Metropolis

Design Environment for Heterogeneous Systems

Metropolis Project Team

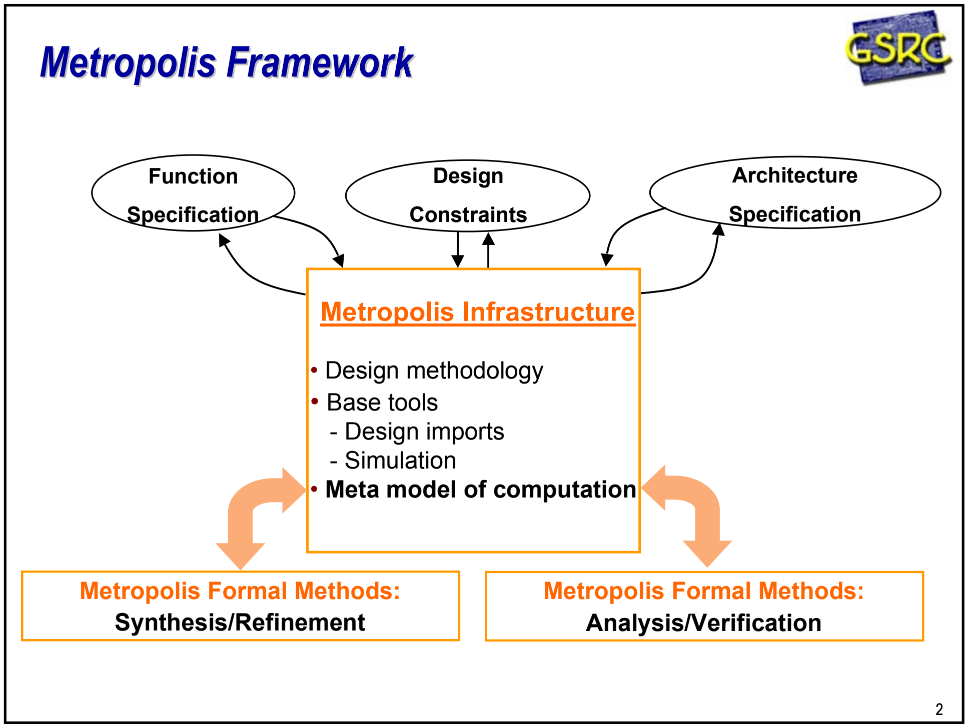
etropolis

SISA SEMICONDUCTOR INDUSTRY SUPPLIERS ASSOCIATION

SIA SEMICONDUCTOR INDUSTRY ASSOCIATION

DUSD(S&T)

DARPA



Metropolis Framework

Metropolis Infrastructure

- Design methodology
- Base tools
  - Design imports
  - Simulation
- **Meta model of computation**

Metropolis Formal Methods:
Synthesis/Refinement

Metropolis Formal Methods:
Analysis/Verification

Metropolis Meta Model

- Do not commit to the semantics of a particular Model of Computation (MoC)
- Define a set of "building blocks":
  - specifications with many useful MoCs can be described using the building blocks.
  - unambiguous semantics for each building block.
  - syntax for each building block ➡ a language of the meta model.
- Represent behavior at all design phases; mapped or unmapped

Question: What is a good set of building blocks?

Metropolis Meta Model

The behavior of a concurrent system:

computation
- f: X → Z
- firing rule

process

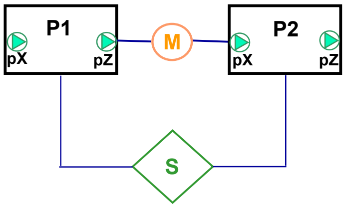
communication
- state
- methods to
  - store data
  - retrieve data

medium

coordination
- constraints on concurrent actions
- algorithms to enforce the constraints

scheduler

```
process P1{
port pX, pZ;
thread(){
  // condition to read X
  // an algorithm for f(X)
  // condition to write Z
}
```



P1.pZ.write() ◆ P2.pX.read()

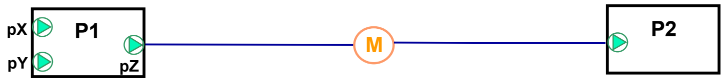
```
medium M{
   int[] storage;
   int space;

   void write(int[] z){ ... }

   int[] read(){ ... }
}
```

Processes and Media

await(cond)[actions]{ statements; }
- wait for cond to hold,
- once cond becomes true, do **statements** exclusively wrt **actions**.



```
process P1{
  port reader pX, pY;
  port writer pZ;
  thread(){
   while(true){
   ...
    await(pX.n()>0 && pY.n()>0)
      [pX.reader,pY.reader]
        z = f(pX.read(), pY.read());

    pZ.write(z);
    ...
    }
}
```

```
medium M implements reader, weiter{
   word storage;
   int n, space;
   void write(word z){
       await(space>0)[this.writer]
             n=1; space=0; storage=z;
    }
    word read(){ ... }
}
```

Constraints

Two mechanisms are supported to specify constraints:

1. Propositions over temporal orders of states
   - ▲ execution is a sequence of states
   - ▲ specify constraints using temporal logic
   - ▲ good for scheduling constraints, e.g.

