A Case Study in DSL Development

An Experiment with Python and Scala

Klaus Havelund Michel Ingham David Wagner

Jet Propulsion Laboratory, California Institute of Technology *

{klaus.havelund, michel.d.ingham, david.a.wagner}@jpl.nasa.gov

Abstract

This paper describes an experiment performed with developing a Domain Specific Language (DSL) for monitoring and control of the launch platform for future Constellation rockets at NASA's Kennedy Space Center in Florida, USA. The Constellation project has been conceived as NASA's replacement of the current aging space shuttle program, with the extended objective of sending humans back to the moon, and subsequently to Mars. The DSL effort was specifically performed for the NASA Constellation Launch Control System (LCS) project. The main experiment was performed using simulators of the existing space shuttle launch platform, and included designing and implementing a prototype in the Python programming language, chosen for its succinct notation. A later study was carried out where part of the DSL was implemented in Scala, and compared to the Python implementation from a linguistic DSL elegance point of view.

Categories and Subject Descriptors D.2.1 [*Software Engineering*]: Requirements/Specifications; D.3.3 [*Programming Languages*]: Language Constructs and Features

General Terms languages, measurement, verification

Keywords domain specific languages, system monitoring and control, Python, Scala

1. Introduction

NASA's Constellation program [3] has as purpose to replace the aging space shuttle fleet with new vehicles. The program's success will require significant upgrades to the ground-based infrastructure needed to assemble, test, and operate these new vehicles. One such system is the Launch Control System (LCS) at the Kennedy Space Center (KSC). In addition to coordinating and controlling the launch sequence, this system will be used to test the spacecraft, launch vehicles, and possibly their component subsystems as they are delivered to the Space Center and assembled for launch, to control various pieces of ground support equipment used during operations, and to ensure the safety of all of these operations.

A considerable expense is the extensive process by which system engineers express requirements for test procedures in prose, software developers translate these requirements into code, and then both sets of experts are engaged in verification of the resulting application's correctness. One element of the study was an investigation of the potential benefits of using a Domain Specific Language (DSL) that systems engineers would be able to use to write executable specifications of monitor and control applications (i.e., capture detailed requirements in a form that would either be directly executable or automatically translatable to software implementation). Engineers at KSC, in collaboration with personnel from JPL (including the authors) conducted a study of DSLs for programming such systems over a roughly one-year period from late 2006 to late 2007. Several existing domain specific languages were studied, as described in [1]. In addition, it was decided to develop a home-grown experimental DSL.

Two approaches to defining such a DSL were considered. The *stand-alone DSL* is a DSL developed from scratch, and translated into a general purpose programming language, or interpreted in such a language. The *integrated DSL* is an extension of an existing general purpose programming language, possibly just as an API. The advantage of a stand-alone DSL is that its size can be kept small, which is an important factor when used by systems engineers, who are normally not programmers. The advantage of an integrated DSL is that it allows for fully general purpose programming (by expert programmers) in cases where this is needed. With a stand-alone DSL one would have to switch from the DSL to the general purpose language in some, possibly rare, situations. While a stand-alone DSL will be suited to monitor and control applications, it will also tend to have more limited development and/or execution environments and tool support (syntax highlighting, debugging, etc.). General-purpose languages offer significant flexibility in terms of implementing required functionality and have impressive portability and maturity characteristics, but suffer primarily in terms

Part of the research described in this publication was carried out at the Jet Propulsion Laboratory, California Institute of Technology, under a contract with the National Aeronautics and Space Administration.

of readability and verifiability considering the targeted user base of systems engineers.

Based on these observations it was decided to develop an integrated DSL, putting emphasis on flexibility, functionality, portability and maturity. An experimental implementation of a DSL was developed as a library in the dynamically typed Python scripting/programming language [5]. Python was selected because it was considered to present the smallest "semantic gap" and shallowest learning curve to a systems engineer user, among the general-purpose programming languages considered (Scala was not considered at the time of the original study in 2006-2007). An additional experimental implementation of elements of the same DSL was later developed in the statically typed programming language Scala [6], as also described in [2]. Scala was not considered main stream at the time of the first experiment, and only came to the attention of the authors thereafter. The language, however, appears to have several advantages for DSL definition. The paper presents the results from this experiment and briefly compares the two efforts. Java and C++ were also considered for implementation of a DSL, but were both considered less appropriate for this task, mainly due to the lack of function values, a concept used extensively in the Python DSL.

