Metropolis Metamodel
Metropolis Objects

- Metropolis elements adhere to a “separation of concerns” point of view.

- **Processes (Computation)**
  
  \[
  P_1 \xrightarrow{\text{Proc}_1} P_2
  \]
  
  **Active Objects**
  
  Sequential Executing Thread

- **Media (Communication)**
  
  \[
  I_1 \xrightarrow{\text{Media}_1} I_2
  \]
  
  **Passive Objects**
  
  Implement Interface Services

- **Quantity Managers (Coordination)**

  \[
  QM_1
  \]
  
  Schedule access to resources and quantities

Copyright A. Sangiovanni-Vincentelli
Metro. Netlists and Events

**Problem Statement**

**Approach**

**Contribution**

**Metropolis Architectures are created via two netlists:**
- Scheduled – generate events\(^1\) for services in the scheduled netlist.
- Scheduling – allow these events access to the services and annotate events with quantities.

![Diagram showing scheduled and scheduling netlists with Proc\(_1\), Proc\(_2\), P\(_1\), P\(_2\), I\(_1\), I\(_2\), Media\(_1\), QM\(_1\), and Global Time nodes.]

**Related Work**


Copyright A. Sangiovanni-Vincentelli
Key Modeling Concepts

- **An event** is the fundamental concept in the framework
  - Represents a transition in the **action automata** of an object
  - An event is owned by the object that exports it
  - During simulation, generated events are termed as **event instances**
  - Events can be annotated with any number of quantities
  - Events can partially expose the state around them, constraints can then reference or influence this state

- **A service** corresponds to a set of **sequences of events**
  - All elements in the set have a common begin event and a common end event
  - A service may be parameterized with arguments

---


Copyright A. Sangiovanni-Vincentelli
Action Automata

• Processes take *actions*.
  – statements and some expressions, e.g.
    \[ y = z + \text{port.f();, } z + \text{port.f()}, \text{ port.f()}, \ i < 10, \ldots \]
  – only calls to media functions are *observable actions*

• An *execution* of a given netlist is a sequence of vectors of *events*.
  – *event*: the beginning of an action, e.g. \( B(\text{port.f()}) \),
    the end of an action, e.g. \( E(\text{port.f()}) \), or null \( N \)
  – the \( i \)-th component of a vector is an event of the \( i \)-th process

• An execution is *legal* if
  – it satisfies all coordination constraints, and
  – it is accepted by all action automata.
Execution semantics

Action automaton:

- one for each action of each process
  - defines the set of sequences of events that can happen in executing the action

- a transition corresponds to an event:
  - it may update shared memory variables:
    - process and media member variables
    - values of actions-expressions
  - it may have guards that depend on states of other action automata and memory variables

- each state has a self-loop transition with the null N event.

- all the automata have their alphabets in common:
  - transitions must be taken together in different automata, if they correspond to the same event.
Action Automata

- $y = x + 1$;

```

B y = x + 1

B x + 1

E x + 1

E y = x + 1

E y := V_{x+1}

* = write y

B x + 1

E x + 1

E x + 1

E y = x + 1

y := any

y := V_{x+1}

write y

x + 1

write x

V_{x+1} := x + 1

V_{x+1} := any

```

<table>
<thead>
<tr>
<th>V_{x+1}</th>
<th>0</th>
<th>5 1</th>
<th>5 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>y</td>
<td>0</td>
<td>0</td>
<td>5 1</td>
</tr>
<tr>
<td>x</td>
<td>0</td>
<td>0</td>
<td>0 0</td>
</tr>
</tbody>
</table>

**B y = x + 1**  **N**  **B x + 1**  **N**  **N**  **E x + 1**  **E y = x + 1**
Semantics summary

• Processes run sequential code concurrently, each at its own arbitrary pace.

• Read-Write and Write-Write hazards may cause unpredictable results
  – atomicity has to be explicitly specified.

• Progress may block at synchronization points
  – awaits
  – function calls and labels to which awaits or constraints refer.

• The legal behavior of a netlist is given by a set of sequences of event vectors.
  – multiple sequences reflect the non-determinism of the semantics:
    concurrency, synchronization (awaits and constraints)
Metropolis Architecture Representation
An architecture component specifies *services*, i.e.

- what it *can* do
- how much it *costs*
Meta-model: architecture components

An architecture component specifies services, i.e.

