Experience Report: Playing the DSL Card

A Domain Specific Language for Component Configuration

Mark P. Jones

Portland State University, Portland, Oregon, USA

mpj@cs.pdx.edu

Abstract

This paper describes our experience using a functional language, Haskell, to build an embedded, domain-specific language (DSL) for component configuration in large-scale, real-time, embedded systems. Prior to the introduction of the DSL, engineers would describe the steps needed to configure a particular system in a handwritten XML document. In this paper, we outline the application domain, give a brief overview of the DSL that we developed, and provide concrete data to demonstrate its effectiveness. In particular, we show that the DSL has several significant benefits over the original, XML-based approach including reduced code size, increased modularity and scalability, and detection and prevention of common defects. For example, using the DSL, we were able to produce clear and intuitive descriptions of component configurations that were sometimes less than 1/30 of the size of the original XML.

Categories and Subject Descriptors D.3.2 [*Programming Languages*]: Language Classifications—Specialized application languages

General Terms Design, Languages

Keywords domain-specific languages, functional programming, Haskell, component configuration, Timber

1. Introduction

This paper describes some experiences from the OGI Timber project, and, in particular, from our investigation of novel programming language technology in the construction of real-time, embedded systems. The language that we developed was called Timber (Black et al. 2002) and inherited a functional core from Haskell and a computational model based on reactive objects (Nordlander et al. 2002) from O'Haskell (Nordlander 1999). One of the novel features introduced in Timber is its support for embedding high-level, declarative timing annotations in source code. In our prototype implementation, these annotations were used to drive a deadline-based scheduler, but we also intended to use them as input to a static analyzer that could provide compile-time guarantees about the real-time behavior of executable code.

Our project was just one part of a large program, involving multiple academic and industrial partners, with an emphasis on shorter-term deliverables and a commitment to conventional, off-the-shelf programming language and middleware technologies such as C++, and CORBA. Our project led to interesting research ideas and prototypes, but it was unclear if we would be able to produce a compiler for Timber before the project ended, let alone evaluate it objectively against the established tools. How then could we demonstrate the benefits of new language technologies in concrete, measurable terms that might convince even our most skeptical partners?

1.1 An Open Experimental Platform

To help address questions like these, one of the industrial partners—a group from Boeing—was tasked with producing an 'open experimental platform' (OEP) that could be shared with other teams. The OEP provided a representative library of configurable software components corresponding closely to components used in aircraft control, navigation, and tracking systems. The OEP also provided a set of configuration tools, a CORBA-based run-time infrastructure, and a collection of sample applications referred to as *scenarios*. As such, the OEP provided industrial context for individual research teams as well as a basis for cross-team collaboration.

A natural role for Timber in the OEP would have been as an implementation language for the real-time components and/or the infrastructure. Unfortunately, neither was a feasible option for us, at least in part, because of the engineering effort required to implement CORBA functionality. Moreover, proprietary functionality had been omitted from the components in the OEP leaving only code for component communication and inter-connect mechanisms. It is easy enough to model such components in Timber, but the results did not seem interesting or complete enough to enable effective comparisons or demonstrations of real-world examples.

1.2 Playing the DSL Card

The remaining part of the OEP provided tools for configuring and connecting collections of software components to build complete applications. For any given scenario, the primary input to these tools was a large, handwritten XML file describing the required components and the connections between them. As the project progressed, I began to recognize that this was an area where functional languages could be put to good use, even if it had little to do with the original vision for Timber. Instead of just trying to sell the community on 'radical' ideas for real-time functional programming, I concluded that it was time to 'play the DSL card.'

More specifically, using Haskell, I developed an embedded domain-specific language (DSL) as an alternative way of describing component configurations. I showed that DSL programs were clear and intuitive; that they were significantly smaller than the XML versions (by a factor of more than 30 in some cases); and that, unlike the original XML, they could also be composed (to support modular construction and collaborative development) and parameterized (to facilitate reuse). I produced tools that not only generated the required XML files from DSL scripts, but also generated graph-

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

ICFP'08, September 22–24, 2008, Victoria, BC, Canada.

