Nodular, Tcl and Software Engineering

Johann Tienhaara
MountainGrouse Software
tech@mountaingrouse.org

Abstract

Nodular is a data modelling library for Tcl. It is coded in pure Tcl and is freely available and distributable under the BSD license.

In a nutshell, Nodular is useful for transforming Unified Modelling Language (UML) [6] design documents into code, and reverse-engineering code into UML design documents. In fact, Nodular is much more widely applicable, but the bulk of this paper is dedicated to discussing Nodular with these aims in mind.

At the time of writing this paper, the process of re-implementing Nodular from the ground up (as version 1.0) has recently begun. The previous version (version 0.8) is useful for modelling potentially large data structures, but is fatally flawed at translating between modelling formats. The memory requirements for even small translation tasks are exorbitant. The new version is much slimmer and more versatile.

This paper briefly outlines Nodular as well as my experiences in developing it. The article is comprised of 4 sections:

1. *Roundtrip Software Engineering and Tcl*: Why Tcl is the best programming language for roundtrip software engineering tools.
2. *Nodular*: Why Tcl needs Nodular for roundtrip software engineering tools.
3. *Under the Hood*: An overview of the structure of Nodular.
4. *Implementation Experiences*: What went wrong, what went right, what other Tcl insights came out of the process?

1 Roundtrip Software Engineering and Tcl

1.1 Translating Data: Experiences With Tcl

Any time a programming language is called for to translate between data formats, I recommend Tcl. This approach is the pup of that big, brutish, nasty dog called Experience. When, during a middleware project, I was faced with translating an XML document (encoded in UTF-8) into an SAP IDOC (encoded in SJIS or Big-5 or a variety of other Asian language-friendly encodings), I searched for solutions using the programming languages already employed: a proprietary LISP-based language and Java. In the summer of 2001, the proprietary LISP-based language at the core of the middleware system did not support the Asian encodings, and support was "planned for Q2 next year". (Ha! Sure.) Java also did not support anything other than Unicode, UTF-8 and a handful of other encodings (and its API for handling these encodings was awkward at best). There were complex, expensive C++ libraries available for the Solaris platform, but I did not have easy access to a C++ compiler.

So I downloaded TclPro 1.4 for Solaris, wrote 5 lines of code, and presto! SJIS and Big-5 IDOCs.

1.2 Throw Out Your JDK

(Or: Why Tcl is the Best Programming Language for Roundtrip Software Engineering Tools. Ever.)

The ease of reading from and writing to data sources with different encodings in Tcl already makes it an excellent starting point for developing roundtrip software engineering tools. Add to that the regular expression libraries, the straightforward support for different data sources (files, sockets, databases, XML DOM trees, and so on), cross-platform availability, and the very fact that Tcl is an easily-extensible scripting language, and Tcl meets most of the requirements for an environment capable of housing flexible, extensible roundtrip software engineering tools.

For these reasons, translating between file formats (for example, from a Dia [4] UML diagram to [incr Tcl] [5] code and a SQL script) and interchanging model files (for example, from a Rational Rose model file to a Dia file to an XMI file) is much more feasible in Tcl than in C, C++, D*b* / .NET, Java, Prolog, Basic, Awk, Sed, and so on.

What, then, is missing?

2 Nodular

2.1 The Missing Link: Nodular

Nodular provides the layer that is missing in Tcl to perform complex translations and transformations between data models.

The Nodular core provides a library for constructing graphs in Tcl. The Nodular nodes are "functional", so each node can be tailored to behave in a specific way to follow the rules of its particular data model. Nodular is thus flexible enough to model just about any kind of data, and can easily be extended or customized by software engineers with specific requirements.

2.2 Nodules: What You Can (and Can't) Do With Nodular

The Nodular distribution includes much more than the core of Nodular, though. A number of "nodules" -- essentially packages of node types, edges and graphs -- come bundled with the core. Table 1 shows the features which are fully or partially implemented in Nodular versions 0.8 and 1.0.

