Automating Engineering with a Domain-Specific Language and a Code Generator *

Al Niessner Oh-Ig Kwoun Belinda Randolph Honghanh Nguyen

Jet Propulsion Laboratory, California Institute of Technology.
4800 Oak Grove
Pasadena, CA 91109

Al.Niessner@jpl.nasa.gov
Oh-Ig.Kwoun@jpl.nasa.gov
Belinda.Randolph@jpl.nasa.gov
honghanh@andrew.cmu.edu

Abstract

The following is an industry experience on using well established concepts, domain-specific language and code generation, to automate the engineering process of sub-system interactions. A domain-specific language was used to improve the communication efficiency among several teams of engineers. A code generator transforms the models formed for communicating into executables that process data. While, the savings in terms of man-effort and schedule-time were high, the single point of failure is end-user adoption. Adoption failure was attributed to two factors: One, the distance between the end-user and the realized benefit. Two, the amount the end-user had to change (become proficient in skills outside of their discipline). In our experience, the lack of an appropriate editor was the dominate cause to both factors in adoption failure.

Categories and Subject Descriptors D.3.3 [*Programming Languages*]: Specialized application languages

General Terms domain-specific modeling language, code generator

Keywords code, domain, generator, language, model, specific

1. Introduction

For more than four decades, JPL has been transforming its raw instrument data into science data through some amount of processing. The implementation of the data processing has changed over the decades, but the procedure for data processing has remained relatively constant and is shown in Figure 1 on the following page. Once the raw instrument data has been received at the processing site, it enters what is called the Science Data Systems (SDS) where it passes through various pieces of software, typically called a Product Generation Engine (PGE), to generate the various levels of Science Data Products[1] (SDP).[2] The new acronyms are samples from the language that evolved over the decades to improve the exchange of information within the SDS domain.

While the language was evolving so was the culture. Without diving into unnecessary details, the culture subdivided SDS into four primary groups of people, system engineers (SE), data engineers (DE), instrument-domain scientists (ADT), and software engineers (DST). The four groups, also known as the end-users, communicate desires, needs, and requirements through a library of word documents, presentations, and pseudo code that are peppered with inconsistent uses of the SDS-specific language that ushers in misinterpretation.

A Domain-Specific Model (DSM) (see Figure 2 on the next page) was organically developed to improve the communication efficiency and reduce effort and time of software development[1, 2]. Each of the SDP's has its own XML document describing the translation of data. Figure 1 shows each PGE generating a corresponding SDP. An XML document (DOC of Figure 2) defines the corresponding data flow within the PGE and data contained in the SDP . A Python code generator, shown in Figure 2, uses the DOC to produce C++ code that is compiled, along with a framework, to a PGE to do the actual work of transforming the data as shown in Figure 1.

* Funding for this work provided by SMAP.

[1] For Earth Science Data (ESD), the data processing levels are defined here http://science.nasa.gov/earth-science/earth-science-data/data-processing-levels-for-eosdis-data-products

[2] Many of the names and acronyms have changed over the decades but their roles and responsibilities have remained largely the same.

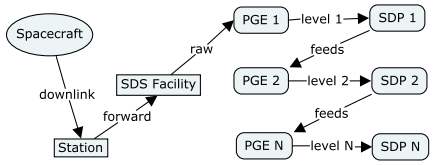


Figure 1. SDS

Simplified view of data flow from the spacecraft to the end products (SDP 1..N). The spacecraft down-links instrument data to a receiving station that then forwards it to an SDS facility. Once at the facility, the raw data is processed to level 1, then level 1 to level 2, etc to level N.

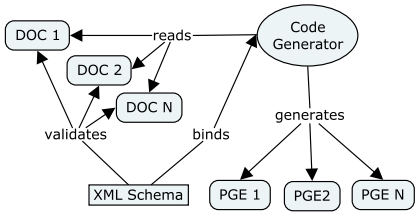


Figure 2. DSM

The DSM has multiple instances of the XML documents (DOC 1..N) that are validated with an XML schema. The XML schema is also used to generate a data binding for the code generator. The code generator produces a PGE for each XML instance. Hence DOC 1..N directly corresponds to PGE 1..N and SDP 1..N of Figure 1.

