Domain-Specific Languages with Scala*

Cyrille Artho[1], Klaus Havelund[2], Rahul Kumar[2], and Yoriyuki Yamagata[1]

[1] AIST, Amagasaki, Japan
[2] Jet Propulsion Laboratory, California Institute of Technology, California, USA

Abstract. Domain-Specific Languages (DSLs) are often classified into external and internal DSLs. An external DSL is a stand-alone language with its own parser. An internal DSL is an extension of an existing programming language, the host language, offering the user of the DSL domain-specific constructs as well as the constructs of the host language, thus providing a richer language than the DSL itself. In this paper we report on experiences implementing external as well as internal formal modeling DSLs with the Scala programming language, known in particular for its support for defining DSLs. The modeling languages include monitoring logics, a testing language, and a general purpose SysML inspired modeling language. We present a systematic overview of advantages and disadvantages of each option.

Keywords: External and internal Domain-specific language, DSL, Scala, modeling, programming, language design, evaluation.

1 Introduction

A domain-specific language (DSL) is a language specialized to a particular domain [10]. DSLs are for example popular in the formal methods and testing communities. A DSL is designed to make the modeling or programming task easier and sometimes eliminates the need for a full-fledged programming language altogether. DSLs are classified into external and internal DSLs. An *external DSL* is a stand-alone language with a customized parser. Examples are XML [7] for representing data and DOT [12] for drawing graphs, for example state machines. In contrast to this, an *internal DSL* extends an existing programming language, the *host language.* A user may employ the host language in addition to the DSL. On the implementation side, the compiler and run-time environment of the host language are reused. Internal DSLs are furthermore divided into shallow and deep embeddings. In a *shallow embedding,* the host language's features are used directly to model the DSL. The constructs have their usual meaning. Conversely, in a *deep embedding,* a separate internal representation is made of the DSL (an

* Part of the work was supported by the Japanese Society for the Promotion of Science (*kaken-hi* grants 23240003 and 26280019), and by NSF Grant CCF-0926190. Part of the work was carried out at Jet Propulsion Laboratory, California Institute of Technology, under a contract with the National Aeronautics and Space Administration.

abstract syntax), which is then interpreted or compiled as in the case of an external DSL. The reuse of an existing programming language reduces development time but can also manifest itself as a constraint for an internal DSL.

Both external and internal DSLs therefore have their own trade-offs. In this paper, we systematically examine these advantages and disadvantages, based on our own practical experience in building numerous external as well as internal modeling and testing DSLs with the Scala programming language. Scala is a modern strongly typed programming language combining object-oriented and functional programming [22]. Scala has several libraries and features that make it suited for DSL development. For external DSLs these include parser combinators, and for internal DSLs these include implicit function definitions, which allow values of one type to be lifted to values of a different type, the ability to call methods on objects without dots and parentheses, case classes, partial functions, call-by-name, user-defined operators composed of symbols, operator overloading, as well as other features such as higher-order functions and lambda expressions.

Our contributions are: (a) a report on our experiences on using Scala to implement external and internal DSLs for formal modeling, three of which have been incorporated into real production systems, (b) a summary of the pros and cons of each type of DSL, and (c) a survey of the various tools and techniques used for creating DSLs with Scala. A higher-level contribution is the message that Scala, as a high-level programming language, can be used for modeling in general, possibly augmented with internal modeling DSLs.

Much work has been carried out studying the development of DSLs [21,10]. The literature is too vast to be surveyed here. Other programming languages have been popular as a platform for DSLs, such as Ruby, Python, and Haskell. One of our through-going themes is DSLs for event monitoring. Such internal DSLs have also been developed in Haskell [29] and Java [6]. Focusing specifically on Scala, an internal DSL for rule-based programming, which we will also present, is presented in [11]. Numerous internal DSLs have been developed in Scala for testing, including for example ScalaTest [30]. Technologies that we have not yet explored, and therefore will not evaluate, include Scala macros [28], Scala virtualized [25], and SugarScala [19], a framework for extending Scala's syntax.

The paper is organized as follows: Section 2 covers the design and implementation of external DSLs, while Section 3 covers internal DSLs. Section 4 compares the key points of different approaches, and Section 5 concludes.

2 External DSLs

An external DSL is a stand-alone language. With Scala, one can develop an external DSL using either a *parser library* or a *parser tool.*

2.1 The parser library approach

Existing parser libraries for Scala include *parser combinators* [24,13] and *parsing expression grammars* [23]. A parser combinator is a higher-order function that

accepts zero or more parsers and returns a new parser. A parser is a function that accepts a string and returns a user-defined data object, typically a parse tree. Parser combinators are based on recursive descent and facilitate modular construction.[3] Generalized LL (GLL) combinators [13] are an alternative to Scala's original parser combinators. The GLL algorithm allows left-recursion and ambiguity, and also provides more elegant handling of semantic actions. Note that the parser combinator libraries in Scala themselves are internal DSLs for writing grammars. A yet different library is Parboiled2, a high-performance Parsing Expression Grammar library [23]. It uses macros to generate a parser at compile time from a grammar written in an internal Scala DSL.

