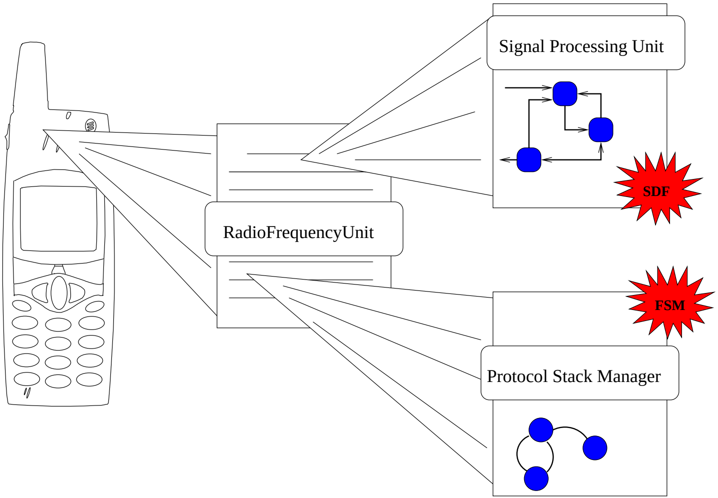
Rialto: A language for heterogeneous computations

Johan Lilius, ES-lab, Åbo Akademi, Turku - FINLAND

Lionel Morel, INRIA-IRISA, Rennes - FRANCE

Content

Context: Design of Heterogeneous Embedded Systems



Context: Design of Heterogeneous Embedded Systems

We need to understand 2 things:

1. What are the appropriate design techniques for describing different application domains
2. How can we combine these different approaches in a uniformed framework

Context: Design of Heterogeneous Embedded Systems

We need to understand 2 things:

1. What are the appropriate design techniques for describing different application domains
2. How can we combine these different approaches in a uniformed framework

This is quite well understood:

- Each engineering domain has a long history and solid models that fit the needs

Context: Design of Heterogeneous Embedded Systems

We need to understand 2 things:

1. What are the appropriate design techniques for describing different application domains
2. How can we combine these different approaches in a uniformed framework

Now This is not so well understood and needs more care:

- That's the long-term research goal in this project.

Our goals in this context

1. Develop a uniform operational mathematical model of Models of Computation
2. Propose a textual language to program these heterogeneous models

Remarks

This language has no "user-friendly" ambition:

- Be simple and (hopefully) "complete" (i.e. powerful enough)
- Serve as a "core-language for design of heterogeneous applications": provide translators to/from Rialto

History

- Rialto 1.0 presented in Dag Björklund thesis
  - Basic language with compilation
  - MoCs encapsulated through built-in policies
  - Code generation for C and VHDL, based on S-graphs
- Rialto 2.0, under development
  - Reflectivity interface: ability to define policies using Rialto syntax
  - Translation to Rialto 1.0 gives access to compiler

Content

Language features

Feature	**Language**					
	Esterel	Polis	SDL	SystemC	CCSS	UML Sc
Concurrency	X	X	X	X	X	X
Hierarchy	X	X	X	X	X	X
Preemption	X			X	X	X
Deterministic	X			O	X	
Communication:					X	
Synchronous	X			X		
Buffered		X	X	X		X
FIFO			X	O		X
Procedural	X	O	O	X	O	
FSM	X	X	X	O	X	X
Dataflow		X	X	X	X	
Multi-rate DF					X	
Software	X	X	X	X	X	X
Hardware	X	X		X	X	

Rialto: the language - some motivations

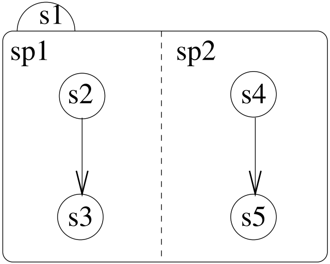
- Many languages use the same syntactic concepts but with different semantics
- These features include:

  concurrency, interrupts, sequence, choice, atomicity, encapsulation

$\Rightarrow$ Let's pinpoint the semantics differences between these interpretations

- Separate syntactic structure from concurrency/scheduling concerns

Language features (example)

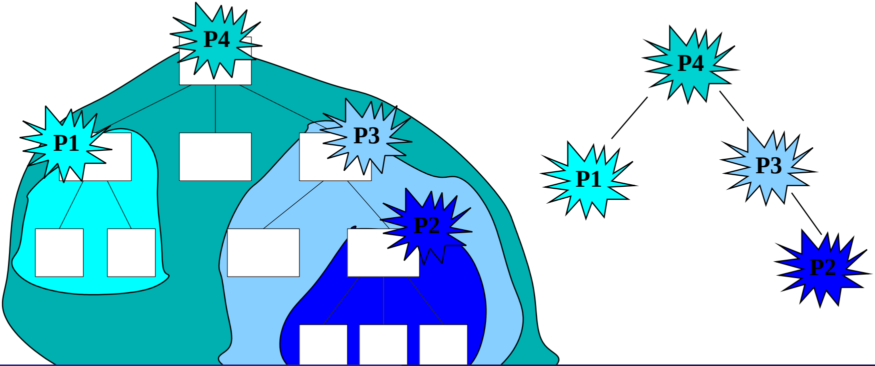



```
begin
s1 : state
      policy interleaving;
begin
l1 : par
sp1 : state
        begin
s2 :       state
           begin
              goto s3;
           endstate; // s2
s3 :       state
           begin
              goto s2;
           endstate; // s3
        endstate; // sp1
||
sp2: state
        begin
s4:        state
           begin
              goto s5;
           endstate; // s4
s5:        state
           begin
              goto s4
           endstate; // s5
        endstate; // sp2
endpar;
endstate; //s1
end;
```

```
program InterleavingTest
   policy interleaving
      var l: label;
      begin
         l := sc.prevProgCtx.
             getLagelFromActiveSet();
         return l;
      end;
```

A hierarchy of blocks and a hierarchy of policies

- As shown in the previous example, Rialto programs are decomposed in "blocks", organized hierarchically
- What a "block" is depends on you: state, component, etc.
- A scheduling policy is attached to each block. It defines how the block should be "interpreted" exactly.