2. Language

2.1 Definition

Numerous publications provide solid advice and engineering trades for designing a DSL metalanguage[3]. However, the SDS-specific language itself was defined over 40+ years of evolution and the organic development nature of this DSM simply extracted the key parts of the language into XML tags and constraints. While misinterpretation was still a problem, with the aid of the code generator to C++ and g++, all of the misinterpretations and multiple uses of terms were quickly resolved.

2.2 Implementation

The important lesson learned about using XML for the DSL is that editing XML is the real problem. Editing was a significant enough impediment to adoption that it has an entire Section 2.3 devoted to it.

A design choice was to make the key terms XML tags and constraints and then use existing XML tools to do much of the heavy lifting for the syntax and semantic checks. Because XML schema was used as the meta-metalanguage[3], XML tools that supported validation, like xmllint, were used for syntax checking. Much of the semantics could have been encoded in the XML schema as well as constraints, but the first edition of the language required extensive help from Python for the semantic checks mostly because of the organic nature of the development. While the semantic checking with Python was less than optimal, using PyXB to validate while reading an XML document made the left over semantic checking effort low but not insignificant.

There were three lessons learned with respect to the syntax and semantic checking: One, syntax and semantic checks saved significant time during code generation. The first code generator performed poorly, to the point of being unusable, because of simple semantic errors in the XML documents. Python was used to apply semantic checks improving the code generator's performance and broadening its adoption. Two, make adding semantic checks easy because it is hard to know checks the full set of checks a priori. There were 7 semantic checks at the start, and the code generator became robust at 17 semantic checks. Three, put as many semantic checks into the XML schema as possible, but use other languages to help if necessary and move the semantic checks into the schema when mature and appropriate. The second edition of XML schema currently being developed incorporates as many of the semantic constraints into the schema as possible. Resulting XML documents demonstrate improved usability and readability from the schema constraint checks, but having semantic checks in any language or tool is more important than delaying their existence and use while they wait to be put in the XML schema.

2.3 Editing

Editing the XML documents (a complete example is contained in Appendix A) is the complete downfall to the DSL using XML because the end user wants to edit the content of the XML and not the XML itself. The XML technology was a good choice from the perspectives of the off-the-shelf tools, the size of user community, the size, variety, and depth of the support community, and the shallow, long learning curve[4]. However, the tag nature of XML raises the noise floor for the end-user that only cares about the content and that is formatted to express internal relationships. In this specific instance, the average XML document is 4000 tags and the end user wants to view subsets of information within these tags in various formats like tables, ordered graphs, etc.

The first attempt to improve editing was to add XSL transforms for viewing the XML documents within a browser in the format the end-user wanted but using a text editor to change the XML document as shown in Figure 3 on page 5.

[3] While maybe not the best engineering choice, using XML schema with XML is not necessary.

[4] In other words, one does not have to digest the entire technology all at once in order to use it, but, rather, can learn small portions of it over long periods of time while using it.

Quick cycles between random edits and what they could view in a browser was sufficient at the beginning. As more people joined the team and the project progressed requiring more information be contained within the XML documents, the XSL transforms fell apart and the end-users became more vocal about the difficulty of using a text editor.

The second attempt was to customize an XML editor to combine the views developed with XSL and direct editing. Light effort at customization with just a few XML editors quickly led to the conclusion that we wanted to edit the content not the XML. The customization of the XML editors was more about displaying various hierarchies of the tags and not building an editable ordered graph using the tags and its attributes to define the nodes and edges and their content to become the labels[5]. The end-users found no value in this approach.

The third attempt was to hire visualization specialists to fill in the knowledge gap of the DSL development team to help produce an editor with end-user desired views. The visualization team wanted to use a browser-cloud approach because it was how they did all of their work. With hindsight, it would have been better to have them play a bit with customizing the XML editors, but the budget was tight and an editor was desperately needed. The budgetary pressure aided in the requirement degeneration[6] to an XML editor.

The fourth attempt, and the one currently slated for deployment while this paper is being published, was also done by visualization experts. However, the focus was kept on content editing[7] (see Figures 4 through 6 on page 5) rather than allowing the requirements devolve, despite budgetary and schedule pressures, to another XML editor. The results of this editor are unknown, but preliminary demonstrations to the end-users were very positive unlike the previous attempts.

