, 10’.

ISBN 978-3-95450-139-7