     "if process P starts to execute a statement s1, no other process can start the statement until P reaches a statement s2."

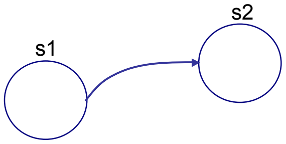
2. Propositions over instances of transitions between states
   - ▲ particular transitions in the current execution: called "*actions*"
   - ▲ annotate actions with quantity, such as time, power.
   - ▲ specify constraints over actions with respect to the quantities
   - ▲ good for real-time constraints, e.g.

     "for any successive actions of starting a statement s1 by process P must take place with at most 10ms interval."

Constraints

1. Propositions over temporal orders of states

State variables
- process:
  - instances of local variables of called functions
  - program counter:
    {**beg(s)**, **end(s)**} for each statement **s**
- medium
  field instances



- execution (s1, s2, ...) : a linear (possibly infinite) order of states such that
  - it starts from the initial state,
  - each adjacent pair is a transition

Propositions on Temporal Order of States

- Linear Temporal Logic (LTL):
  propositions over state variables
  - temporal operators: **X, U, F, G**
  - logical operators: &&, !, ||, **->**, **<->**
  - ltl() method to specify constraints
- Built-in constructs on the LTL:
  excl, mutex, simul

constraints{...} can appear anywhere in the meta-model programs.

```
medium M{
  word storage;
  int n, space;
  void write(word z){
  wr: {
    await(space>0)[this]
        n=1; space=0; storage=z;
    }
  }
  word read(){
  rd: {
    await(n>0)[this]
        n=0; space=1; return storage;
    }
  }
  constraints{
      process p, q;
      ltl(G(pc(p)==beg(wr) ->
          F(pc(q)==end(rd))));
  }
}
```

Constraints

2. Propositions over instances of transitions between states

- Action: instantiation of a transition in an execution (s1, s2, ...)

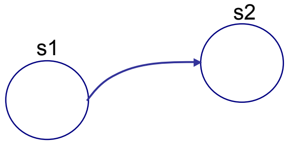
  action *a = (p, s_c, s_n, o)*

  p : process object

  s_c : current value of the program counter of *p*

  s_n : next value of the program counter of *p*

  o : occurrences of the transition $s_c \rightarrow s_n$ by *p* in the execution



- Quantity: annotated with the set A of actions of the current execution
  - The domain D of the quantity, e.g. *real* for the global time
  - The operations and relations on D, e.g. subtraction, <, =
  - The relation between D and A, e.g. *gt(a)* denotes the global time of the action *a*
  - Constraints on the quantity and actions, e.g.

    for all actions a1, a2, if a2 follows a1 in the execution, *gt(a1) < gt(a2)*

Constraints using Actions

◆ public final class Action {
```
    process p;
    pcval sc, sn;
    int o;
}
```

◆ public class Gtime extends Quantity {
```
    static double t;
    double sub(double t2, double t1){...}
    boolean equal(double t1, double t2){ ... }
    boolean less(double t1, double t2){ ... }
    double gtime(Action a){ ... }
    constraints{ ... }
}
```

```
process P1{
  port reader pX, pY;
  port writer pZ;
  thread(){
   while(true){
   ...
      await(pX.n()>0 && pY.n()>0)
          [pX.reader,pY.reader]
l1:          z = f(pX.read(), pY.read());
l2:   pZ.write(z);
      ...
    }
  }
}
```

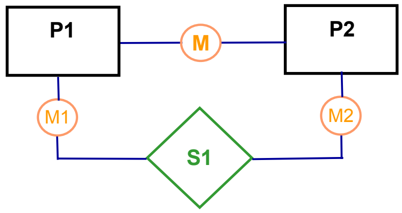
```
constraints{
    Action a1, a2;
    Gtime gt;
    lfo(a1.p()==a2.p() && a1.sn()==beg(l1)
        && a2.sn==end(l2) && a1.o()== a2.o()
        ->  gt.gtime(a2) - gt.gtime(a1) < 5);

    lfo(a1.p()==a2.p() && a1.sn()==beg(l1)
        && a2.sn==beg(l1)
        && a2.o()-a1.o()==1
        ->  gt.gtime(a2) - gt.gtime(a1) < 10);
```

```
constraints{
    maxLatency(l1, l2, 5);
    maxRate(l1, 10);
}
```

Schedulers

- Scheduler specifies an algorithm for *some* constraints.



