⊔: The Universal Short Graph Language

Laurent Fournier – lfournie@rockwellcollins.com

version: 0.2 [JCstC]*

January 4, 2012

Abstract

Model Based Engineering widelly use two dimensions graph-based diagrams. Because these diagrams represent viewpoints of a system, including dataflow, workflow and software architecture, they are the building blocks artifacts for specification, modeling and simulation activities. In particular code generation from this high level representation is mandatory. The data format for these diagrams may be graphical; bitmap or vectorial, unfortunatelly mixing rendering/layout data with semantics. XML based formats are also used like XMI for MOF/UML. However, those formats suffering of several drawbacks like unreadability, unsusefull verbosity and not well adapted structure for representing graphs. HUTL and JSON are not used. The *Graphviz* DOT language or the simple YUML syntax have nice features but lacks to provide native typing and nesting. Our proposal in this paper is a typed graph dedicated language called , provinding the very minimal syntax for graphic and code generation. The language is mainly defined by one given Regular Expression. We present a non formal interpretation of the language, with examples from various models like UML, *SysML*, *Marte*, AADL, *Xcos*, *Kaos*, Entity-Relation Graph, Tree Diagram, Network graph, Flowchart, Petri-Net, State Machine, Markov Chain, Behavior Tree, Flow-based programming diagram,... This language is a universal representation for input of multi-model code generator tools. We provide as a proof of principe some simple example of code generation for C, Ada, Python, Java, Ocaml, Ruby and Scala Coding Languages. Generation can produce textual representation for AADL,SDL and Lustre (Scade) and rely on their own chain to generate code. Graphic generation is also supported for Tikz to documents and SVG for web viewer/editor. Actually, we are introducing the concept of *differential dual editing*; where textual and graphical editing are well supported by our language. All source code for a prototype parser and code generators, document generator in *Python* is attached to this PDF file.

1 A Short Language for Graphs

A graph is represented as a couple (V, E) of arrays of Nodes/Vertices and Edges/Arcs. Each edge references source node(s) and destination node(s). More exactly the connexion point of a node is a *port*). Both edges and nodes are objects having an attribute list defined in a type (a class). A node can be nested, thus including another graphs. The curly brackets are delimiters for nesting. A particular attribute is named *label* and delimited with quotes (simple, double or string of three double quotes for multilines labels). This label is a free string displayed on the node/edge.

For a given edge of type T, there is nine differents links represented as:

undirected	`-T-`
simple right	`-T>`
simple left	`<T-`
bidirectional	`<T>`
full right	`>T>`
full left	`<T<`
opposite	`>T<`
source right	`-T<`
source left	`>T-`

Nodes have a name identifier that is a regular word of the *unicode* encoding. There is no need to give an identifier for edges since they are defined by a couple of (source,destination) nodes ports. Naming nodes allows to reference them easilly all node links without repeating all the nodes attributes. The edge/node type can also be named or implicit and it can support some constructor parameters inside parentheses and comma separator. Edge type name can be just one unicode character long to make the link not too long between the head and

*the first five characters of the base64 encoding of the SHA1 digest of `u.py` source file. Please compare it with the one published at `https://github/pelinquin/u` to check or get the last release.

the tail using characters [– <>]. It is also a good practice to use foreign languages characters to reference types so they can be distinguished from node's name (instances). Nodes have ports so edges can reference them directly. This is important for nodes types where the connected point of the node and edge has semantic meaning, because these ports are named or because ports position is important (top, bottom, right,left).
The USGL language does not include any geometric properties for the graph layout. The shape/color of each nodes/edge type is defined elsewhere. Also the graph layout may be defined by set of algorithms for improving rendering and limiting edge crossing or set manually but all geometric data is never usefull for our language. We argue that our small language is complete enought for Model to Code generation.

2 Ready for code generation

To generate code from an intance one must first define a semantic mapping between the node/edge type features and coding constructions. This mapping can be based on graphical patterns related to code patterns. For instance, we must state that the `-I>` relation represents inheritance for a UML class diagram.