2. The Python DSL

The system should in a simplified view support communication between (i) CONTROL: computers running *applications* and human operators interacting with *displays*, and (ii) PLATFORM: the shuttle launch platform and equipment around it, also referred to as *end items*. The CONTROL should be able to read telemetry emitted from the PLATFORM, verify its well-formedness, and submit commands back to the PLATFORM, possibly guided by humans via communication on console displays in the launch control room. Middleware and gateways connect the components in this network. We shall focus attention on the modeling of the measurement database and a function `verify_within` for monitoring that some condition becomes true within some time frame.

2.1 Measurements

From the point of view of a control application, the state can be seen as a mapping from measurement variable names (each associated with a sensor in an end item) to measurements. A measurement object is an instance of the following class, for which only some of the methods are shown:

```
class Measurement:
  def __init__(self, id, value): ...
  def getId(self): return self.id
  def getValue(self): return self.value
  def getTime(self): return self.time
  def __lt__(self, other): ...
  ...
```

A measurement object contains a name, a value[1], and a time tag. The time tag is automatically inserted in the object upon creation. The class defines a set of mathematical relational methods for comparing values (`__lt__` , `__eq__` ,), corresponding to the relational operators $<$, $=$, etc. The methods are named in such a way that they overload the built-in relational symbols. For example, given two measurements `m1` and `m2`, they can be compared using traditional syntax: `m1 < m2`. One has to define all arithmetic operations on measurements this way if one wants to use the standard arithmetic notation on measurement objects. The measurements are stored in a mapping from names to Measurements in an object `ms` of the class `MeasurementService`:

```
class MeasurementService:
  def publish(self, name, value): ...
  def getByName(self, name): ...
  def __getattr__(self,name):
    return self.getByName(name)
  def __setattr__(self, name, value):
    self.publish(name, value)

ms = MeasurementService()
```

When end items change status at the PLATFORM the new measurements are stored in the `ms` object. The function `__getattr__(self,name)` is called by the Python interpreter on an attribute when the attribute is referred to in `ms` but not found in the object. That is, a reference of the form `ms.x` results in a call of `ms.__getattr__("x")` if `x` is not defined as a method or field in `ms`. It is now possible to write statements like:

```
if ms.pressure < 300:
  doSomething()
```

Note specifically that it is not possible to refer to `pressure` without some additional notation to cause a lookup in the database for the real value. For this to work it is necessary that the script writer has access to the `ms` object, or generally an object providing a `__getattr__` method. An alternative is to define a reader function as follows:

```
def read(name):
  return ms.getByName(name)
```

If we define Python variables for the measurement names, for example:

```
pressure = "pressure"
```

we can alternatively write:

```
if read(pressure) < 300:
  doSomething()
```

[1] Values are here for simplicity considered to be integers. In the real system values can have different types.

It is also possible for scripts to publish measurements to be shared with other scripts that run in parallel, operating different parts of the launch platform. This is done by calls of the function `set` defined as follows:

```
def set(name,value):
  ms.publish(name,value)
```

For example, a shared measurement is published as follows:

```
set("pressure1",80)
```

The setter counterpart to the `__getattr__` method in the `MeasurementService` class above is the `__setattr__` method. It allows us to achieve the same effect by writing:

```
ms.pressure1 = 80
```

The function `derive(X,F)` defines a new measurement named X derived from another expression, in Python represented as a function F. A reference to `ms.X` will thereafter return the value of `F()`. One can for example define a variable `"pressure2"` to be derived from `pressure` as follows:

```
derive("pressure2",lambda: ms.pressure * 1000)
```

Note the use of a lambda abstraction to delay the evaluation of the expression `ms.pressure * 1000`.

