The ⊔ Language

Laurent Fournier*

*lfournie@rockwellcollins.com

December 7^{th} 2011

What ⊔ is?

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

- Symbol: ⊔
- Name: "square cup"
- Universality (close to "u")
- Unicode character ⊔: (U+2294)
- License: GPL v3

Graph

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

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

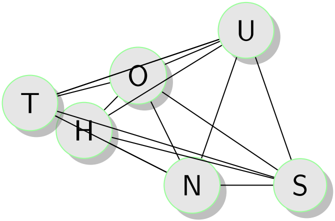
...a set of nodes (vertices) and a set of arcs (edges) between nodes.

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

Edge links some origin nodes set to some destination nodes set.
p and q are ports references
Rmq: $|i| > 1$ or $|j| > 1$ for multi-links.
Some attributes list is attached to each node v_i and each edge e_k.

⊔ THONUS features

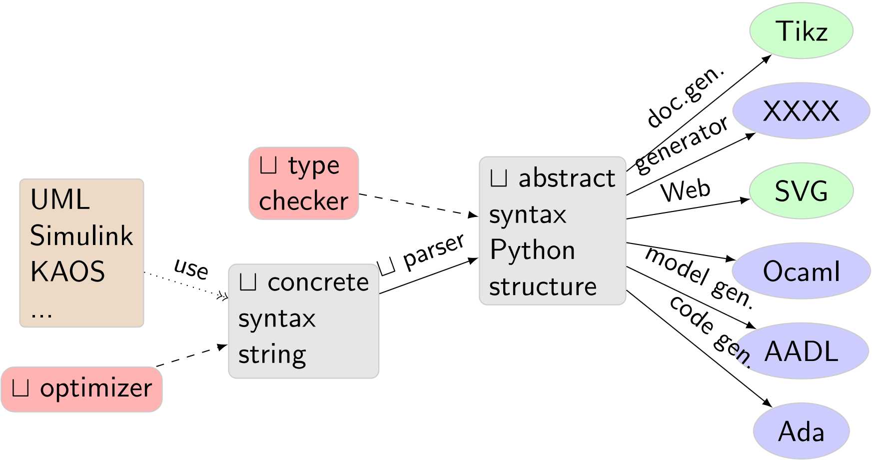
- T-yped
- H-ierachical
- O-nline
- N-eutral
- U-nicode
- S-hort



⊔ at a glance

Hello -> "World!"	Hello World!
A B foo bar	Some Nodes
A->B C<-D -- E	Nodes with links
A{B C {D}}	Node Composition
A->{B C}	Mulilinks
A"This is 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	Indexed and named ports

The big picture (with ⊔!)



⊔ facts

Structure

⊔ only manages the structure of the graph, not the semantics.
⊔ parser builds an Abstract Syntax Tree (a Python data Structure)
The types libraries are doing the real job!

Rendering

Graphics rendering is almost a matter of code generation.
Just customize the generator to style your graphs.

Pipes

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

Syntax building elements

- Nodes:
  - `ID`: a unicode word to identify the node
  - `port`: a named or indexed port (type compatible)
  - `label`: a string on possibly several lines separator is simple quote, double quote or triple quotes
  - `type`: Type name available in the node types library
  - `args`: arguments list
- Edges:
  - `(<>-=)`: Arrow head
  - `label`: a string on possibly several lines separator is simple quote, double quote or triple quotes
  - `type`: Type name available in the edge types library
  - `args`: Edge Arguments
  - `(<>-=)`: Arrow tail
- Blocks:
  - `{...}`: delimiters

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 a 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 Models (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 *attributes* versus *elements* dilemma
- XML is unreadable in practice
- Transformations are complex (XSLT)
- Type checking using DTD, XSD, RelaxNG

⊔ Types

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

A types library:

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

For instance, two different nodes types may be rendered with different shapes/decorations in SVG but may map to the same class construction for Scala 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 source code.

Overload nodes rules

rules:

Node definition and **Node usage** are identical!
Node, Edges accumulates properties:

`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

rule:

Edge has no id!

`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`

16 possible arrow types for each edge type `X`:

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

Candidate model formalisms

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

Expected code generation

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

Graphic generation

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

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

- Balance 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
TEX principle: nice graphic output is a requirement !

Today needs

- a theoretical support,
- a constraint definition language (Real,OCL,...),
- a better types definition (currently dictionnary of properties),
- templates for many code generators,
- an embedded and large test set,
- plugins for formal model checkers and theorem provers.

Next about ⊔!

All is on the forge:

`https://github/pelinquin/u`

Source code:

See PDF attached file:u.py

...and generate this beamer.