The ideal editor would have a DSM, like graphml, that allows a team to generate a content editor within the meta-metalanguage. The fourth attempt at an editor is specific to this DSL, but it is easy to see how to generalize it and use the <xs:annotation> in the DSL's XML schema to pass information to the editor for layout and relationships of content. A DSM tool aimed at generating content editors would be the biggest boon to the quick development of a DSM[8].

[5] If they are capable of such customization, then learning about it was beyond the given budgetary constraints.

[6] Degeneration because every software engineer that looked at the XML document saw XML. In turn, the requirement to edit the content degenerated to mean the XML tag in an XML hierarchy and not the content within the tags in a more context-intuitive composition and layout.

[7] LyX is the best example of a content editor.

[8] Attempts to use UML[4, 5] or Eclipse[6] did not succeed for the same reasons as using XML.

3. Code Generator

3.1 Definition

Since the DST had already defined most of the external tools and libraries they required and wrapped them with a framework, the code generator was very straight forward. Reading through the APIs and perhaps experimenting with some oddities or corner cases was all that was needed. The generated C++, FORTRAN 2003, and XML code connects the ADT developed science algorithm code to the DST supplied C++ framework.

3.2 Implementation

The code generator was implemented in Python using PyXB library for parsing, validating, and data binding. The code generator first validated the XML documents with syntax and semantic checks as described in the DSL in Section 2.2 on the preceding page. Conversion of objects from the PyXB to C++ code resulted in several thousand lines of C++ for each PGE. While it is very difficult to precisely measure the manpower saved, a conservative first-order estimate indicates 5 man-years of savings for a one man-year effort for both the DSL and code generator.

3.3 Testing

Testing of the generated code added some rigor to the code generator itself. The first level of testing were the compilers. The compilers highlighted misinterpretations of the DSL from context sensitive definitions[9] or word misuse, which resulted in improvements in the XML schema, semantic checks, and code generator. The second level was static analysis tools like Coverity. With a false positive ratio below 5%, the static analysis tools highlighted nearly 100 poor code generator choices that were quickly remedied. The third level of testing was a stub from the code generator that allowed the software team to run the generated code without the addition of the ADT science algorithms exercising all of the generated code. Since the generated code was exercised in isolation, run-time problems found in the PGE after ADT science algorithm integration had to be the purview of ADT and not the code generator.

While it would be nice to say that the DSM experiment presented here was done formally and with a control, it be wholly misleading to do so. However, there was one team that shunned the DSM approach resulting in an experimental control group. In order to call them a control group, they use the same framework and external libraries as the DSM group and the code developed versus generated do the same task. Both the code by the control group and that by the code generator used the same compiler and compiler flags achieving the same level of code quality or defect density. Coverity was used to statically analyze both the control group's code and the generated code, but both groups did

[9] Some words had different meanings depending on which team was using them.

not use the results the same. In the control group, only one team member of six used Coverity result to fix problems that had been identified. In contrast, the code generator was modified to drive the problems to zero. The control group was responsible for 6 PGEs while the code generator was 17 PGEs. Lastly, running the external libraries, framework, and generator stub allowed the generated code to pass another level of testing that the control group did not have. The end result was fewer delivered bugs, faster repairs, and less frantic deliveries as compared to the control group.

4. Conclusion

There are two reasons for the high return on investment: One, the language was predefined from four decades or more of repetition reduced the investment cost. Two, the code quality cycle saves effort that scales with N because any problem detected in one PGE would be fixed in the code generated resulting in all PGEs being fixed.

The primary obstacle to adoption was the ability of the end-user to easily design, edit, and view their models. General modeling languages, editors, and tools, such as UML and SysML, are outside the domain of expertise for most of the SDS end-user obscuring their benefits and relevance. Presenting the model in an end-user specified layout increased adoption. Find or build a content editor as soon as possible even though it significantly increases the investment because, unless the end-user realizes immediate benefit from the DSL, they will resist any changes from the status quo.

Acknowledgments

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

We would like to thank all those that helped us improve the tools for wider adoption with special thanks to those that suffered the tools in their infancy.