Copyright © 2008 ACM 978-1-59593-919-7/08/09. . . $5.00

ical displays of the configurations for visual inspection. And I produced error checking tools and an automatic, reverse-engineering tool that turned their XML descriptions into corresponding DSL code. Using these tools, I discovered literally hundreds of errors (in around 50K lines of sample XML code), none of which could have occurred if the descriptions had been authored using the DSL.

While most of the participants in the program had remained unconvinced by my earlier attempts to sell them on functional programming, the DSL work and the results that I described generated quite a buzz. The Boeing team were enthusiastic about our approach, and at least one of their engineers started writing DSL (i.e., functional) programs as a result! Several other teams also expressed an interest in using the DSL, or in using similar ideas in their own projects. Unfortunately, so far as I have been able to determine, these activities did not continue after the project ended. I believe, however, that we demonstrated functional languages as a powerful 'language middleware' that supports rapid development of (useful!) domain specific languages.

Of course, embedded DSLs are widely used in the functional programming community (Hudak 1996), and so we were really just following an established strategy that had already been seen to work well in other domains. That is why I refer to this approach as 'playing the DSL card,' but it is also why I chose not to attempt to publish anything about this work at that time (or, even to publicize it outside my own group). Subsequently, however, I've come to regret that because, although the idea is well-known, it is hard to find concrete data to show that the technique has measurable benefits. Although several years have passed since the work was done, I believe that the results in this paper can help to fill that gap, and I am happy to document and share them as an experience report.

1.3 Paper Outline

In Section 2, we illustrate the role of the DSL using a simple example, and give details of our comparison between the XML and DSL descriptions of different scenarios. A brief summary of the DSL and its implementation is provided in Section 3. Finally, Section 4 describes some additional and important benefits that we obtain from the DSL. Obviously, there is not enough space to describe the target domain in full, or to explain every aspect of the model of component-based software that the DSL was designed to support. We ask that the reader may occasionally be prepared to "suspend disbelief" and accept that the details that have been omitted are not, in fact, important for the purposes of this paper!

2. A DSL for Component Configuration

During the project, the OEP developers produced a collection of 19 scenarios, each of which paired an informal description (a combination of English text and component diagrams) with a machine-readable encoding in XML. Figure 1 shows the component diagram for 'Scenario 1.1', the simplest example in the OEP. This system has three components—called `gps`, `airframe`, and `navDisplay`—that are connected in a pipeline. (The box in the top right is a comment and indicates that `gps` should be triggered at 40Hz.) A portion of the corresponding XML code is shown in Figure 2. The XML format was designed primarily to be read by tools that automatically generate 'startup' code in C++ that will instantiate, configure, and connect the appropriate set of components when the system is powered up. This creates a semantic gap between the high- and low- level views of the system that must be bridged by authors of XML descriptions. One example is that a *single* logical connection between components requires entries in *two* distinct parts of the XML file: one in the description of the sending component, and a second in the description of the receiving component.

The goal of a DSL is to capture the terminology, idioms, and notation of a specific domain, allowing domain experts to focus attention on solving problems quickly and effectively without being distracted by lower level concerns. Figure 3 shows a complete, formal description for Scenario 1.1 using our DSL. The three components

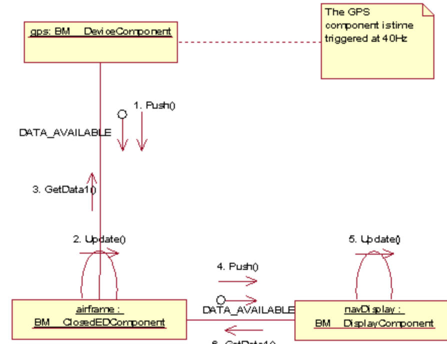

Figure 1. Component diagram for Scenario 1.1.

```
<?xml version="1.0"?>
<!DOCTYPE CONFIGURATION SYSTEM "Configuration.dtd">
<CONFIGURATION>
  <PROCESSOR>
    <NAME> OCP_P1 </NAME>
    <HOME>
      <HOME_TYPE> BM__CLOSED_ED_COMPONENT </HOME_TYPE>
      <ID>
        <NAME> BM__CLOSED_ED_COMPONENT </NAME>
      </ID>
      <COMPONENT>
        <ID>
          <NAME> AIRFRAME </NAME>
        </ID>
        ...
      </COMPONENT>
    </HOME>
    ...
  </PROCESSOR>
</CONFIGURATION>
```