Nodule	**Summary**	**Nodular v0.8**	**Nodular v1.0**
nodLib	The “Nodular library”:	N/A	Fully implemented.

	automatic parser and compiler nodes, data type nodes, and useful constraints.		
nodEBNF	Backus-Naur Form grammar nodes and file parsing.	Partially implemented.	Fully implemented.
nodXPath	Nodes used to parse XPath strings for indexing nodXML nodes or arbitrary Nodular nodes.	Informally prototyped.	Not yet implemented.
nodUML	UML data model nodes.	Class diagrams implemented.	Under development.
nodDia	Dia nodes, file parsing and Dia-to-UML transformations.	Class diagram transformations implemented.	Not yet implemented.
nodXML	XML data model nodes and XML file parsing.	Fully implemented.	Not yet implemented.
nodSQL	SQL data model nodes and UML-to-SQL transformations.	Basic CREATE TABLE statements implemented.	Not yet implemented.
nodXMLSchema	XML schema nodes.	Fully implemented.	Not yet implemented.
nodJava	Java data model and J2SDK, J2EE nodes, UML-to-Java transformations.	Partially implemented.	Not yet implemented.

Table 1. Nodule matrix for Nodular versions 0.8 and 1.0.

3 Under the Hood

3.1 What is Nodular?

The core of Nodular is a library of graph routines and data structures. The Nodular core provides procedures to create labelled nodes (or "vertices") and connect them to each other with doubly-labelled edges (or "arcs").

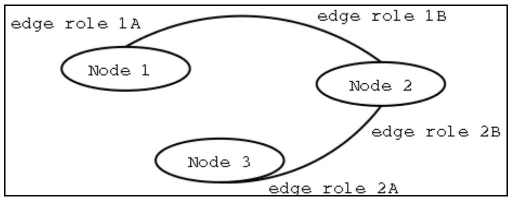

Figure 1. Nodes and edges in Nodular.

Nodular is essentially a directed graph (or "digraph")[1], with a few distinctions that are covered in sections 3.3 ("Nodular Graphs") and 3.4 ("Functional Nodes").

3.2 Graphs

The nodes in a directed graph can be used to represent complex data structures, and the edges can be used to represent the relationships between data structures. However, in order for the digraph to be a useful medium for representing data models, two elements must be added to it:

1) a label describing each node's role in a particular edge; and
2) the ability to connect a node to itself along an edge.

For the purposes of this discussion I will refer to a directed graph that includes these adjustments as an "augmented directed graph" or an "augmented digraph".

For example, the structure of the XML data model might be represented as the following augmented directed graph:

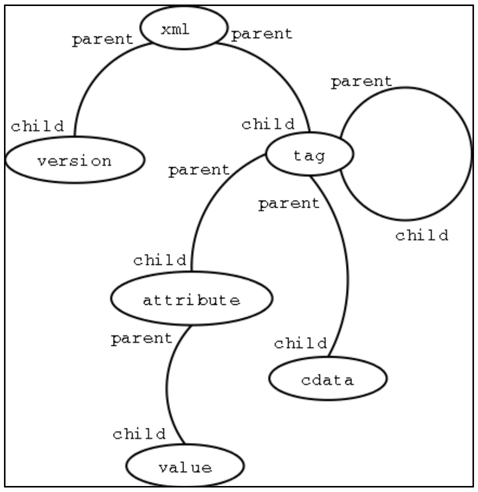

Figure 2. A Nodular graph representing the XML data model.

Then a particular XML document might be represented as a number of nodes that are connected to the previous augmented digraph by (class / instance) edges. See Figure 3 for the graph of the following XML data:

```
<?xml version=1.0?>
<foo bar="hello">
  <foo2>
    world!
  </foo2>
</foo>
```

[1] For a dictionary of graph theory terminology, the following website is a good resource:
http://www.utm.edu/departments/math/graph/glossary.html.
There is also an interesting exploration of graph theory in Tcl available at:
http://mini.net/tcl/2473.