Example: Data automata *Data Automata* (Daut) [15] monitor program executions at run time; execution traces are checked against specifications in Daut's logic. A trace is a sequence of events, each of which is a named tuple of values. The logic is defined using Scala's original parser combinators [24]. Assume as an example that a multi-threaded program has been instrumented to emit events reflecting the acquisition and release of locks: $acquire(t, l)$ and $release(t, l)$ represent events where thread t acquires or releases lock l, respectively. A simple way of detecting deadlock potentials between two threads is to monitor whether threads take locks in a circular manner: one thread acquiring some lock l_1 and then some lock l_2 without having released l_1, and another thread acquiring these two locks in the opposite order. Even if a deadlock does not occur, the circular pattern represents a potential for it to occur. The monitor shown in Figure 1, formulated in the DSL named Daut, detects such potentials.

```
monitor LockOrder {
  acquire(t1,a) -> {
    release(t1,a) -> ok
    acquire(t1,b) :: !(b = a) -> {
      acquire(t2,b) -> {
        release(t2,b) -> ok
        acquire(t2,a) -> error } } } }
```

Fig. 1. External Daut DSL: lock order monitor.

Constructing a parser for a language usually begins with designing an abstract syntax: data structures that represent the result of parsing. In Scala we usually define such as case classes, as shown in Figure 2 (top) for the first few top-level case classes for this DSL.[4] In this example, a specification is a list of

[3] The standard Scala library [26] originally included a parser combinator package. However, because it is rarely used, it was moved to an external library [24].

[4] A case class in Scala is like a class with the additional properties that it supports pattern matching over objects of the class, that the **new** keyword is not needed to create objects of the class, and equality is pre-defined based on the arguments.

automata; an automaton has a name and contains a lists of states. A state is defined by a collection of state modifiers (specifying the kind of state, for example whether it is an initial state or final state); a name; possible formal parameters (states can be parameterized with data); and transitions. A transition in turn consists of a pattern that must successfully match an incoming event, and an optional condition which must evaluate to true, for the transition to be taken, resulting in the right-hand side (a state expression) to be executed.

AST.scala:

```
case class Specification(automata: List[Automaton])

case class Automaton(name: Id, states: List[StateDef])

case class StateDef(modifiers: List[Modifier], name: Id,
  formals: List[Id], transitions: List[Transition])

case class Transition(pattern: Pattern, condition: Option[Condition],
  rhs: List[StateExp])
```

Grammar.scala:

```
def specification: Parser[Specification] =
  rep(automaton) ^^ {case automata => transform(Specification(automata))}

def automaton: Parser[Automaton] =
  "monitor" ~> ident ~
    ("{ " ~> rep(transition) ~ rep(statedef) <~ "}") ^^
    {case name ~ (transitions ~ statedefs) =>
        if (transitions.isEmpty) Automaton(name, statedefs) else { ... } }

def transition: Parser[Transition] =
  pattern ~ opt("::" ~> condition) ~ ("->" ~> rep1sep(stateexp, ",")) ^^
    {case pat ~ cond ~ rhs => Transition(pat,cond,rhs)}
```

Fig. 2. External Daut DSL: part of the Scala AST and parser combinator grammar.

Figure 2 (bottom) shows part of the parser expressed using Scala's parser combinator library. The parser generates an abstract syntax tree (AST) in the above-mentioned data structure. A `specification` is a `Parser`, which when applied to an input (as in `specification(`*input*`)`) produces an object of the (case) class `Specification`. A `specification` consists of zero or more (`rep`) `automaton`. The code after the `^^` symbol is a function, which takes as argument the parsed sub-data structures and returns the `Specification` object, in this case obtained by a transformation. An `automaton` is introduced by the keyword `monitor`, followed by an identifier, and a block delimited by `{` and `}`, which contains a sequence of transitions and a sequence of state definitions. Finally, a transition consists of a pattern, an optional guard condition, and one or more target states following the arrow. Transitions at the top level of a monitor represent an initial state. The symbols `~>`, `<~`, and `~` are methods in the parser combinator DSL that are used to represent spaces. This illustrates one of the

drawbacks of using an internal DSL such as parser combinators: the notation can get slightly inconvenient, lowering readability.

2.2 The parser tool approach

In the parser tool approach, a tool generates the parser program from a grammar. Parser tool frameworks that can be used with Scala include ScalaBison [27] and ANTLR [1]. ScalaBison [27] is a Scala parser generator, and hence allows Scala code to be written directly as the actions generating Scala objects. ANTLR [1] is Java-based and generates a parse tree as a Java object. This then has to be traversed (visited) by a Scala program and translated into a Scala tree. The advantage of ANTLR is its robust parsing algorithm. We use ANTLR for implementing the K language—a SysML-inspired language for specifying both high-level system descriptions as well as low-level implementations of a system. Note that ANTLR itself provides an external DSL for writing grammars.

