Metropolis

Design Environment for Heterogeneous Systems

Metropolis Project Team

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
- 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**
  - \( f: X \rightarrow Z \)
  - firing rule

- **Communication**
  - state
  - methods to
    - store data
    - retrieve data

- **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
  }
}

medium M{
  int[] storage;
  int space;
  void write(int[] z){ ... }
  int[] read(){ ... }
}
```

```
P1.pZ.write() \& P2.pX.read()
```
Processes and Media

```
await(\text{cond})[\text{actions}](\text{statements};)
  * wait for \text{cond} to hold,
  * once \text{cond} becomes true, do \text{statements} exclusively wrt \text{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, writer{
  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:
    - \((\text{beg}(s), \text{end}(s))\) for each statement \(s\)
- **medium** field instances

- execution \((s_1, s_2, \ldots)\): 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.
Constraints

2. Propositions over instances of transitions between states

- Action: instantiation of a transition in an execution \((s_1, s_2, \ldots)\)
  
  \[
  \text{action } a = (p, s_c, s_n, o)
  \]

  \(p\) : 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 \(a_1, a_2\), if \(a_2\) follows \(a_1\) in the execution, \(gt(a_1) < gt(a_2)\)

Constraints using Actions

- public final class Action {
  
  \[
  \begin{align*}
  \text{process } p; \\
  \text{pcval } sc, sn; \\
  \text{int } o;
  \end{align*}
  \]

  process P1{
    
    \[
    \text{port reader } pX, pY; \\
    \text{port writer } pZ; \\
    \text{thread}(){}
    \]

    \[
    \begin{align*}
    \text{while(true){} } \\
    \text{\quad await}(pX.n()>0 \&\& pY.n()>0) \text{[pX.reader,pY.reader]} \\
    \text{\quad l1: } z = f(pX.read(), pY.read()); \\
    \text{\quad l2: } pZ.write(z); \\
    \text{\quad …} \\
    \text{\quad }\}
    \end{align*}
  }

  \]

- public class Gtime extends Quantity {
  
  \[
  \begin{align*}
  \text{static double } t; \\
  \text{double sub(double t2, double t1)\{...\}} \\
  \text{boolean equal(double t1, double t2)\{...\}} \\
  \text{double less(double t1, double t2)\{...\}} \\
  \text{double gtime(Action a)\{...\}} \\
  \text{constraints\{ ...\}}
  \end{align*}
  \]

  constraints{
  
  \[
  \begin{align*}
  \text{Action } a_1, a_2; \\
  \text{Gtime } gt; \\
  \text{if}(a_1.p()==a_2.p() \&\& a_1.sn()==\text{beg(l1)} \\
  \quad \&\& a_2.sn==\text{end(l2)} \&\& a_1.o()==a_2.o()) \\
  \quad \rightarrow gt.gtime(a_2) - gt.gtime(a_1) < 5); \\
  \text{if}(a_1.p()==a_2.p() \&\& a_1.sn()==\text{beg(l1)} \\
  \quad \rightarrow \text{gt.time(a1) < 10);}
  \end{align*}
  \]

  constraints{
  
  \[
  \begin{align*}
  \text{Action } a_1, a_2; \\
  \text{Gtime } gt; \\
  \text{if}(a_1.p()==a_2.p() \&\& a_1.sn()==\text{beg(l1)} \\
  \quad \rightarrow \text{gt.time(a1) < 10});
  \end{align*}
  \]
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

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

- Metropolis Formal Methods:
  - Synthesis/Refinement
  - 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

---

**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.