References

[1] Christopher Preschern, Andrea Leitner, and Christian Kreiner. Domain-Specific Language Architecture for Automation Systems: An Industrial Case Study. *European Conference on Modelling Foundations and Applications Graphical Modeling Language Development Workshop,* 2012.

[2] Steven Kelly and Risto Pohjonen. Worst Practices for Domain-Specific Modeling. *IEEE Software*, vol26, no. 4, pp 22-29, July/Aug, 2009.

[3] Janne Luoma, Steven Kelly, Juha-Pekka Tolvanen Defining Domain-Specific Modeling Languages: Collected Experiences. *OOPSLA Workshop on DSM*, 2004.

[4] Juha-Pekka Tolvanen. Industrial Experiences on Using DSLs in Embedded Software Development. *Proceedings of Embedded Software Engineering Kongress (Tagungsband)*, Dec 2011.

[5] Mark Dalgamo and Matthew Fowler. UML vs. Domain-Specific Languages. *Methods and Tools*, 16(2):2 - 8, 2008.

[6] Steven Kelly. Tools for Domain-Specific Modeling. *Dr Dobb's Journal*, Spetember 2004.

A. Example XML Document

See Algorithm on page 6.

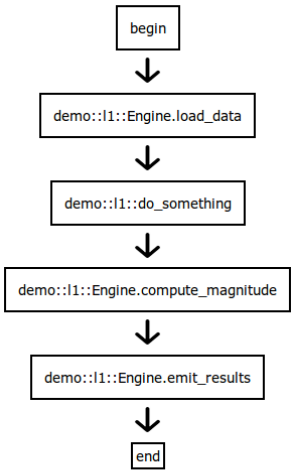
Product Name: LevelOne

Table of Contents:

Comments

```
      The code interface is quite simple since it is a demo of what the
code generator can do and how the pge.py works.

      First, we load the data with  Engine.load_data(). The load reads
from several sources and collates them for later use. It stores them as
state in the derivative of Engine for the other methods to use. While PIX
allows parameters to be passed, sometimes hiding the information in the
state of the class seems more appropriate with C++.

      Second, we do something just to illustrate that all methods do not
need to be bound to the Engine class.

      Third, we are back to operating on the data by computing the 2-D
magnitude of the 1-D samples by multiplying them. It shouls make a fun
surface, but is not overly complex since doing math is not the item of
interest.

      Lastly, we take the results back out of the Engine class and put
them in a container for the rest of the world.
```

Interface



Interface Tree

Product

Groups:

- /results

/results

Element	Shape	Concept	Bytes	Signed	Unit	Min	Max	Comment
x_axis	SpatialArray	real	8	NA	mm	0.0	10.0	
y_axis	SpatialArray	real	8	NA	mm	0.0	10.0	
magnitude	XaxisYaxisArray	real	8	NA	eV	-1.0	1.0	

Back to Table list

Shapes:

0 mean Extensible

Name	Dimensions	Index Ordering
SpatialArray	Spatial= 0,	irrelevant
XaxisYaxisArray	Xaxis= 0, Yaxis= 0,	slowest...fastest

Figure 3. XSL View of Content
The view of the same information within a text editor would be Appendix A.

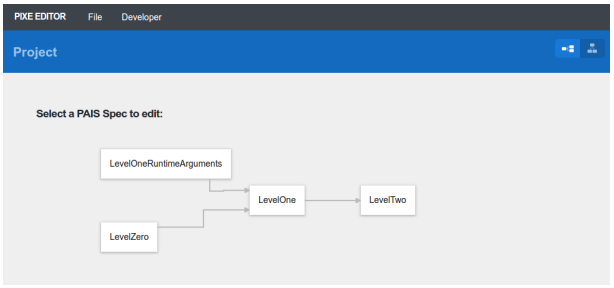


Figure 4. Content Editor Front-Page
The directed graph is based on the <method> content (see Appendix A) of the XML documents.