Example.k:

```
class Instrument {
  var id : Int
  var name : String
  var powerLevel : Real }

class Spacecraft {
  var id : Int
  var instruments : [Instrument]
  req InstrumentsCount :
    instruments.count > 0 &&
    forall i,j:instruments . i != j => i.id != j.id }

assoc SpacecraftInstrument {
  part Spacecraft : Spacecraft
  part instruments: Instrument 1 .. 10 }
```

Grammar.g4:

```
grammar K;

classDeclaration:
  'class' Identifier extends? '{' memberDeclarationList? '}';

assocDeclaration:
  'assoc' Identifier  '{' assocMemberDeclarationList? '}';

expression:
  '(' expression ')' #ParenExp
  | Identifier #IdentExp
  | expression '.' Identifier #DotExp
...
```

Fig. 3. External K DSL: example of a model and ANTLR grammar.

Example: The K language Figure 3 shows an example model in K. Class `Instrument` specifies the instrument id, name, and power level. Class `Spacecraft` models the spacecraft, which contains several instruments with a unique id, as specified by constraint `InstrumentsCount`. Definition `SpacecraftInstrument` specifies that each `Spacecraft` can be *associated* with a maximum of 10 instruments, i. e., each `Instrument` contains a reference to the `Spacecraft` and the `Spacecraft` contains references to all instruments that are a *part* of it.

Using ANTLR, we create a tool chain to parse and process K artifacts. ANTLR accepts grammars specified in *Extended Backus-Naur-Form* (see Figure 3).Patterns are specified using rules and sub-rules that can be marked optional (`?`) and repetitive (`*`, `+`). Each rule can also be named using the `#` notation at the end of a rule. Keywords are specified in single quotes. In the example, rule `classDeclaration` specifies how classes are defined by an identifier, an optional *extends* declaration, and an optional class member declaration list enclosed within curly brackets. Similarly, we define the grammar for associations. The `expression` rule specifies how expressions can be constructed in K. Rule precedence is determined by the order of the occurrence of the rules.

Given a grammar, ANTLR produces a lexer and parser (in Java). ANTLR further enables one to create a *visitor* to visit the nodes in the generated parse tree. We implement a visitor in Scala by importing the required Java libraries and *stitching* the code together to access the ANTLR parse tree in Scala. Figure 4 shows a snippet of the visitor for the K language. The visitor makes use of classes defined in the K AST, also shown in Figure 4. Both snippets correspond to visiting expression nodes in the parse tree and creating expression declarations in the AST. For example, the visitor function `visitDotExp` takes as input a context `ctx` of type `DotExpContext`. This is used to extract the expression `e` and the identifier `ident` from the ANTLR parse tree. Together, these are used to create an instance of `DotExp`, which is defined in the K Scala AST. The K Scala visitor produces the complete AST, which can be used for code generation and analysis. We currently use the AST to transform K code to JSON and back. The robustness of ANTLR is a benefit, but there is also significant effort involved in creating the visitor that produces the Scala AST from the Java AST. A small change in the grammar can produce a cascading series of changes to the visitor.

3 Internal DSLs

An internal DSL extends the host language with custom constructs. We cover three variants: Annotations, shallow embedding, and deep embedding.

With *annotations,* a host program is annotated with information on existing language constructs. The host language needs to permit annotations, as is the case for Java and Scala. When using embedding, the extension is implemented as a library in the host language without any additional techniques. Here we distinguish between shallow and deep embedding. *Shallow embedding* uses the host language's features directly to model the DSL. The constructs have their usual meaning. In contrast, in a *deep embedding* one creates a separate representation

Visitor.scala:

```
override def visitDotExp(ctx: ModelParser.DotExpContext): AnyRef = {
  var e: Exp = visit(ctx.expression()).asInstanceOf[Exp]
  var ident: String = ctx.Identifier().getText()
  DotExp(e, ident) }

override def visitParenExp(ctx: ModelParser.ParenExpContext): AnyRef = {
  ParenExp(visit(ctx.expression()).asInstanceOf[Exp]) }
```

AST.scala:

```
trait Exp
case class ParenExp(exp: Exp) extends Exp
case class DotExp(exp: Exp, ident: String) extends Exp
case class IdentExp(ident: String) extends Exp
```

Fig. 4. External DSL: part of the ANTLR visitor and AST for K.

of the DSL: an abstract syntax (AST), which is then interpreted or translated, as in the case of an external DSL.

3.1 Annotations

Annotations in Java or Scala associate extra information with classes, fields, and methods. Java annotations accept a possibly empty list of key-value pairs of parameters, where parameters can be of primitive types, strings, classes, enumerations, and arrays of the preceding types. To be used at run-time with Scala, annotations have to be defined as Java annotations with run-time retention since Scala's annotations currently do not persist past the compilation stage.