Figure 2. A portion of the the XML description for Scenario 1.1. Elided parts of the source are shown here as "`...`"; the full XML source contains 130 lines, plus an additional 40 lines of comment.

```
  import OEP

  scenario
   = do processor "P1"
        airframe   <- new closedEDComp("AIRFRAME")
        gps        <- new deviceComp("GPS")
        navdisplay <- new displayComp("NAV_DISPLAY")

        trigger 40 gps

        gps      <=> airframe
        airframe <=> navdisplay
```

Figure 3. A complete DSL description of Scenario 1.1.

described previously, as well as the 40Hz trigger on the `gps` can be seen clearly here, and the two 'pipeline' connections between components are described as single logical entities, corresponding to the high-level view of the scenario. Clearly, the DSL code is much shorter than the XML version, but it still contains all of the necessary information for this scenario. In particular, our DSL toolset

includes a program, `dsl2xml`, that will generate an XML configuration file automatically from a corresponding DSL description. In programming language terms, `dsl2xml` is just a compiler translating from high-level DSL code to low-level XML. Other programs in our toolset also fit this pattern, including `dsl2dot`, which 'compiles' DSL code into graphical descriptions using the dot language of the AT&T Graphviz tools. Although we did not attempt it during the project, we could also have built a `dsl2c++` compiler to generate C++ startup code directly, without any need for XML.

The numbers in Table 1 summarize each of the OEP scenarios, listing the total number of components (#comps) and the number of lines of code (LOC) in both the (original) XML and the DSL versions[1]. This data shows that the DSL covers all of the scenarios

Scenario	#comps	XML LOC	DSL LOC	ratio
1.1	3	130	13	10
1.2	6	247	20	12.3
1.3	8	411	27	15.2
1.4	50	2,671	93	28.7
1.5	13	479	38	12.6
1.6	4	184	15	12.3
1.7	5	241	18	13.4
1.8	3	141	16	8.8
1.9	81	3993	166	24.1
1.10	15	562	46	12.2
1.11	9	426	27	15.8
1.12	9	387	27	14.3
2.1	397	23,263	754	35
3.1	4	147	20	7.3
3.2	7	258	26	9.9
3.3	17	600	49	12.2
3.4	91	4,280	213	20.1
3.5	3	129	20	6.4
4.1	572	31,796	1,038	30.6

Table 1. Line Counts for the Example Scenarios

in the OEP, and that it provides a significant reduction in lines of code in all cases, with the biggest reductions occurring in the largest examples, which is a good indicator of increased scalability.

One can argue that XML was an easy target, and it is true that XML formats are often quite verbose. In addition, the particular format that we were targeting had a fair degree of redundancy, duplicating some parts of the data multiple times and so increasing both the length of the input and the potential for introducing inconsistencies and errors. Nevertheless, many XML formats exhibit similar characteristics, so we believe that results similar to those in Table 1 could be seen in other applications.

3. Language Overview and Implementation

In this section, we summarize the DSL from the perspective of an engineer who might be using it to describe a particular system of components (Section 3.1), and then we briefly describe some aspects of its implementation (Section 3.2).

3.1 A DSL User's Perspective

Using the DSL notation, a typical scenario takes the form:

```
scenarioName = do command_1
                  ...
                  command_n
```

[1] The scenario names reflect four general categories of increasing size and complexity: (1.x) simple uniprocessor; (2.x) realistic uniprocessor; (3.x) simple multiprocessor; and (4.x) realistic multiprocessor.