- **what it *can* do:**
  - interfaces, methods, coordination (awaits, constraints), netlists
- **how much it *costs:***
  - quantities, annotated with events, related over a set of events

```java
interface BusMasterService extends Port {
    update void busRead(String dest, int size);
    update void busWrite(String dest, int size);
}
```

```java
medium Bus implements BusMasterService {
    port BusArbiterService Arb;
    port MemService Mem; ...
    update void busRead(String dest, int size) {
        if(dest== ... ) Mem.memRead(size);
    }
}
```

Meta-model: quantities

- The domain D of the quantity, e.g. \textit{real} for the global time,
- The operations and relations on D, e.g. subtraction, $<$, $=$,
- The function from an event instance to an element of D,
- Axioms on the quantity, e.g.
  
  the global time is non-decreasing in a sequence of vectors of any feasible execution.

\begin{verbatim}
class GTime extends Quantity {
    double t;
    double sub(double t2, double t1){...}
    double add(double t1, double t2){...}
    boolean equal(double t1, double t2){ ... }
    boolean less(double t1, double t2){ ... }
    double A(event e, int i){ ... }
    constraints{
        forall(event e1, event e2, int i, int j):
            GXI.A(e1, i) == GXI.A(e2, j) -> equal(A(e1, i), A(e2, j)) &&
            GXI.A(e1, i) < GXI.A(e2, j) -> (less(A(e1, i), A(e2, j)) ||
               equal(A(e1, i), A(e2. j)));
    }
}
\end{verbatim}
**Meta-model: architecture components**

- This modeling mechanism is generic, independent of services and cost specified.
- Which levels of abstraction, what kind of quantities, what kind of cost constraints should be used to capture architecture components?
  - depends on applications: *on-going research*

**Transaction:**
- Services:
  - fuzzy instruction set for SW, execute() for HW
  - bounded FIFO (point-to-point)
- Quantities:
  - #reads, #writes, token size, context switches

**Virtual BUS:**
- Services:
  - data decomposition/composition
  - address (internal v.s. external)
- Quantities: same as above, different weights

**Physical:**
- Services: full characterization
- Quantities: time

Copyright A. Sangiovanni-Vincentelli
Quantity resolution

The 2-step approach to resolve quantities at each state of a netlist being executed:

1. **quantity requests**
   for each process $P_i$, for each event $e$ that $P_i$ can take, find all the quantity constraints on $e$.

   In the meta-model, this is done by explicitly requesting quantity annotations at the relevant events, i.e. `Quantity.request(event, requested quantities)`.

2. **quantity resolution**
   find a vector made of the candidate events and a set of quantities annotated with each of the events, such that the annotated quantities satisfy:
   - all the quantity requests, and
   - all the axioms of the Quantity types.

   In the meta-model, this is done by letting each Quantity type implement a `resolve()` method, and the methods of relevant Quantity types are iteratively called.
   - theory of fixed-point computation
Quantity resolution

- The 2-step approach is same as how schedulers work, e.g. OS schedulers, BUS schedulers, BUS bridge controllers.
- Semantically, a scheduler can be considered as one that resolves a quantity called *execution index*.
- Two ways to model schedulers:
  1. As processes:
     - explicitly model the scheduling protocols using the meta-model building blocks
     - a good reflection of actual implementations
  2. As quantities:
     - use the built-in request/resolve approach for modeling the scheduling protocols
     - more focus on resolution (scheduling) algorithms, than protocols: suitable for higher level abstraction models


<table>
<thead>
<tr>
<th>Feature</th>
<th>Metropolis</th>
<th>Rapide</th>
<th>ForSyDe</th>
<th>SPADE</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mapping</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>Quantity Managers</td>
<td>x</td>
<td>No</td>
<td>No</td>
<td>No; collectors in bldg blocks</td>
</tr>
<tr>
<td>Event Based</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>No</td>
</tr>
<tr>
<td>Pure Architecture Model</td>
<td>x</td>
<td>x</td>
<td>No; Functional tied to Arch.</td>
<td>x</td>
</tr>
</tbody>
</table>
**Programmable Arch. Modeling**

- **Computation Services**
  - PPC405
  - MicroBlaze
  - SynthMaster
  - SynthSlave

  **Computation Services**
  Read (addr, offset, cnt, size), Write(addr, offset, cnt, size),
  Execute (operation, complexity)

- **Communication Services**
  - Processor Local Bus (PLB)
  - On-Chip Peripheral Bus (OPB)
  - BRAM

  **Communication Services**
  addrTransfer(target, master)
  addrReq(base, offset, transType, device)
  addrAck(device)
  dataTransfer(device, readSeq, writeSeq)
  dataAck(device)

- **Other Services**
  - OPB/PLB Bridge
  - Mapping Process

  **Task Before Mapping**
  Read (addr, offset, cnt, size)
• Coordination Services

- PPC Sched
- MicroBlaze Sched
- PLB Sched
- OPB Sched
- BRAM Sched
- General Sched

Request (event e)
- Adds event to pending queue of requested events

PostCond()
- Augment event with information (annotation). This is typically the interaction with the quantity manager.