2.2 The verify_within function

The DSL monitoring functions offer capabilities for testing the values of named measurements as a function of time. A monitoring function is characterized along five dimensions: the condition to check, for example `pressure < 300`, the period within or during which the condition should hold, a reaction to be executed in case the property is violated, whether checking should continue if the property gets violated, and finally whether the construct is blocking (in which case the calling application will wait until the verification has been performed), or whether the verification is spawned to the background. In a non-blocking case where a reaction is to be executed upon violation of the condition, there is a further choice between letting this reaction execute in parallel with the calling application or letting it interrupt the calling application. All these functionalities were implemented via one single heavily parameterized function, hidden from the user, and then called in a collection of functions available to the user for sending commands as well as verify telemetry.

We shall focus on one function called `verify_within` which checks that a given condition becomes true within a certain time period, otherwise a reaction optionally provided in the call is executed. The command is blocking. If the condition becomes true within its duration, then the function returns immediately without waiting for the duration to expire. Its definition has the form (body not shown):

```
def verify_within(condition,duration,
  reaction=DIALOG, name=""):
  ...
```

The reaction and name are both optional parameters (can be left out in the call) with default values. `DIALOG` is constant (an integer) indicating that a dialog with the user should be started in case no other reaction is provided. The following call verifies that the value of `pressure` becomes bigger than or equal to 300 within 10 seconds, otherwise a message is sent to a display:

```
verify_within(
  lambda: ms.pressure >= 300,
  10,
  lambda: display("pressure error"),
  "P1"
)
```

Note how lambda abstractions are used to delay evaluation of the condition and the reaction until the `verify_within` function calls these in its body. This allows for example the condition to be re-tested repeatedly in the body.

3. The Scala DSL

Scala provides, as described in [4], all the features Python supports relevant for definition of the DSL, such as higher order functions and default arguments (introduced in Scala version 2.8.0). However, in Scala the body of a lambda abstraction can contain any number of statements and expressions. This is in contrast to a lambda abstraction in Python, the body of which can only contain one expression[2]. In addition to this, Scala offers some features that further ease definition of domain specific languages:

- Call-by-name, also referred to as automatic closure construction. This allows to delay the evaluation of statements and expressions without using lambda abstractions, making the DSL notation more user-friendly.
- Curried functions, allowing arguments to be naturally separated without having to appear in a comma-separated argument list.
- Overloaded methods (functions defined in classes or objects), allowing minor variants of a DSL construct to have the same name (for major variants one might want to have different names). Method overloading to some extent can be used to avoid default arguments, and thereby a comma-separated list of arguments.
- Infix notation for method calls, permitting definition of DSL constructs that appear as built-in language constructs.
- Implicit conversions: Scala allows for the definition of so-called implicit functions that convert values of one type to values of another type. Such functions are applied

[2] The body of a lambda abstraction in Python can be a call to a function, the body of which can contain several statements, but this is not considered convenient in this context.

automatically in places where it will correct a type problem.

In the following discussion, it will be demonstrated how these concepts can be used to define the `verify_within` construct in three variations. First, however, it will be demonstrated how measurements can be modeled, allowing for maximal notational convenience.

3.1 Measurements

Measurements are defined in a class similar to the Python equivalent[3]:

```
class Measurement(id:Symbol, value:Int) {
  private val time:Long =
    System.currentTimeMillis()

  def getId:Symbol = id
  def getValue:Int = value
  def getTime:Long = time
}
```

A main difference is that values and methods are now typed. Measurement ids are represented as symbols in the Symbol data type, which contains quoted identifiers such as `'pressure`. Symbols are slightly more convenient to write than their string counterparts: `"pressure"`. Second, measurements are published in a mapping from symbols to measurements in the measurement service object `ms` (we only show the methods):

```
object ms {
  def publish(name:Symbol,value:Int)
    {...}
  def getByName(name:Symbol):Measurement =
    {...}
  ...
}
```

Assume a value declaration of the form (a user may decide to get rid of symbol quotes this way):

```
val pressure = 'pressure
```