3 Type checking

Introduce here how to constraint nodes type with edge type...Do we need an OCL like language for that? Can we propose a language for pre/post conditions and invariant checking?

4 Differential dual editing

The basic idea is that the editors tools enable both graphical and textual editing at the very same time with minimum toogleling between the graphic and the textual mode. Graphic updates are made by difference. Exactly as a text editor insert ou remove a character in a text flow without processing all the document, the corresponding modified graphic nodes or edges are updated without processing all other nodes, edges. A background task may compute the best layout for the graph at a some change level. Because has a very simple syntax close to the graphic, all atomic operation has a direct equivalence in text mode or in graphic mode. There is no need to store or share the image dump of the graph except for printing. The file can be shared between modelers, editor tools and code generators and remaining short, human readable and meaningfull for rendering a graph.

5 Parsing

Comments (`#...`) are first removed from code before parsing. The Parser resulting structure is not exactly the regular expression group. Nodes and egdes are tuple:

$$\begin{aligned} Node: &\quad (Name, Type, Label, Arguments, Children) \\ Edge: &\quad (Arrow, Type, Label, Arguments, sourceNodes, sourcePorts, destNodes, destPorts) \end{aligned}$$

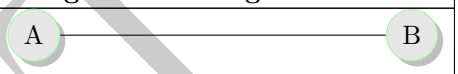
The parser replaces the raw block ($G6$) by the references of inner nodes. $G2$ is not stored in the node but in the edge using it. ($G5$) and ($G10$) arguments strings are precomputed as arrays. For edges, $G7$ and $G11$ are concatenated.

```
__RE_U__ = r'''                    # RegExp with 10 groups
    (?:                            # Token is NODE:
     (?=(?:[^\W\d_]|:[^\W\d_]|[\"\']))  #  check not empty
     (?:([^\W\d_]\w*)|)            #  Name       G1
     (?:[\"\']([^\"]*)[\"\'])?     #  Label      G2
     (?:\.(\w+))?                  #  Port       G3
     (?::([^\W\d_]\w*)|)           #  Type       G4
     (?:\(([^\)]*)\))?             #  Arguments  G5
    )|(?:                          # Or EDGE:
     ([\-=<>])                     #  Head       G6
     (?:\"([^\"]*)\")?             #  Label      G7
     (?:(\w)|)                     #  Type       G8
     #(?:([^\W\d_]\w*)|)           #  Type       G8
     #([^\W\d_a-zA-Z])?            #  Type       G8
```

```
    (?:\(([^\)]*)\))?                    #  Arguments  G9
    ([\-=<>])                            #  Tail       G10
  )
'''
```

6 The Test Set

The test set is an array of cases. Cases have a name, an code sample and the expected *Python* structure as a (V, E) couple of nodes and edges. The parser use that test set to check against expected data. From the Nodes and Edges arrays, one can either generate vector graphics (TikZ in this table but could also be SVG on a web server) or generate source code. Each node and edge type define a particular node/edge shape for graphic or a particular construction in a programming language. Generating graphic code requests also a layout algorithm for placing nodes and edges. Because layout never carry graph semantics, we can run an automatic algorithm with simple lisibility criteria like balacing nodes on the page or avoiding edge crossing.

Case	Test name	U code	TikZ generated diagram
01	Simple	`#simple dot diagram` `A->B`	

References

[1] Leslie Lamport *LaTeX: A Document Preparation System*. Addison Wesley, Massachusetts, 2nd Edition, 1994.

[2] XXX *TikZ*.

[3] OMG *XMI*.

[4] OMG *MOF*.

[5] G. Booch, J. Rumbaugh, and I. Jacobson *The Unified Modelling Language User Guide*. Addison-Wesley,1999.

[6] OMG *Human Usable Textual Notation*. 2005.

[7] Emden R. Gansner. *The DOT language*. 2002.

[8] Tobin Harris *yUML* yuml.me 2002.

[9] XXX *SVG*.

[10] XXX *TGF*.

[11] XXX *graphML*.

[12] XXX *GXL*.

[13] XXX *AADL*.

[14] XXX *Lustre*.

[15] XXX *SDL*.

The end of the document