The most frequently used commands are as follows:

- A `processor name` command specifies that subsequently defined components (up to the next `processor` command) should be located on the processor called `name`. This provides a way to describe distribution of components across multiple processors.
- A `comp <- new componentType("NAME")` command introduces a new component, `comp`, of the specified `componentType` with a label `"NAME"` that can be used, among other things, for debugging purposes. (This notation was, of course, chosen to echo the C++ syntax used by the OEP developers.) Components must be introduced before they are used. The DSL supports 25 different component types—covering all of the examples in the OEP—and is easily expanded to include new types.
- A `trigger freq comp` command indicates that the component `comp` should be triggered at a frequency of `freq` Hertz.
- There are three commands for connecting components:
  - `l ==> r` creates a "push" connection, indicating that `l` will generate events that should be passed on to `r`.
  - `l <=> r` creates a "push/pull" connection, indicating that `l` should notify `r` when new data becomes available, and that `r` can call back to `l` to query for specific data values.
  - `l <== r` creates a "pull" connection, indicating that `l` should be able to request data from `r`.

  Simple conventions make these notations easy to memorize: (i) The left component is (usually) the one that initiates communication; and (ii) Arrows indicate the direction(s) of data flow.
- For convenience, there are also variants of `trigger`, `==>`, and `<=>` that can handle multiple components in a single command. For example, we can specify a 20Hz trigger for three components using `trigger 20 [c1,c2,c3]`, or arrange for notifications from three event generator components to be forwarded to a single event handler component using `[c1,c2,c3] => c`.

Even with just the constructs described here, it is possible to describe most of the uniprocessor scenarios, such as the code in Figure 3. The DSL does support a few additional commands—principally to support aspects of concurrency and distribution—that, for reasons of space, are not described here.

One other significant feature of the DSL is the support that it provides for attaching "annotations" to commands using the syntax `command # annotation`. For example, the following, annotated command specifies that the events from a collection of sensors should be correlated before they are delivered to `airframe`:

```
[gps, ins, adc, radar] ==> airframe   # correl
```

Annotations override settings that are otherwise filled with (carefully chosen) defaults. The notation was chosen to resemble a comment syntax, and to allow for specification of multiple annotations where necessary, as in `command # a1 # a2`. As a result, annotations provide a lightweight mechanism for handling special cases, while otherwise avoiding clutter.

3.2 Implementation

Our DSL is implemented as an *embedded* DSL, meaning that it is, in fact, using the syntax of the host language, Haskell, together with code from a small library of functions that define the DSL constructs and operators. For example, the `do` keyword that is used in defining a scenario is, of course, just the same `do` keyword that is used in Haskell to introduce a monad comprehension. The particular monad that is used for the DSL is called `OEP` and combines elements of both a state and an environment monad:

```
newtype OEP a = OEP (Env -> State -> (a, State))
```

The environment component, Env, stores default settings that are used for creating new components, establishing connections, between components, etc. These are the settings that the annotation syntax is used to override. The State component tracks the current processor name and a list of components called a Configuration:

```
type Configuration = [Component]
```

The full definition of Component involves 16 auxiliary datatypes and runs to a total of about 60 lines. Each Component value, for example, includes details about the name and type of the component as well as details of the components to which it is connected. Individual DSL commands can modify these structures when they are executed. For example, a command c ==> d modifies the description of c to indicate that it will serve as a supplier to d and, at the same time, modifies the description of d to indicate that it will receive events from c. Of course, we could have chosen any number of alternative representations for Component. The key idea, however, is that, as we execute a sequence of commands in a scenario, we build up a data structure—essentially an abstract syntax—to represent the corresponding configuration. This is important because it avoids making a premature commitment to how the DSL constructs will be interpreted; instead, we can use any given scenario in a generic manner and then apply a specific function to the resulting Configuration to obtain a particular result. For example, the DSL library includes the following functions for generating XML code and dot graphs for a given configuration (unsurprisingly, these provide the functionality for the dsl2xml and dsl2dot tools that were described in Section 2.):

```
config2XML :: Configuration -> XML
config2dot :: Configuration -> [String]
```

Of course, it would have been possible to implement an embedded DSL like the one described in this paper using a host language other than Haskell. One of the benefits of Haskell, however, is that it allows us to provide a clean and uncluttered syntax for DSL code. For example, the DSL implementation quietly uses type classes to allow the # operator to be overloaded to deal with multiple distinct types of annotation, and to support the option to use either a single component or a list of components as the target of a trigger or the source of a ==> connector. The resulting DSL is (arguably) easier for a user to learn because they do not have to distinguish between multiple, similar names for different versions of a single concept.