Without any further definitions, a measurement would be accessed as follows:

```
if (ms.getByName(pressure).getValue < 300)
  doSomething()
```

or, if the above is defined as a function named `read`:

```
def read(s:Symbol) = {
  ms.getByName(s).getValue
}
```

we can at best write:

```
if (read(pressure) < 300)
  doSomething()
```

In order to avoid such still slightly heavy notation, an implicit conversion function can be defined from symbols to integers:

```
implicit def conv1(s:Symbol):Int = {
  ms.getByName(s).getValue
}
```

The name of this function is unimportant since it will be applied by the Scala compiler under the hood to make expressions type check. That is, this function allows us to write:

```
if (pressure < 300)
  doSomething()
```

Scala will automatically apply the appropriate conversion function (`conv1` in this case) to obtain an integer from `pressure`. Hence the condition is equivalent to:

```
conv1('pressure) < 300
```

It is of course a question whether such implicit conversions are safe to use for programming safety critical systems. It is possible for a user to create unsafe code due to unexpected type conversions. As an example, a user could define a function to convert milliseconds to seconds:

```
def seconds(milliseconds:Int):Double = {
  milliseconds/1000
}
```

and later apply it as follows:

```
if (seconds('pressure) < 80)
  goForLaunch()
```

The wrongly programmed condition would then type check since `'pressure` is a `Symbol` now occurring where an integer is expected, causing the `conv1` conversion function to be applied to make it type check, whereas in this case we would want this condition not to type check.

Derived values can be defined as follows using Scala's `def` construct:

```
def pressure2 = pressure * 1000
```

Each time `pressure2` is now referenced, the expression `pressure * 100` is evaluated according to the rules already described, including application of the implicit conversion function. The corresponding Python notation was:

```
derive(pressure2,lambda: ms.pressure * 1000)
```

Implicit conversion from symbols to integers in addition gives us all the operators on integers to work on measurements, without having to define them as methods in the `Measurement` class, as we did in Python.

[3] Scala allows for an even more succinct definition of this class using `val` constructor arguments, thereby avoiding getter-functions.

3.2 The verify_within Construct : Solution 1

In the following we shall illustrate three approaches to define the `verify_within` construct. In Scala, we have to declare the type of a reaction explicitly. This can be done as follows using Scalas version of algebraic types, namely case classes:

```
abstract class Reaction
  case object DIALOG
    extends Reaction
  case class CODE(code:()=>Unit)
    extends Reaction
```

We can now define a function with the following signature:

```
def verify_within(
  cond:()=>Boolean,
  time:Int,
  reaction:Reaction = DIALOG,
  name:String = "") = {
  ...
}
```

This definition looks much like its corresponding definition in Python, except for the added type information. With such a definition we would be able to write:

```
verify_within(() => pressure>=300, 10,
  DIALOG)

verify_within(() => pressure>=300, 10,
  CODE(() => display("pressure error")))
```

Note the required application of the `CODE(...)` object constructor. If, however, we define an implicit conversion function:

```
implicit def conv2(code:()=>Unit):Reaction =
{
  new CODE(code)
}
```

we can now, instead of the previous call of `verify_within` above, write:

```
verify_within(() => pressure>=300, 10,
  () => display("pressure error"))
```

This now has the same appearance as the Python code, but has the advantage of being statically typed.

3.3 The verify_within Construct : Solution 2

In the second version, we define a function that is curried (ignoring the name-argument), and which uses call-by-name formal parameters for the condition:

```
def verify_within
      (cond:=>Boolean)
      (time:Int)
      (reaction:Reaction) = { ... }
```

Currying will make applications of the function more readable. The first parameter to the function, the condition, has the type: '`=> Boolean`', representing a call-by-name Boolean parameter type. When applying the function to a Boolean expression, this expression is not evaluated until it is referred to inside the body of `verify_within`. For example, we can write:

```
verify_within (pressure>=300) (...) (...)
```

without the expression `pressure>=300` being evaluated at call time. In addition, we can define an implicit conversion function from call-by-name statements to reactions:

```
implicit def conv3(code:=>Unit):Reaction =
{
  new CODE(() => code)
}
```