Example: Modbat's configuration engine Many command line programs allow options to be set via environment variables or command line arguments. Existing libraries parse command line arguments, but still leave it to the user to check semantic correctness, such as whether a value is within a permitted range. This problem also occurs for Modbat, a model-based test tool [2].

```
@Doc("overrides environment variable CLASSPATH if set")
var classpath: String = "."

@Choice(Array("one", "two", "many"))
var simpleNumber = "one"

@Range(dmin = 0.0, dmax = 1.0) @Shorthand('p')
var defaultProbability = 0.5
```

Fig. 5. Example of annotations for configuration variables.

In Modbat, configuration options are expressed as annotated Scala variables. An internal library analyzes the annotations and parses command line argu-

ments, checking whether a parameter matches the name of an annotated variable. If so, the default value of the right variable is overridden by the new value. For example, the value for `defaultProbability` can be overridden by command line option `--default-probability=0.6`. Figure 5 shows an example. `@Doc` provides a documentation string for a usage/help output; `@Choice` limits options to a predetermined set; `@Range` defines the allowed range of a value; and `@Shorthand` defines a one-letter shorthand for a command line option.

It is also possible to use inheritance, naming conventions, or library calls, to represent or set certain attributes. However, these solutions are more limiting than annotations or require more code. Indeed, when Java annotations became available, many tools (such as JUnit [20]) adapted them.

3.2 Shallow embedding

In a shallow embedding the host language's features are used directly, with their usual meaning, to model the DSL, which is typically presented as an application programming interface (API).

Example: Data automata Recall the lock order monitor expressed in an external DSL in Figure 1. Figure 6 shows a version of this monitor expressed in an internal shallow DSL, also described in [15], and variants of which are described in [4,14]. Event types are defined as case classes sub-classing a trait `Event`.[5] The monitor itself is defined as a class sub-classing the trait `Monitor`, which is parameterized with the event type. The trait `Monitor` offers various constants and methods for defining monitors, including in this case the methods `whenever` and `state`, and the constants `ok` and `error`, which are not Scala keywords. The `LockOrder` monitor looks surprisingly similar to the one in Figure 1.

```
trait Event
case class acquire(thread:String,lock:String) extends Event
case class release(thread:String,lock:String) extends Event

class LockOrder extends Monitor[Event] {
  whenever {
    case acquire(t1, a) => state {
      case release(`t1`, `a`) => ok
      case acquire(`t1`, b) if b != a => state {
        case acquire(t2, `b`) => state {
          case release(`t2`, `b`) => ok
          case acquire(`t2`, `a`) => error } } } } }
```

Fig. 6. Internal shallow Scala DSL: lock order event definitions and monitor.

[5] A trait in Scala is a module concept closely related to the notion of an abstract class, as for example found in Java. Traits, however, differ by allowing a more flexible way of composition called mixin composition, an alternative to multiple inheritance.

The complete implementation of the internal DSL is less than 200 lines of code, including printing routines for error messages. Conversely, the similar external DSL is approximately 1500 lines (nearly an order of magnitude more code) and less expressive. The parts of the implementation of the internal shallow DSL directly relevant for the example in Figure 6 are shown in Figure 7.

A monitor contains a set of currently active states, the *frontier*. All the states have to lead to success (conjunction semantics). A transition function is a *partial function* (a Scala concept), which maps events to a set of target states. A state object (of the case class `state`) contains a transition function, which is initialized with the `when` function. A state is made to react to an event using `apply`.[6] There are several forms (sub-classes) of states, including `error`, `ok`, and `always` states that stay active even if the transition function applies to an event. Functions `state` and `always` take a transition function and return a new state object with that transition function. Function `whenever` creates an `always` state from the transition function and adds it to the set of initial states of the monitor.

```
class Monitor[E <: AnyRef] {
  private var states: Set[state] = Set()
  type Transitions = PartialFunction[E, Set[state]]

  class state {
    private var transitions: Transitions = noTransitions

    def when(transitions: Transitions) {
      this.transitions = transitions }

    def apply(event: E): Option[Set[state]] =
      if (transitions.isDefinedAt(event))
        Some(transitions(event)) else None }

  case object error extends state
  case object ok extends state
  class always extends state

  def state(transitions: Transitions): state = {
    val e = new state; e.when(transitions); e }

  def always(transitions: Transitions): state = {
    val e = new always; e.when(transitions); e }

  def whenever(transitions: Transitions) {
    states += always(transitions) } }
```

Fig. 7. Internal shallow Scala DSL: part of implementation.

The type of a transition function suggests that it returns a set of states. In Figure 6, however, the result of transitions (on the right of `=>`) are single states, not sets of states. This would not type check was it not for the definition of the

[6] The `apply` method in Scala has special interpretation: if an object `O` defines a such, it can be applied to a list of arguments using function application syntax: `O(...)`, equivalent to calling the `apply` method: `O.apply(...)`.

implicit function `convSingleState`, which lifts a single state to a set of states, here shown together with a selection of other implicit conversion functions:

```
implicit def convSingleState(state: state): Set[state] = Set(state)
implicit def convBool(b: Boolean): Set[state] = Set(if (b) ok else error)
implicit def convUnit(u: Unit): Set[state] = Set(ok)
implicit def convStatePredicate(s: state): Boolean = states contains s
```

These other implicit functions support lifting for example a Boolean value to an `ok` or `error` state (such that one can write a Boolean expression on the right-hand side of `=>`); the `Unit` value to `ok` (such that one can write statements with side-effects on the right-hand side); and finally a state to a Boolean, testing whether the state is in the set of active states, used in conditions.