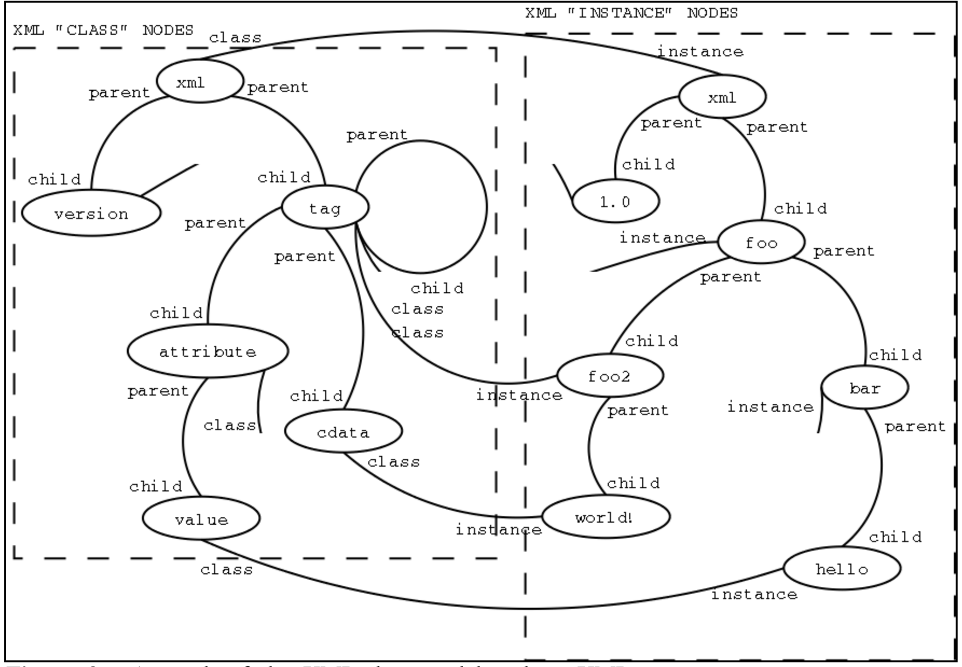

Figure 3. A graph of the XML data model and an XML document.

Why would anyone want to represent nice, clean XML data with this mess of nodes and edges?

1) An augmented directed graph can be used to model anything;
2) an augmented digraph is easily created by a machine; and
3) an augmented digraph is easily interpreted by a machine.

Therefore an augmented digraph is useful in areas such as enterprise application integration (EAI), which involves taking data from one data source, transforming it, and storing it in another data source. (For example, taking sales information from a web server as an XML document, transforming it to an SAP IDOC to begin sales processing, and then transforming it to an SAP IDOC to begin sales order processing, and then transforming it to a set of SQL inserts and updates to store the transaction in a data warehouse.) Enterprise application integration is one domain that could benefit from Tcl and Nodular.

However, enterprise application integration is not the focus of this paper. Software engineering is. But some of the automatable tasks in software engineering -- such as automatically generating skeleton or prototype code from detailed design documents, or generating test suites from requirements documents -- use the same principles of transforming data that EAI uses.

3.3 Nodular Graphs

I have claimed that an augmented directed graph is an excellent machine-understandable format for representing complex data models. What, then, is so special about a Nodular graph? Presumably any library of digraph nodes and edges and routines can be easily tweaked to be useful for data modelling. There are certainly many such libraries available with far more comprehensive search algorithms than Nodular.

3.4 Functional Nodes

Nodular extends the basic nodes of graph theory by making them "functional". That is, a Nodular node is essentially a function which either evaluates to itself or to some other node; and its edges are dynamically determined by a function. The functional nature of Nodular nodes facilitates representing pointers, foreign keys, inheritance relationships, and so on.

These minor changes to an augmented directed graph infuse Nodular with the flexibility and power required to model a wide variety of data formats.

4 Implementation Experiences

4.1 Early Experiences

The immediate predecessor to Nodular was an XML schema-to-Dia UML class diagram transformer [7]. Written in the fall of 2001, xsd2dia is simple, limited and ugly, because it was a quick hack. Nevertheless, a few noteworthy insights came out of the exercise:

1) Tcl has no library to support the gzip [1] format (which is used by the Dia diagramming tool). This would be an invaluable addition to an extension such as TclLib.
2) Automatically determining reasonable positions for elements in a structured diagram can be achieved in numerous ways, with varying degrees of success depending on the particular shapes and relations being diagrammed. A Tcl tool that supports multiple methods of graphical placement would be valuable for both batch processing applications (such as generating Dia diagrams) and Tk diagramming using the canvas widget. (Nothing as complicated as graphviz [3] is called for here -- just some simple algorithms to minimize the number of crossed edges between shapes in a diagram.)