With these definitions we can now write:

```
verify_within (pressure>=300) (10) (DIALOG)
```

and:

```
verify_within(pressure>=300) (10) {
 display("pressure error")
}
```

In this last call, the implicit conversion function `conv3` is applied to the block: '`{display("pressure error")}`'. This notation appears more user-friendly than solution 1 and the Python solution. The solution is also safe in the sense that should one forget some of the arguments, as in:

```
verify_within (pressure>=300) (10)
```

the compiler will emit the error message: "missing arguments for method `verify_within`". It is only possible to omit arguments to a curried function if the call appears in a context where the type of the partial application matches or if the call is followed by the symbol '_'.

3.4 The verify_within Construct : Solution 3

The final solution consists of attempting to give a programming language feel to the syntax. Instead of the above solution we would want to write:

```
verify (pressure>=300) within 10 onfail
  DIALOG
```

and:

```
verify (pressure>=300) within 10 onfail {
  display("pressure error")
}
```

Consider that such a construct were to be defined by a grammar with a non-terminal for each phrase starting with a key-

word[4], using $\langle N \rangle$ to indicate a non-terminal N and [...] to indicate optional:

$\langle \textit{Verify} \rangle \leftarrow \textbf{verify}\ [\langle \textit{Name} \rangle]\ \langle \textit{Cond} \rangle\ \langle \textit{Within} \rangle$
$\langle \textit{Within} \rangle \leftarrow \textbf{within}\ \langle \textit{Int} \rangle\ \langle \textit{OnFail} \rangle$
$\langle \textit{OnFail} \rangle \leftarrow \textbf{onfail}\ \langle \textit{Reaction} \rangle$

This grammar structure can be emulated in an object-oriented language, as illustrated by the following Scala object, which is intended to define the above grammar and its semantics:

```
object Verify {
  def verify(cond:=>Boolean):Within =
  {
    new Within("", cond)
  }

  def verify(name:String)
            (cond:=>Boolean):Within =
  {
    new Within(name, cond)
  }

  protected class Within(name:String,
                         cond:=>Boolean)
  {
    def within(time:Int):OnFail = {
      new OnFail(name, cond, time)
    }
  }

  protected class OnFail(name:String,
                         cond:=>Boolean,
                         time:Int)
  {
    def onfail(reaction:Reaction) {
      // implementation of construct
    }
  }

  implicit def conv4(code:=>Unit):Reaction =
  {
    new CODE(() => code)
  }
}
```

The object defines two overloaded `verify` functions, corresponding to the optional nature of the property name, one method not taking a name as argument and one taking a name. Each of these functions return an object of the nested class `Within`. The `Within` class itself defines a `within` method, which when applied to a time value returns an object of the class `OnFail`. This class finally defines the `onfail` method, which at this point has access to all information (name, condition, time and reaction), and hence can execute the semantics of the construct (not shown). The implicit conversion function converts code to reactions. With this definition we can write:

```
verify(pressure>=300).within(10).onfail({
  display("pressure error")
})
```

However, Scala allows method calls using infix notation. That is, given an object o and a method m in o, instead of writing $o.m(a)$ as in traditional OO languages, it is possible to write: '$o\ m\ a$' omitting the dot ('.') and the parentheses around the argument. We can therefore write:

```
verify (pressure>=300) within 10 onfail {
  display("pressure error")
}
```

The construct now appears like any other programming construct and from a DSL design point of view looks ideal (ignoring opinions on how this particular construct should be designed, which is not the topic of this paper).

The approach, however, has a couple of minor issues associated with it, that can make programming unsafe. First, it is for example possible to write only part of the construct, as in:

```
verify (pressure>=300) within 10
```

leaving out the `onfail {...}` part. The compiler will not complain. Furthermore, if no special runtime checking is performed to detect this (for example by ensuring that all begun constructs are ended before a new is begun), no warnings will be issued during runtime, either. However, a solution for this problem might be implemented in a future version of the Scala compiler, as communicated by Scala's designer Martin Odersky [4]. An analysis will be provided for checking whether a function call has side-effects or not. Odersky suggests in [4] that with such an analysis it is possible to disallow pure expressions as statements, essentially disallowing the partially instantiated DSL construct above (note that all the effect is in the `onfail` method).

