The ⊔ Language

Laurent Fournier*

*lfournie@rockwellcollins.com

December 7^{th} 2011

Rockwell Collins

What ⊔ is?

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

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

Rockwell Collins

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.

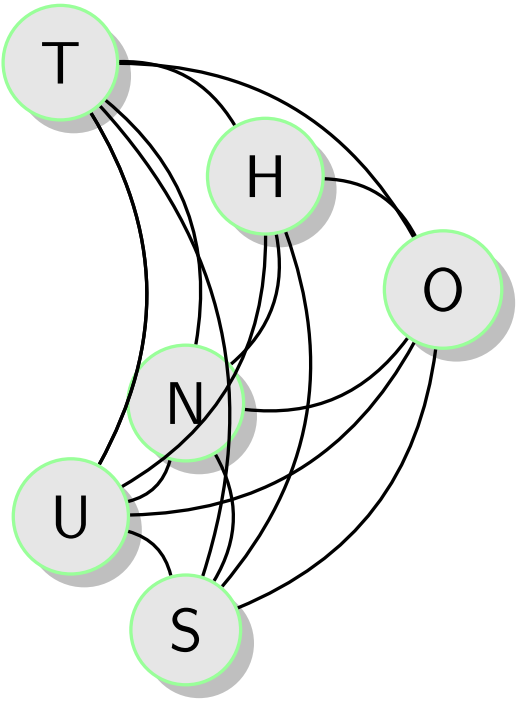
$$e_k = (\{v_{i_p}\}, \{v_{j_q}\})$$

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.

Rockwell Collins

⊔ THONUS features

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



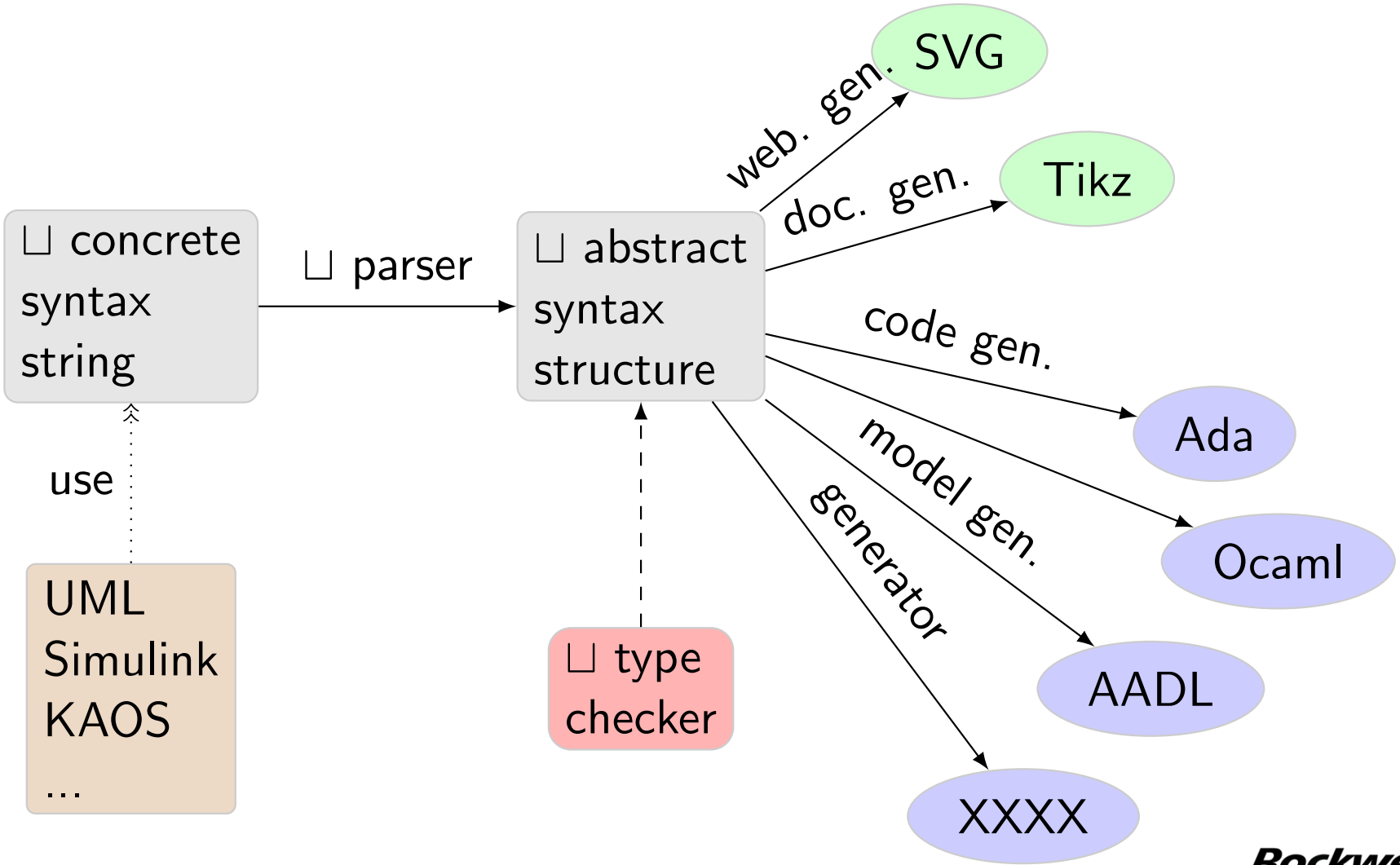
Rockwell Collins

⊔ 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

Rockwell Collins

The big picture



Rockwell Collins

The main principles

Structure

␣ only manage the structure of the graph, not the semantics.
␣ parser builds an Abstract Syntax Tree (a Python data Structure)
Types libraries are doing the real job.

Rendering

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

Pipes

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

Rockwell Collins

Syntax building blocks

- for Nodes:
  - `ID`: an 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
- for 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
- for blocks:
  - `{...}`: delimiters

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

Rockwell Collins

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

Rockwell Collins

⊔ 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.

Rockwell Collins

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.

Rockwell Collins

Overload nodes rules

rules:

Node Definition and Node Usage are identical!
Node,Edges accumulates properties:

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

Rockwell Collins

Edge rules

rule:

Edges have no ids!

```
A->B A->B              ≠  A->B
A -x> B A -y> B        ≠  A -y> B A -y> B
A -(1)> B A -(2)> B    ≡  A -(2)> B A -(1)> B
A -(1)> B A -(2)> B    ≠  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

Rockwell Collins

Expected code generation

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

Rockwell Collins

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)
TEXprinciple: nice graphic output is a requirement !

Rockwell Collins

Needs

- A theoretical 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.

Rockwell Collins

Next about ⊔!

All is on the forge:

`https://github/pelinquin/u`

Source code:

See PDF attached file:u.py
and generate this beamer

Rockwell Collins