Example monitor:

```
val lock_order = process (p => ?? {
  case acquire(t1, a) =>
    p || ?? { case release(`t1`, `a`) => STOP
              case acquire(`t1`, b) if a != b =>
                ?? { case acquire(t2, `b`) =>
                  ?? { case release(`t2`, `b`) => STOP
                       case acquire(`t2`, `a`) => FAIL }}}})
```

CSP$_E$ implementation:

```
abstract class Process { ... }

class Rec (f: Process => Process) extends Process {
  override def acceptPrim(e: AbsEvent): Process = f(this).accept(e) }

object CSPE {
  def process (f: Process => Process) = new Rec (f)
  def ??(f: PartialFunction[AbsEvent, Process]) = new ParamPrefix(f) }

class ParamPrefix(f: PartialFunction[AbsEvent, Process]) extends Process {
  override def acceptPrim(e: AbsEvent): Process =
    if (f.isDefinedAt(e)) f(e) else this }
```

Fig. 8. Example monitor written in CSP$_E$ and partial implementation.

Example: CSP$_E$ CSP$_E$ (CSP for events) is a run-time verification tool that uses a notation similar to Hoare's Communicating Sequential Processes (CSP) [18]. CSP$_E$ allows specification of "concurrent" processes. The top of Figure 8 shows the lock order monitor in CSP$_E$. Compared to the lock monitor of Figure 1, the major difference is that parallel composition of the top level process (`p || ...`) is required to run the monitor continuously. The similarity to data automata in the previous example is evident. The role of `p || ...` is similar to `always` and `process` is similar to `whenever` in data automata. The pattern match clauses are almost the same, except for `??`.

In CSP$_E$, recursive definitions are implemented by functions that take a process and return a new process (see Figure 8, bottom). Function `??` takes a partial function from events to processes (monitors), and creates a new process. That

process evaluates the given function if it is defined for a given event, and waits otherwise. The monitor specification supplies the event as the first argument of a process (before `=>`) and the behavior of the process to be executed after receiving the event, as the second one. Internally `acceptPrim` takes the first argument and executes `f` to continue monitoring the right-hand side of the expression.

CSP_E is implemented as an internal DSL. The main reason for this is to interface with external logging tools. The shallow embedding furthermore simplifies the implementation. However, due to this, the grammar of CSP_E slightly deviates from the standard CSP notation. For example, parametric events and the recursive definition of processes are more complicated than in standard CSP.

3.3 Deep embedding

In a deep embedding, a DSL program is represented as an abstract syntax tree (AST), which is then interpreted or translated as in the case of an external DSL. The AST is generated via an API. We shall show two deep DSLs, one for writing state machine driven tests, and one for rule-based trace monitoring.

Example.scala:

```
import Model._

class Example {
  def action { /* code */ }

  "a" -> "b" := { action } }
```

Transition.scala:

```
class Transition(val src: String,
                 val tgt: String) {
  var transfunc: () => Any = null

  def := (action: => Any) {
    transfunc = () => action
    Model.addTrans(this) } }
```

Model.scala:

```
object Model {
  /* implicit conversion for
     transition declaration */
  implicit def stringPairToTrans
    (tr: (String, String)) = {
    new Transition(tr._1, tr._2) }

  /* model data */
  val transitions =
    ListBuffer[Transition]()

  def addTrans(tr: Transition) {
    transitions += tr } }
```

Fig. 9. Implementation of a deep DSL: a miniaturized version of Modbat, given as an example (**Example.scala**) and two classes showing a partial implementation.

Example: Modbat The model-based test tool Modbat generates test cases from extended finite-state machines [2]. Modbat has been used to verify a Java model library and a SAT solver [2]. A Modbat model contains definitions of transitions: source and target states, and transition actions (code to be executed when a transition is taken). Figure 9 shows a simple example model and a minimalist implementation that registers the model data at run time. The key Scala features that are used for deeply embedding the DSL are the definition of a custom operator `:=` in Transition.scala, together with an implicit conversion of a string pair `"a" -> "b"`, to a transition (in Model.scala).

The design of Modbat's DSL mixes deep embedding for transitions, with annotations and shallow embedding for code representing transition actions on the system under test. The main goal is to make the syntax more declarative and concise where possible (e. g., to declare transitions), while avoiding too many new constructs when Scala code is used (hence the use of annotations and API functions where appropriate). Shallow embedding is ideal for transition actions as they have to interact with the Java run-time environment during test execution.