Another problem is due to Scala's semicolon inference. It is not possible to write for example:

```
verify (pressure>=300) within 10
onfail {
  display("pressure error")
}
```

where the `onfail` DSL keyword has been moved to a line for itself. The compiler will infer a semicolon after the first line, and will subsequently not be able to associate the name `onfail` with the method defined in the `OnFail` object which is the result of the first line. There is no obvious solution to

[4] This is not the most succinct formulation of a grammar for this construct, but serves to illustrate the encoding of the grammar in Scala.

this problem beyond avoiding such line breaks, or enclosing the entire statement in between parentheses `( ... )`, in which case line breaks are allowed. Note, however, that fortunately it is possible to write:

```
verify (pressure>=300) within 10 onfail {
  display("pressure error")
}
```

4. Evaluation and Conclusions

We have above seen 3 approaches to defining a DSL construct in Scala. The first approach corresponds to the way a DSL is defined in Python. The notation is just as succinct as in Python, and more succinct than it would be in Java due to Java's lack of function values. As an added advantage, in contrast to the Python solution, the Scala solution is statically typed, which will prevent many programmer mistakes. The second approach, using overloading, currying and call-by-name, yields a solution that from a notational point of view is even more succinct and clear. This solution is still as safe as the previous solution with respect to the compiler being able to detect programming errors. On the other hand, the third approach, which attempts to give more of a textual programming language feel to the syntax, was demonstrated unsafe in the current version 2.8.0 of Scala, but might become safe in future versions of Scala.

Common for the Scala solutions is the use of implicit conversion functions. This concept does introduce a potential for programming errors due to conversions being automatically performed in locations where it is not intended. This powerful and useful programming language feature on its own could deserve further investigation. Are there for example ways to make implicit conversions safe, potentially by restricting their application to certain contexts?

Scala seems to have qualities that make it a good platform for *integrated* DSL development (where the DSL is an extension of a general purpose programming language) compared to Python, Java and C++. It offers language concepts that make it possible to construct a DSL optimized for ease of use. It is statically typed, with a notationally succinct flavor comparable with a scripting language like Python (amongst other things due to type and semicolon inference). It is compatible with Java, which would make it easy to integrate with other parts of a system, for example the middleware. It has a concurrency model more appropriate for the DSL than Python's, which only allows one thread to execute at any point in time, preventing utilization of multi-core machines. Of things that could be desired from Scala based on this experience: (i) Ability to check completion of DSL constructs as mentioned above (can be solved, as suggested by Martin Odersky [4], with a compiler check for statements without side-effects and where the return value is not used). (ii) Permission to omit parentheses in function applications: it is a slight annoyance that one has to for example put parentheses around the time value in solution 2 above. (iii) Alternative handling of parameterless functions: if a DSL construct ends with a method that takes no arguments (modeling a terminating keyword), and the `()` is left out in the call to make it look like a keyword, the compiler will look at the next line and regard this as a superfluous argument in case that line is non-empty (and give an error message). (iv) Domain specific declarations: it would be interesting if it was possible to allow domain specific declarations in addition to domain specific statements and expressions, although this was not needed for this application experiment.

References

[1] Development of a Prototype Domain-Specific Language for Monitor and Control Systems. M. Bennett, R. Borgen, K. Havelund, M. Ingham and David Wagner. IEEE Aerospace Conference, Big Sky, Montana, March 1-8, 2008.

[2] Prototyping a Domain-Specific Language for Monitor and Control Systems. M. Bennett, R. Borgen, K. Havelund, M. Ingham and David Wagner. Journal of Aerospace Computing, Information, and Communication, 2010. To appear. Extended version of [1].

[3] Constellation Program: `www.nasa.gov/exploration`.

[4] Personal communication with Martin Odersky, Jan 5, 2010.

[5] Python: `python.org`.

[6] Scala: `www.scala-lang.org`.