The ⊔ Language

Laurent Fournier*

*lfournie@rockwellcollins.com

December 7th 2011

What ⊔ is?

The ⊔ language is a **Universal Graph Language**;

- Symbol: $\bigsqcup$
- Name: “square cup”
- Universality (close to “u”)
- Unicode character ⊔: (U+2294)
- License: GPL v3

Graph

A graph: $G = (V, E)$
where:

$$V = \{v_i\} \text{and} E = \{e_k\}$$

a set of nodes (vertices) and a set of edges between nodes.

$$e_k = (\{v_{i_p}\}, \{v_{j_q}\})$$

an edge links an origin nodes set to some destination nodes set.
p and q are ports references.
$|i| > 1$ or $|j| > 1$ for multi-links.
Each node v_i or edge e_k is a key attached to some attributes list.

⊔ features extscthous

- Typed
- Hierachical
- Online
- Unicode
- Short

⊔ at a glance

`Hello -> World`	Hello World!
`A B C`	Nodes
`A->B C<-D`	Links
`A{B C}`	Composition
`A->{B C}`	Mulilinks
`X"A Node label"`	Labels
`A:T B:Class`	Typed nodes
`A(2 4) B(f 6 8)`	Arguments
`A <Y> B`	Typed arc
`A"name":T(arg) A->B A<-C`	Id reuse
`A.pin2 -> B.5`	Ports

The main principles

Structure

⊔ only manage the structure of the graph, not the semantics of Nodes and Edges. The ⊔ parser builds an Abstract Syntax Tree (a Python data Structure) Type libraries are doing the real job.

Rendering

Graphics rendering is a matter of code generation. Customize the generator to style graphs.

Pipes

To generate code, ⊔ provides and uses UNIX like piped small tools from the graph Abstract Syntax Tree.

Syntax building blocks

- for Nodes:
  - `\w+`: an unicode word to identify the node
  - `\.(\w+|\d+)`: a named or indexed port (interpreted iff the node is connected)
  - `"[^"]+"`: a label on possibly several lines separator is simple quote, double quote or triple quotes
  - `:\w+`: Type
  - `(.+)`: arguments list
- for Edges:
  - `(<>-=)`: Arrow head
  - `"[^"]+"`: Label on possibly several lines separator is simple quote, double quote or triple quotes
  - `:\w+`: Type
  - `([^)]+)`: Arguments
  - `(<>-=)`: Arrow tail

Rockwell Collins

From the Dot (Graphviz) Language

- Dot[1] is not typed
- Dot composition (cluster) is not generic
- Dot ports are not (well) implemented
- Dot is not minimal (`A->B` raises syntax error)
- Dot mixes structure and layout
- Limited Dot layout algorithms (nodes place + arc path)

[1] AT&T Bell Laboratories

From the XML format

- XML is for XHTML what ⊔ is for (UML,Simulink,...)
- XML is basically suited for trees not graphs
- XML has a lot of glue characters
- XML does not enforce id on each elements
- XML use Xlink,Xpath for referencing
- XML raises attribute versus elements dilemma
- XML is unreadable in practice
- Transformations are complex (XSLT)
- Type checking using DTD,XSL,RelaxNG

⊔ Types

- User defines is own types library for:
  - Used Nodes
  - Used Edges

The types library:

- defines the semantics of the input formalism (UML,Scade,...)
- maps to output patterns (Ada,SVG,...)
- defines a **Domain Specific Language**
- customize graphic output

Two different nodes types may rendered with different shapes/decorations in SVG but may maps to the same class construction for Python generation.

Semi-Formal and Formal

A semi-formal node is a typed node with informal (english) sentence in its label.
A formal node is a types node with all attributes valid stream from formal languages.
The label may be used to embedd procedure, function, class definition on several lines.
The arguments may be used call, customize or instantiate.
The type definition may include default code.

Overload nodes rules

Node Definition and Node Usage are identical!

Node accumulates:

`A"hello" A:T`	$\equiv$	`A"hello":T`
`A"hello" A->B`	$\equiv$	`A"hello"->B`
`A->B A"hello"`	$\equiv$	`A"hello" A->B`
`A B C A`	$\equiv$	`C A B`
`A"label1" A"label2"`	$\equiv$	`A"label2"`
`A(arg1) A(arg2)`	$\equiv$	`A(arg2)`
`A:T1 A:T2`	$\equiv$	`A:T2`
`A{A}`	$\equiv$	`A`

Edge rules

Edges have no ids!

`A->B A->B`	$\neq$	`A->B`
`A -x> B A -y> B`	$\neq$	`A -y> B A -y> B`
`A -(1)> B A -(2)> B`	$\equiv$	`A -(2)> B A -(1)> B`
`A -(1)> B A -(2)> B`	$\neq$	`A -(2)> B A -(2)> B`

Expected code generation

- c
- java
- tikz
- vhdl
- scala
- python
- svg
- lustre
- ocaml
- aadl
- lua
- haskell
- ada
- sdl
- ruby
- systemc

Graphic generation

- SVG for Web publishing
- Tikz for TEX and PDF exporting

The layout (nodes placement and edge path) does not carry semantics
Do not let User define it, let advanced algorithms do the layout with goals:

- Equilibrate the nodes in the canvas
- Minimize edge crossing
- Find best path for edges
- Follow graphic design rules
- Follow Typographic rules

The same graph may have several styles (Themes) ...beautiful graphic output is a requirement ! (TEXprinciple)

Candidate model formalisms

- UML
- SysML
- AADL-Graph
- Marte
- Xcos
- Kaos
- Entity-Relation-Graph
- Tree-Diagram
- Network-Graph
- Flowchart
- Petri-net
- State-Machine
- Markov-Chain
- Behavior-Tree

Edge shape type

There is 16 possible arrow types for each edge type:

`-X>`	`=X>`	`>X>`	`<X>`
`-X<`	`=X<`	`>X<`	`<X<`
`-X-`	`=X-`	`>X-`	`<X-`
`-X=`	`=X=`	`>X=`	`<X=`

Where

`X` is an edge type; none or only one unicode character.

Needs

- A theoritical support
- A Constraint Definition Language (Real,OCL,...)
- A better types definition (currently dictionnary of properties)
- A Support for many code generators
- An Embedded and large test set
- Plugins for formal model checkers and theorem provers.

Next about ⊔!

Forge:

`https://github/pelinquin/u`

Source code:

See PDF attached file:u.py