Resolve()
- Uses algorithm to select an event from the pending queue.
1. Create template system description.

2. Generate many permutations of the architecture using this template and run them through programmable platform tool flow.

3. Extract the desired performance information from the tool reports for database population.

Need to tie the model to actual implementation data!
Create database **ONCE** prior to simulation and populate with independent (modular) information.

1. **Data detailing performance based on physical implementation.**

2. Data detailing the composition of communication transactions.

3. **Data detailing the processing elements computation.**

**Work with Xilinx Research Labs**


Copyright A. Sangiovanni-Vincentelli
Arch. Refinement Verification

- Architectures often involve hierarchy and multiple abstraction levels.
- These techniques are limited if it is not possible to check if elements in hierarchy or less abstract components are implementations of their counterparts.
- Asks “Can I substitute M1 for M2?”

1. Representing the internal structure of a component.
2. Recasting an architectural description in a new style.
3. Applying tools developed for one style to another style.


<table>
<thead>
<tr>
<th>Refinement Technique</th>
<th>Description</th>
<th>Metropolis</th>
</tr>
</thead>
<tbody>
<tr>
<td>Style/Pattern Based</td>
<td>Define template components. Prove they have a desired relationship once. Build arch. from them.</td>
<td>Potential; TTL YAPI</td>
</tr>
<tr>
<td>Event Based</td>
<td>Properties (behaviors) expressed as event lists. Explicitly look for this event patterns.</td>
<td>Discussed</td>
</tr>
<tr>
<td>Interface Based</td>
<td>Create structure capturing all behavior of a components interface. Compare two models.</td>
<td>Discussed</td>
</tr>
</tbody>
</table>
1. Select an application and understand its behavior.
2. Create a Metropolis functional model which models this behavior.
3. Assemble an architecture from library services or create your own services.
4. Map the functionality to the architecture.
5. Extract a structural file from the top-level netlist of the architecture created.

**File for Xilinx EDK Tool Flow**

---

**Example Design**

**JPEG Encoder Function Model (Block Level)**

- Preprocessing
- DCT
- Quantization
- Huffman
- Structure Extractor
- Top Level Netlist
- SynthMaster
- SynthSlave
- On-Chip Peripheral Bus
- MicroBlaze
- BRAM
1. Feed the captured transactions to the structural file to the permutation generator.

2. Feed the permutations to the Xilinx tools and extract the data.

3. Capture execution info for software and hardware services.

4. Provide transaction info for communication services.

Permutation 1

Permutation 2

Permutation N

Platform Characterization Tool (Xilinx EDK/ISE Tools)

Characterizer Database

File for Xilinx EDK Tool Flow

Permutation Generator

Software Routines

int DCT (data){
  Begin
  calculate ...
  }

Manual

Hardware Routines

DCT1 = 10 Cycles
DCT2 = 5 Cycles
FFT = 5 Cycles

Manual

32 Bit Read = Ack, Addr
Automatic

ISS Info

Transaction Info

Char Data

Copyright A. Sangiovanni-Vincentelli
Example Design Cont.

Backend Tool Process:
1. Abstract Syntax Tree (AST) retrieves structure.

2. Control Data Flow Graph - **Depth**
   - **FORTE** – Intel Tool
   - **Reactive Models** – UC Berkeley

3. Event Traces – **Refinement Properties**
   - **Vertical Refinement**
   - **Horizontal Refinement**

1. Simulate the design and observe the performance.

**Execution time 100ms**
**Bus Cycles 4000**
**Ave Memory Occupancy 500KB**

2. Refine design to meet performance requirements.

3. Use Refinement Verification to check validity of design changes.
   - **Depth**, **Vertical**, or **Horizontal**
   - **Refinement properties**

4. Re-simulate to see if your goals are met.

**Execution time 200ms**
**Bus Cycles 1000**
**Ave Memory Occupancy 100KB**

Copyright A. Sangiovanni-Vincentelli