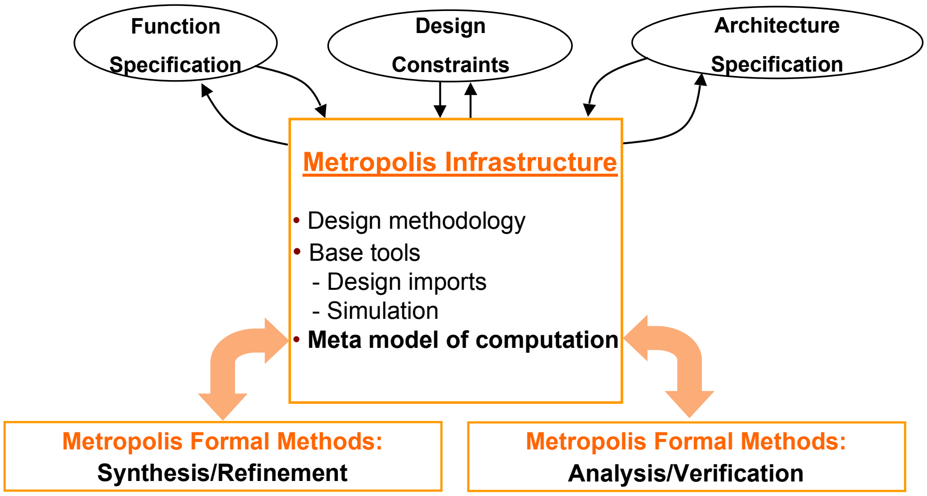
```
scheduler S1{
  port SMsched port0, port1;
  ...
  void doScheduling(void){
      // priority scheduling
  }
}
```

- The algorithms are used during simulation.
- Typically, later in the design phase, thread() is added to a scheduler,
  - to specify protocols to communicate with the controlled processes,
  - to call doScheduling() as a sub-routine.

At that point, the scheduler becomes a process.

- Schedulers may be hierarchical.

Metropolis Framework



Function Specification

Design Constraints

Architecture Specification

Metropolis Infrastructure

- Design methodology
- Base tools
  - Design imports
  - Simulation
- **Meta model of computation**

Metropolis Formal Methods:
Synthesis/Refinement

Metropolis Formal Methods:
Analysis/Verification

Formal Models

Formal model: derived from the meta-model for applying formal methods

- **Mathematical formulations** of the semantics of the meta model:
  - each construct (‘if’, ‘for’, ‘await’, ...)
  - sequence of statements
  - composition of connected objects
  - ★ the semantics may be abstracted
- **Restrictions** on the meta model

Formal methods (verification and synthesis) applicable on given models

---

Formal Models

Example: Petri nets

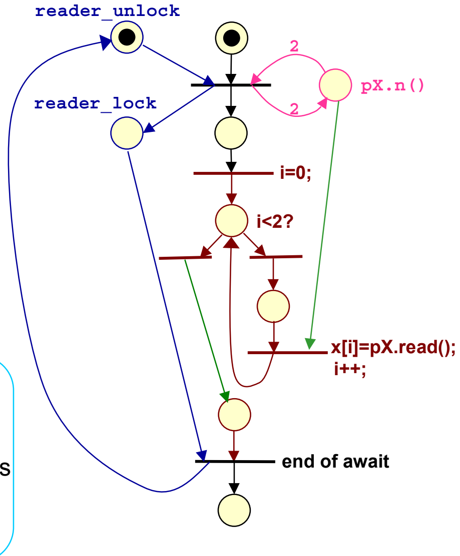
```
await(pX.n()>=2)[pX.reader]
  for(i=0; i<2; i++) x[i]=pX.read();
```