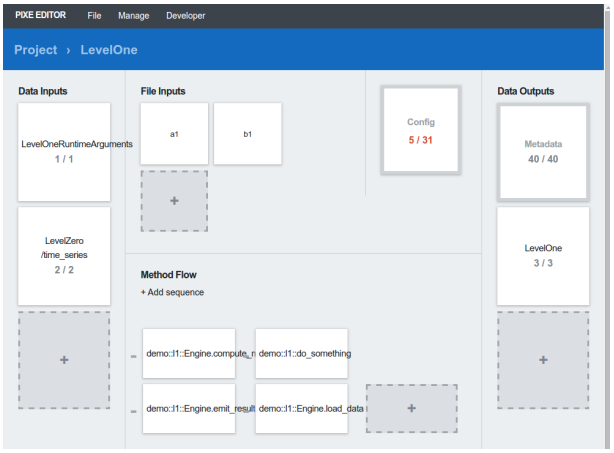


Figure 5. Content Editor Interface-Page
Contains a directed graph based on <method> content and then encapsulates larger <product> concepts (see Appendix A) in the DSL into active boxes that expand to different views when selected.

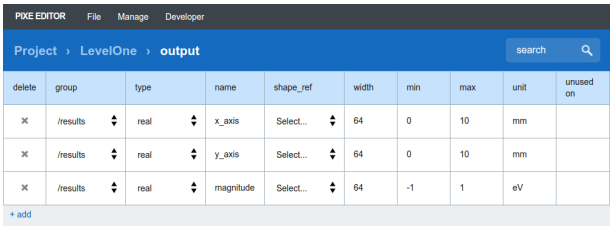


Figure 6. Content Editor Advanced-Page
Represents the <nodes> content (see Appendix A) as table.

Algorithm 1 Complete Example

```
<?xml version="1.0" ?>
<algorithm name="LevelOne"
           xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
           xsi:noNamespaceSchemaLocation="http://smap-sds-web.jpl.nasa.gov/schema/pix.xsd">
  <interface language="C++">
    <annotation app="code">
The code interface is quite simple since it is a demo of what the code generator can do and how
the pge.py works.
First, we load the data with Engine.load_data(). The load reads from several sources and
collates them for later use. It stores them as state in the derivative of Engine for the other
methods to use. While PIX allows parameters to be passed, sometimes hiding the information in
the state of the class seems more appropriate with C++.
Second, we do something just to illustrate that all methods do not need to be bound to the
Engine class.
Third, we are back to operating on the data by computing the 2-D magnitude of the 1-D samples
by multiplying them. It shouls make a fun surface, but is not overly complex since doing math is
not the item of interest.
Lastly, we take the results back out of the Engine class and put them in a container for the
rest of the world.
    </annotation>
    <flow>
      <sequence>
        <call>demo::l1::Engine.load_data</call>
        <call>demo::l1::do_something</call>
        <call>demo::l1::Engine.compute_magnitude</call>
        <call>demo::l1::Engine.emit_results</call>
      </sequence>
    </flow>
    <method name="demo::l1::Engine.compute_magnitude"/>
    <method name="demo::l1::do_something"/>
    <method name="demo::l1::Engine.emit_results">
      <output name="mag" node="/results/magnitude"/>
      <output name="x" node="/results/x_axis"/>
      <output name="y" node="/results/y_axis"/>
    </method>
    <method name="demo::l1::Engine.load_data">
      <annotation app="code">
Load the data from the level 0 product and the ancillary file.
The ancillary file is a simple binary that is simply two float
values that both conversions from V to eV and scale to the sensor.
The data in level 0 is defined by the container.
      </annotation>
      <input name="ifn" node="/ancillary_file_name"
             prod="LevelOneRuntimeArguments"/>
      <input name="x" node="/time_series/x" prod="LevelZero"/>
      <input name="y" node="/time_series/y" prod="LevelZero"/>
    </method>
  </interface>
  <product>
    <nodes>
      <real max="10.0" min="0.0" name="/results/x_axis"
            shape="SpatialArray" units="mm" width="64"/>
      <real max="10.0" min="0.0" name="/results/y_axis"
            shape="SpatialArray" units="mm" width="64"/>
      <real max="1.0" min="-1.0" name="/results/magnitude"
            shape="XaxisYaxisArray" units="eV" width="64"/>
    </nodes>
    <shape name="SpatialArray" order="irrelevant">
      <dimension name="Spatial" extent="0"/>
```