4. Additional Benefits of the DSL

From the results in Table 1, we have already seen that use of the DSL leads to a significant reduction in code size in comparison to the original versions that were written in XML. This is certainly a strong positive for the DSL because, with less code to write, and less accidental complexity to deal with, there is more time for a DSL user to focus on higher level details, and there is potential for better scalability, and for increased productivity.

The DSL also made it possible, for the first time, to construct new scenarios in a modular fashion, or to build parameterized scenarios that could be quickly configured to suit the needs of a particular application. Prior to the introduction of the DSL, the XML descriptions were produced by hand, by individuals, using simple editing tools and the inevitable (but error-prone) cut-and-paste techniques. A scenario that logically consisted of two smaller and disjoint subsystems would, nevertheless, be forced into a single, monolithic XML file that tangled the two subsystems together. With the DSL, two scenarios can be developed completely independently and then combined very naturally using an expression of the form do scenario1; scenario2. The ability to support modular construction is essential in scaling to handle very large configurations developed by members of a collaborative team.

Because the DSL is embedded in Haskell, there is no need for DSL code to be limited to just the commands described in Section 3. To avoid overwhelming DSL users with too many extra details, we used this facility only lightly in our handwritten examples. It was clear, however, that there were many opportunities for programmatic construction of large and complex configurations.

Finally, but perhaps most importantly, the DSL proved to be a highly effective tool for detecting and eliminating bugs. Our first DSL examples were produced as a result of (painstaking) manual, reverse engineering of XML files. Immediately, we began to discover a surprising number and range of bugs including: ill-formed XML; inconsistent spelling of component names; incorrectly typed components; incorrect trigger frequencies; redundancy (components that had been configured to generate events, even though there were no registered listeners); etc. Many of these problems, we believe, were the result of the way in which the examples had been constructed, but almost none of them could have occurred if the XML code had been generated from the DSL. The only exceptions were specification errors (for example, where the informal text specified a 10Hz trigger while the XML specified 100Hz). Even problems like that are much easier to spot in DSL code because of the dramatic reduction in code size.

It soon became clear that we could automate the process of converting XML descriptions into DSL code, leading to the development of a new tool, xml2dsl, that also checked the input XML code for validity. The OEP evolved during the project, adding new functionality, new component types, or new scenarios in each release. Each time, with relatively little effort, we were able to update the DSL to track the OEP. The xml2dsl tool proved to be very useful in managing these updates; once we had updated the xml2dsl implementation, we could quickly generate new DSL versions for each of the scenarios without further reverse-engineering effort. (The DSL LOC figures in Table 1 reflect the size of DSL code generated using xml2dsl, but our handwritten DSL code was always of a similar size.) Of course, we reported the errors that we found in the OEP examples so that they would be fixed in the next release. Even so, because they also included new features or examples, we continued to find around 50–100 new problems in each release. Even the very last release that we studied before the project ended included no fewer than 80 components that were unnecessarily generating events without any registered listeners, as well as several more serious problems with the distribution mechanisms, in which proxy components had been incorrectly configured without corresponding masters (or vice versa). All of these problems were detected by xml2dsl, and none of them could have occurred if the configurations had been authored using the DSL.

Acknowledgments

The work reported in this paper was sponsored in part by DARPA, contract #F33615-00-C-3042, as part of the PCES program.

References

Andrew P. Black, Magnus Carlsson, Mark P. Jones, Richard Kieburtz, and Johan Nordlander. Timber: A Programming Language for Real-Time Embedded Systems. Technical Report, OGI School of Science & Engineering, April, 2002.

Paul Hudak. Building Domain-Specific Embedded Languages. In *ACM Computing Surveys*, **28A**(4), December, 1996.

Johan Nordlander. Reactive Objects and Functional Programming. Ph.D. thesis, Dept. of Computing Science, Chalmers University of Technology, Göteborg, Sweden, 1999.

Johan Nordlander, Mark Jones, Magnus Carlsson, Dick Kieburtz, and Andrew Black. Reactive Objects. In *Proceedings of the Fifth IEEE International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC 2002)*, Arlington, VA, 2002.