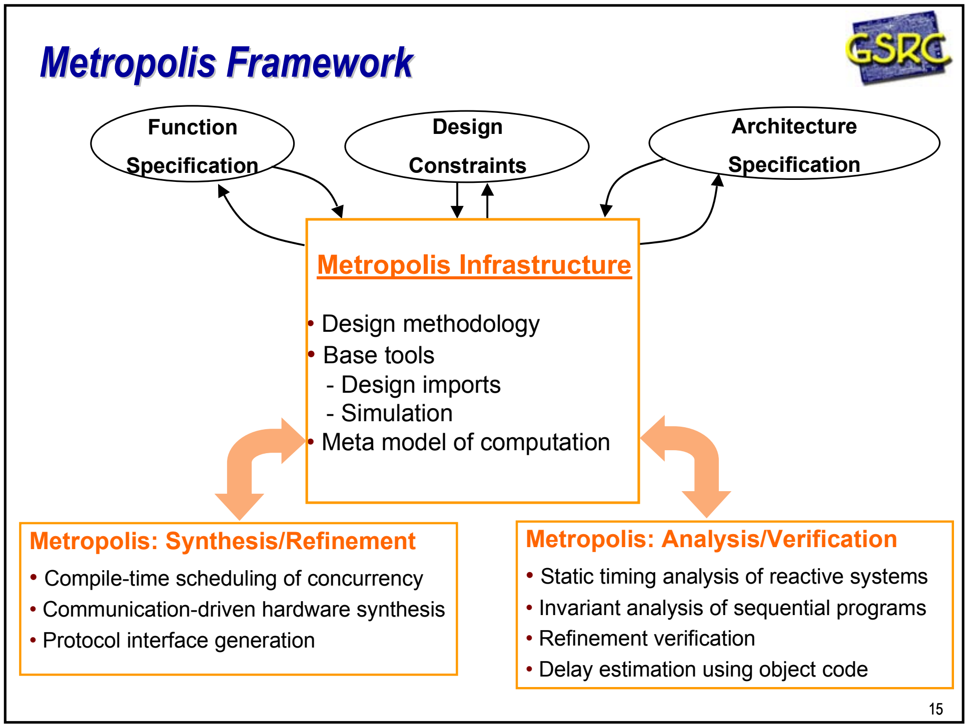
Restriction:

condition inside await is conjunctive.

Formal Methods on Petri nets:

- analyze the schedulability
- analyze upper bounds of storage sizes
- synthesize schedules





Metropolis Framework

Function Specification

Design Constraints

Architecture Specification

Metropolis Infrastructure

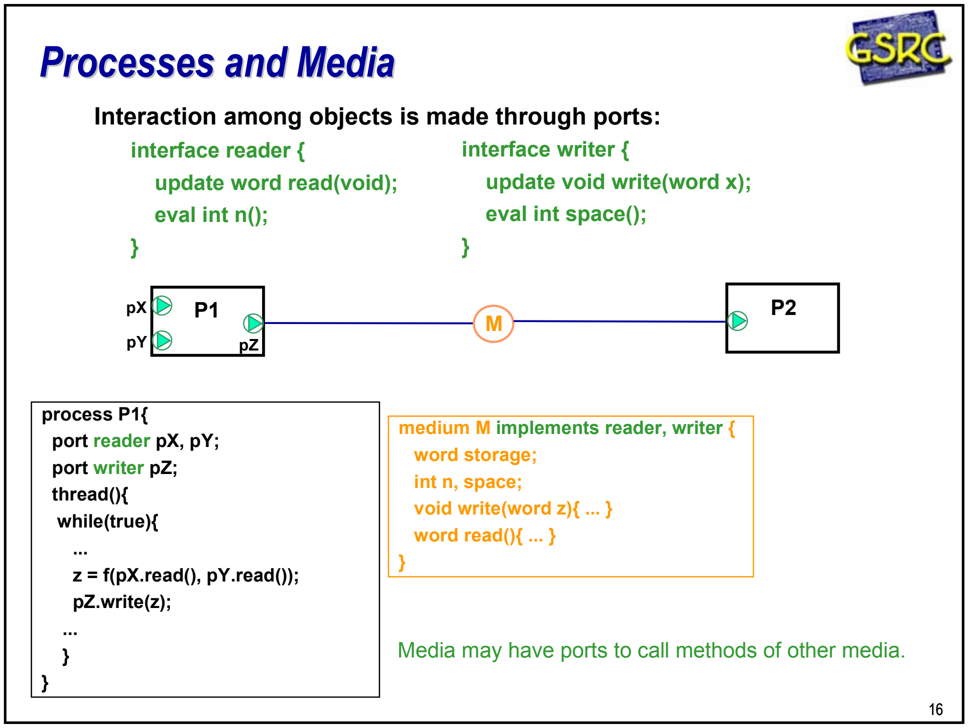
- Design methodology
- Base tools
  - Design imports
  - Simulation
- Meta model of computation

Metropolis: Synthesis/Refinement

- Compile-time scheduling of concurrency
- Communication-driven hardware synthesis
- Protocol interface generation

Metropolis: Analysis/Verification

- Static timing analysis of reactive systems
- Invariant analysis of sequential programs
- Refinement verification
- Delay estimation using object code



Processes and Media

Interaction among objects is made through ports:

```
interface reader {
   update word read(void);
   eval int n();
}
```

```
interface writer {
   update void write(word x);
   eval int space();
}
```

```
process P1{
 port reader pX, pY;
 port writer pZ;
 thread(){
  while(true){
    ...
    z = f(pX.read(), pY.read());
    pZ.write(z);
   ...
  }
}
```

```
medium M implements reader, writer {
   word storage;
   int n, space;
   void write(word z){ ... }
   word read(){ ... }
}
```

Media may have ports to call methods of other media.