Example: LogFire LogFire is an internal (mostly) deep Scala DSL [16]. It was created for writing trace properties, as were the earlier described data automata DSLs in Sections 2.1 and 3.2 respectively. LogFire implements the Rete algorithm [9], modified to process instantaneous events (in addition to facts that have a life span), and to perform faster lookups in a fact memory. A monitor is specified as a set of rules, each of the form:

```
name -- condition1 & ... & conditionN |-> action
```

```
class LockOrder extends Monitor {
  val acquire, release = event
  val Locked, Edge = fact

  "acquire" -- acquire('t, 'l)                    |-> insert(Locked('t, 'l))
  "release" -- Locked('t, 'l) & release('t, 'l)   |-> remove(Locked)
  "edge"    -- Locked('t, 'l1) & acquire('t, 'l2) |-> insert(Edge('l1, 'l2))
  "cycle"   -- Edge('l1, 'l2) & Edge('l2, 'l1)    |-> fail() }
```

Fig. 10. Internal deep Scala LogFire DSL: lock order monitor.

Figure 10 illustrates the lock order property expressed as rules in LogFire. The rules operate on a database of facts, the *fact memory*. Rule left-hand sides check incoming events, as well as presence or absence of facts in the fact memory. Right-hand sides (actions) can modify the fact memory, issue error messages, and generally execute any Scala code (here the DSL becomes a shallow DSL). Class `Monitor` defines features for writing rules, for example the functions: `event`, `fact`, `--`, `&`, `|->`, `insert`, `remove`, and `fail`. Recall that in Scala, method names can be sequences of symbols, and dots and parentheses around method arguments are optional. Each rule definition in the monitor above is a sequence of method calls, that last of which is the call of the method `|->`, which produces an internal representation (an abstract syntax tree) of the rule as an object of a class `Rule`, which is then passed as argument to a method `addRule(rule: Rule)` in the Rete module. The abstract syntax class `Rule` is in part defined as:

```
case class Rule(name: String, conditions: List[Condition], action: Action)
case class Action(code: Unit => Unit)
```

A rule consists of a name; a left-hand side, which is a list of conditions, interpreted as a conjunction; and a right-hand side, which is an action. Conditions

```
implicit def R(name: String) = new {
  def --(c: Condition) = new RuleDef(name, List(c)) }

class RuleDef(name: String, conditions: List[Condition]) {
  def &(c: Condition) = new RuleDef(name, c :: conditions)

  def |->(stmt: => Unit) {
    addRule(Rule(name, conditions.reverse, Action((x: Unit) => stmt))) } }
```

Fig. 11. Internal deep Scala LogFire DSL: rule syntax implementation.

use deep embedding for optimization purposes. Actions are implemented as Scala code using a shallow embedding (functions from `Unit` to `Unit`).

The definitions in Figure 11 support the transformation of a rule entered by the user to an AST object of the class `Rule`. The implicit function `R`, lifts a string (a rule name) to an anonymous object. That object defines the `--` operator, which when applied to a condition returns an object of the class `RuleDef`. This class in turn defines the condition conjunction operator `&` and the action operator `|->` defining the transition from left-hand side to right-hand side of the rule. This operator calls `addRule`, which adds the rule to the Rete network. The implicit function `R` gets invoked by the compiler automatically when a string is followed by the symbol `--`, to resolve the type "mismatch" (as no `--` operator is defined on strings). The individual conditions in a rule are similarly constructed with the help of the following implicit function, which lifts a symbol (the name of an event or fact) to an object, which defines an `apply` function:

```
implicit def C(kind: Symbol) = new {
  def apply(args: Any*): Condition = ... }
```

The complete interpretation by the Scala compiler of the rule `"release"` in Figure 10 becomes:

```
R("release").--(C('Locked).apply('t,'l)).&(
  C('release).apply('t,'l)).|->(remove('Locked))"")
```

4 Discussion

We discuss the characteristics of five approaches to DSL implementation from our experience with DSLs for formal modeling (see Table 1 for a summary).

Parser generation The external tool approach requires an extra *parser generation* and compilation stage, where a parser is first generated from a grammar specification, and then compiled. The other approaches have no code generation stage, which slightly facilitates development.

AST generation All approaches except the internal shallow approach generate ASTs. AST generation can complicate matters and at the same time be a facilitator. It influences topics such as transformation analysis, executability, Turing completeness, ease of development, and flexibility of syntax, as discussed below.

Table 1. Characteristics of different DSL implementation approaches. Signature: $\surd$ means yes, $\times$ means no, $\otimes$ means no unless an effort is made to make it happen, and $++$, $+$, $-$, $--$ rank different approaches from best to worst.