4.2 Nodular Version 0.8 Experiences

I have successfully used the first implementation of Nodular (version 0.8) for two tasks:

1) converting complex XML schemas into [incr Tcl] code (for commercial software); and
2) converting small UML class diagrams (in Dia) to code (Java and SQL in particular).

I had hoped to progress further with the second task by transforming several large UML models depicting a system into code and data. But when a program running on a machine with 128 megabytes of memory requires over half a gigabyte of RAM to transform a scaled-down system model into code, progress on the task is impossible.

In retrospect, creating a separate Tcl namespace for each node, with its own procedures and data, was not wise. Especially when the number of nodes in the graph swells to over 10,000!

4.3 Nodular Version 1.0: the Bleeding Edge

The current version of Nodular, being developed from the ground up, uses a similar approach to [incr Tcl] and TclDOM [2] for naming nodes. Each Nodular node has an identifier which is used as an index into a table where its label and functions are stored. This approach is memory efficient (in contrast with the unwieldy approach taken for Nodular 0.8), and prompts the question: would a library to provide this support for structural data in Tcl be widely useful to application developers? For example, a "struct" namespace containing procedures which:

- maintain a counter for each type of structure being generated;
- generate a name (comprised of the structure type and the counter for the type -- for example, "myclass0", "node57", "astruct999", and so on);
- store and retrieve data for each instance of a structure, indexed by structure variable names (for example, to facilitate retrieval of the "edges" variable of structure instance "node73").

Nodular version 1.0 is still in the early stages of development, but the memory footprint is infinitely smaller than its predecessor (version 0.8). Accessing 100,000 nodes in Nodular version 1.0 requires about 60 megabytes of RAM.

Nevertheless, Nodular is slow.

This is partly due to its very nature -- a structure so simple it can be used as the atomic layer to build up complex structures -- but also largely due to implementation decisions. For starters, no caching has been implemented. This means that a complicated *getEdges* function might be evaluated repeatedly during a long operation. This inefficiency becomes apparent when, for example, a very simple EBNF grammar takes two minutes to parse using the Nodular EBNF parser. A complex grammar might take hours.

Building a nodule to represent a complex data model is also a time-consuming process. Usually there are numerous approaches to translating a data model's structure into a Nodular graph. Each approach has advantages and disadvantages and must be considered carefully before and during implementation. With experience, patterns of good nodule-building technique and standard procedures may emerge to reduce the complexity of designing a nodule.

The task of building the graph to represent a data model is also very labour-intensive. This procedure begs for a simple macro-based graphical user interface (written in Tk, of course) to reduce the time required to build a nodular graph.

However, the process of transforming one data model into another (once each data model has been encoded as a Nodular graph) is easy as pie. The transformation scripts themselves can be graph-based, so visual editing of transformation scripts is also possible. This should render the task of tweaking the transformation scripts fairly easy for Nodular (or even Tcl) neophytes.

Flaws and all, Nodular gives Tcl quite a potent foundation for roundtrip software engineering tasks. Runtime and learning-curve efficiencies will likely be concerns for future versions of Nodular.

For more information on the Nodular project, visit:

`http://www.sourceforge.net/projects/nodular`

Or contact:

Johann Tienhaara
`tech@mountaingrouse.org`

References

[1] Adler, Mark and Jean-loup Gailly. The gzip home page. http://www.gzip.org/
[2] Ball, Steve, et al. Chapter 1. TclDOM. http://tclxml.sourceforge.net/tcldom.html
[3] Ellson, John, et al. Graphviz. http://www.research.att.com/sw/tools/graphviz/
[4] Larsson, Alexander, et al. Dia a drawing program. http://www.lysator.liu.se/~alla/dia/
[5] McLennan, Michael. [incr Tcl] – Object-Oriented Programming in Tcl/Tk. http://incrtcl.sourceforge.net/itcl/
[6] Object Management Group. UML Home Page. http://www.uml.org/
[7] Tienhaara, Johann. Xsd2dia. http://www.mountaingrouse.org/xsd2dia.html