Semantics

- 2-level semantics based on a SOS formalization
- atomic statements have SOS rule to define semantics
- Interpretation is in 2 phases:
  - program interpretation
  - policy interpretation
  - cf. macro/micro-step semantics in Statecharts

Semantics: State Configuration

Definition

State Configuration $sc = (active, suspended)$ where:

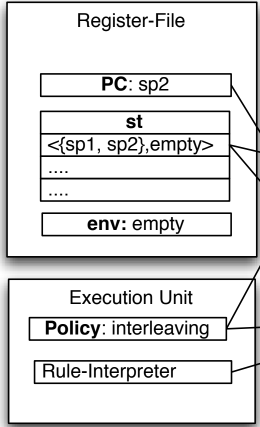
- $sc.active$ the set of active labels (labels of statements that need to be executed)
- $sc.suspended$ the set of suspended labels (labes that have been suspended)

Definition

Run-time Configuration Rialto program stack (st,env,pc), where:

- the program stack *st* stores state configuration
- the environment for the program *env* stores variables' values
- the program counter *pc* points to the currently executed statement

Semantics: Intuition - The Rialto Machine



Label	Statement	Policy
l1	program InterleavingTest	default
l5	l := sc.prevProgCtx. getLabelFromActiveSet()	default
l6	return l	default
s1	state	interleaving
l1	par (sp1, sp2)	interleaving
sp1	state	interleaving
s2	state	interleaving
l12	goto s3	default
s3	state	default
l13	goto s2	default
sp2	state	interleaving
s4	state	interleaving
l12	goto s5	default
s5	state	default
l13	goto s4	default

Semantics

Template Rule

$$\frac{\mathcal{P}[PC] = \text{``}stmt\text{''} \quad \text{``}otherconditions\text{''}}{\text{``}stmtstatechange\text{''} \quad PC = \perp}$$

Parallel composition

$$\frac{\mathcal{P}[PC] = \textbf{par} \;\; \textbf{stmt}(\| \; \textbf{stmt})^* \;\; \textbf{endpar} \land PC \neq \perp}{st.active = st.active \backslash \{PC\} \cup children(PC) \land PC = \perp}$$

Content

RialtoMachine and Policy Interaction

- The RialtoMachine can be in two *modes*
  1. Executing the program, or
  2. Executing a policy
- The $\perp$ special label is used to switch between modes
- The job of the policy is to select the right statement (according to the MoC) and put it into the program counter.

Reflexivity

- In Rialto 1.0, policies were fixed and implemented in the compiler
- In Rialto 2.0, policies are defined in Rialto 2.0
- Mechanisms:
  - Access to program state
  - Access to program structure
- Currently implemented through built-in functions

Semantics: Entering a policy

"Every time a statement is interpreted, give control to the policy..."

- This is performed by setting the **PC** to "$\bot$" (done in every SOS rule)
- Then, entering a policy is defined by:

$$\frac{PC = \bot}{\begin{array}{l} PC = lub(st.active).policyDesc \\ push(st, PC) \end{array}}$$

Semantics: Exiting a policy

Now, how do we get back to executing the "real" program?

- specific **return** statement:

$$\frac{\mathcal{P}[PC] = \textbf{return l} \wedge PC \neq \perp}{\begin{array}{l} PC = Env[l] \\ pop(st) \end{array}}$$

- **l** is "computed" by the policy itself

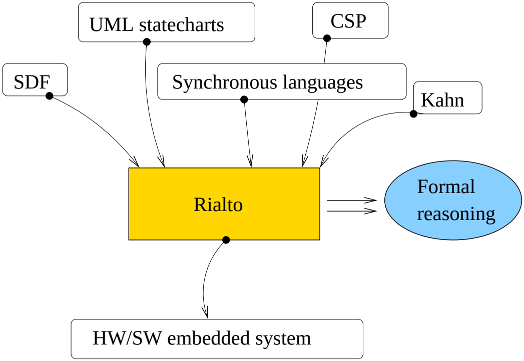
Content

Conclusions

- Approach seems to work?
- Can be used to give semantics to (subsets of) UML in a nice and consistent way
- Has code-generation
- Has UML front-end

Ongoing and future work (cont'd)

- Explore various modeling paradigms/languages and see how they fit Rialto
- Explore generation of efficient HW/SW implementations from Rialto



Ongoing and future work

- Connect to denotational semantics
  - Tagged Value model
  - ForSyDe
- Rialto could be given a semantics in terms traces
- Prove that the traces of a Rialto program in a certain MoC have the properties as specified in the Tagged value model

Ongoing and Future Work (cont'd)

- Case-study (jpeg encoder/decoder)
- Explore communication part (data): for the moment, limited to Fifos
- Study correspondance between MoConcurrency and MoCommunication (what is the adequate style of communication for a given style of concurrency?)
- Modeling of synchrony hypothesis

The Rialto team

- Pr. Johan Lilius, Åbo Akademi, Turku, Finland
- Dr. Lionel Morel - IRISA/INRIA Rennes, France
- M.Sc. Student Andreas Dahlin, Åbo A
- M.Sc. Student Markus Dahlgård, Åbo A
- Alumni: Dag Bjorklund, PhD2005 who defined a first version of Rialto

For more info...

http://mde.abo.fi/confluence/display/Rialto20/Home

Thank You!