	External		Internal		
	tool	lib	annotations	deep	shallow
Parser generation	$\surd$	$\times$	$\times$	$\times$	$\times$
AST generation	$\surd$	$\surd$	$\surd$	$\surd$	$\times$
Enables transformation, analysis	$\surd$	$\surd$	$\surd$	$\surd$	$\times$
Directly executable	$\times$	$\times$	$\times$	$\times$	$\surd$
Turing complete	$\otimes$	$\otimes$	$\times$	$\surd$	$\surd$
Ease of development	$--$	$-$	$++$	$+$	$++$
Flexibility of syntax	$++$	$++$	$--$	$-$	$--$
Quality of syntax error messages	$+$	$+$	$++$	$--$	$-$
Ease of use	$++$	$++$	$+$	$-$	$--$

Enables transformation and analysis An AST allows us to transform and analyze the DSL. Specifically, it allows us to optimize any code generated from the AST, a capability particularly important for the shown monitoring DSLs. This is not directly possible in the internal shallow approach, which is one of its main drawbacks. One could potentially use reflection, but the Scala reflection API does not support reflection on code. Alternatively one can use a bytecode analysis library such as ASM [8] or the Scala compiler plugin. However, these solutions require a great level of sophistication of the developer.

Directly executable The internal shallow approach has the advantage that the DSL is directly executable since the host language models the DSL. There is no AST to be interpreted/translated. This again means that internal shallow DSLs are faster to develop, often requiring orders of magnitudes less code.

Turing complete Annotations typically carry simple data, and are usually not Turing complete (although they can be made to be). Internal DSLs are by definition Turing complete since they extend a Turing complete host language (in our case, Scala). For internal shallow DSLs the host language is directly part of the DSL, thus making the DSL itself Turing complete. Our experience with internal DSLs is that the user of the DSL will use the host language constructs in case the DSL is not applicable to a particular problem. As an example, an internal DSL lends itself to writing "glue code" to connect the DSL with another system, such as the system under test in case of a test DSL. It is more challenging to turn external DSLs into Turing complete languages.

Ease of development External DSLs developed using a library (such as Scala's parser combinators) seem easier to develop than using a parser generator tool (such as ANTLR) due to the reduced parser generator step. However, using a parser generator such as ANTLR facilitates grammar development itself since

ANTLR accepts more grammars than for example Scala's parser combinators. Annotations-based DSLs are easy to develop since the Java compiler and the core libraries support annotations. However, it is not possible to extend the syntax or scope of annotations in any way. It furthermore appears that internal DSLs are easier to develop than external DSLs, and that internal shallow DSLs are the easiest to develop.

Flexibility of syntax Our experience with internal DSLs is that it can be a struggle to achieve the optimal syntax. This is mostly due to limitations in operator composition and precedence in Scala. In an external DSL one is completely free to create any grammar as long as it is accepted by the parser.

Quality of syntax error messages External DSLs have a potential for good error messages, depending on the toolkit used. Internal DSLs often result in error messages that can be intimidating to users, especially if not used to the host language. In the case of deep internal DSLs, conversion functions may show up in compiler errors; or a missing symbol may result in a lack of conversion, in which case no error message is shown or a completely wrong one. Furthermore, for a deep internal DSL a type checker has to be developed from scratch. In contrast, shallow internal DSLs have the advantage that Scala's type system takes care of type checking.

Ease of use Annotations and internal DSLs are usually adopted easily if the users are already host language programmers. As an illustration, in spite of being originally a research tool, TraceContract [4], a variant of the internal data automaton DSL illustrated in Section 3.2, was used throughout NASA's LADEE Moon mission for checking all command sequences before being sent to the spacecraft [5]. Similarly, the internal DSL LogFire [16], illustrated in Section 3.3, also originally a research tool, is currently used daily for checking telemetry from JPL's Mars Curiosity Rover [17]. These adoptions by NASA missions were not likely to have happened had these DSLs been external limited stand-alone languages. On the other hand, if a user is not already a host language programmer (and is not willing to learn the host language), it may be easier to adopt an external DSL. For example, we developed an external monitoring DSL much along the lines of data automata, and had non-programmers use it for testing without much training [3]. More interestingly perhaps, the SysML-inspired external modeling DSL K, illustrated in Section 2.2, is planned to be adopted by a JPL's future mission to Jupiter's Moon Europa, for modeling mission scenarios.

5 Conclusions

We have presented five approaches to implementing domain-specific languages (DSLs) in Scala, illustrated by application to formal modeling and testing languages. External DSLs use either (1) a parser generator, or (2) a parser library. Internal DSLs extend the host language by (3) annotations of existing language

elements, (4) deep embedding where an abstract representation of the program is computed, or (5) shallow embedding of functions that directly execute. Our experience shows that each approach has its strengths; in particular, external DSLs can offer a fully flexible syntax while internal DSLs are easier to develop and in the case of shallow embedding, are directly executable. Mixed approaches are common, in particular for internal DSLs. Future work includes leveraging macro-based and compiler-based approaches, which promise to combine some of the strengths of the techniques discussed here.

References

1. Antlr. `http://www.antlr.org`.
2. C. Artho, A. Biere, M. Hagiya, E. Platon, M. Seidl, Y. Tanabe, and M. Yamamoto. Modbat: A model-based API tester for event-driven systems. In Valeria Bertacco and Axel Legay, editors, *Proc. 9th Haifa Verification Conf. (HVC 2013)*, volume 8244 of *LNCS*, pages 112–128, Haifa, Israel, 2013. Springer.
3. Howard Barringer, Alex Groce, Klaus Havelund, and Margaret Smith. Formal analysis of log files. *Journal of Aerospace Computing, Information, and Communication*, 7(11):365–390, 2010.
4. Howard Barringer and Klaus Havelund. TraceContract: A Scala DSL for trace analysis. In *Proc. 17th Int. Symposium on Formal Methods (FM 2011)*, volume 6664 of *LNCS*, pages 57–72, Limerick, Ireland, 2011. Springer.
5. Howard Barringer, Klaus Havelund, Elif Kurklu, and Robert Morris. Checking flight rules with TraceContract: Application of a Scala DSL for trace analysis. In *Scala Days 2011, Stanford University, California*, 2011.
6. Eric Bodden. MOPBox: A library approach to runtime verification. In Koushik Sen and Sarfaz Khurshid, editors, *Proc. 2nd Int. Conf. on Runtime Verification (RV 2011)*, volume 7186 of *LNCS*, pages 365–369, San Francisco, USA, 2011. Springer.
7. T. Bray, J. Paoli, M. Sperberg-McQueen, E. Maler, and F. Yergeau. Extensible markup language (XML). *World Wide Web Consortium Recommendation REC-xml-19980210*, 1998.
8. E. Bruneton, R. Lenglet, and T. Coupaye. ASM: a code manipulation tool to implement adaptable systems. In *Adaptable and Extensible Component Systems*, Grenoble, France, 2002.
9. Charles Forgy. Rete: A fast algorithm for the many pattern/many object pattern match problem. *Artificial Intelligence*, 19:17–37, 1982.
10. Martin Fowler and Rebecca Parsons. *Domain-Specific Languages*. Addison-Wesley, 2010.
11. Mario Fusco. Hammurabi—a Scala rule engine. In *Scala Days 2011, Stanford University, California*, 2011.
12. E. Gansner and S. North. An open graph visualization system and its applications to software engineering. *Softw. Pract. Exper.*, 30(11):1203–1233, 2000.
13. GLL combinators. `https://github.com/djspiewak/gll-combinators`.
14. Klaus Havelund. Data automata in Scala. In Martin Leucker and Ji Wang, editors, *Proc. 8th Int. Symposium on Theoretical Aspects of Software Engineering (TASE 2014)*, Changsha, China, 2014. IEEE Computer Society Press.
15. Klaus Havelund. Monitoring with data automata. In T. Margaria and B. Steffen, editors, *Proc. 6th Int. Symposium On Leveraging Applications of Formal Methods, Verification and Validation.*, volume 8803 of *LNCS*, pages 254–273, Corfu, Greece, 2014. Springer.

16. Klaus Havelund. Rule-based runtime verification revisited. *Software Tools for Technology Transfer (STTT)*, 17(2):143–170, 2015.
17. Klaus Havelund and Rajeev Joshi. Experience with rule-based analysis of spacecraft logs. In Cyrille Artho and Peter Ölveczky, editors, *Proc. 3rd Int. Workshop on Formal Techniques for Safety-Critical Systems (FTSCS 2014)*, Communications in Computer and Information Science (CCIS), Luxembourg, 2014. Springer.
18. C. Hoare. Communicating sequential processes. *Communications of the ACM*, 26(1):100–106, 1983.
19. F. Jakob. SugarScala: Syntactic extensibility for Scala. Master's thesis, Technische Universität Darmstadt, 2014.
20. J. Link and P. Fröhlich. *Unit Testing in Java: How Tests Drive the Code*. Morgan Kaufmann Publishers, Inc., 2003.
21. M. Mernik, J. Heering, and A. Sloane. When and how to develop domain-specific languages. *ACM Comput. Surv.*, 37(4):316–344, 2005.
22. M. Odersky, L. Spoon, and B. Venners. *Programming in Scala: A Comprehensive Step-by-step Guide*. Artima Inc., USA, 2nd edition, 2010.
23. Parboiled. `https://github.com/sirthias/parboiled2`.
24. Parser combinators. `https://github.com/scala/scala-parser-combinators`.
25. T. Rompf, N. Amin, A. Moors, P. Haller, and M. Odersky. Scala-Virtualized: Linguistic Reuse for Deep Embeddings. *Higher Order and Symbolic Computation*, August-September:1–43, 2013.
26. Scala. `http://www.scala-lang.org`.
27. ScalaBison. `https://github.com/djspiewak/scala-bison`.
28. Scala macros. `http://scalamacros.org`.
29. Volker Stolz and Frank Huch. Runtime verification of concurrent Haskell programs. In *Proc. of the 4th Int. Workshop on Runtime Verification (RV 2004)*, volume 113 of *ENTCS*, pages 201–216. Elsevier, 2005.
30. B. Venners. ScalaTest, 2014. http://www.scalatest.org.