Metropolis - Design Environment for Electronic Systems – Overview and Architecture Modeling Proposal

Metropolis Project Team
Speaker: Douglas Densmore – UC Berkeley
Thanks to: Yosinori Watanabe (Cadence Berkeley Labs)
Outline

◆ Metropolis Big Picture
◆ What Can Be Done with Metropolis?
◆ How Are Designs Represented In Metropolis?
◆ Architecture Modeling Proposal
◆ Conclusions
Platform Based Design is composed of three aspects:

- Top Down Application Development
- Bottom Up Design Space Exploration
- Platform Development

Orthogonalization of concerns

- Functionality and Architecture
- Behavior and Performance Indices
- Computation, Communication, and Coordination.

Metropolis is a design environment implementing the concepts of Platform Based Design.
Metropolis Project Big Picture: Target and Goals

◆ Target: Embedded System Design
  ♦ Set-top boxes, cellular phones, automotive controllers, …
  ♦ Heterogeneity:
    ♦ computation: Analog, ASICs, programmable logic, DSPs, ASIPs, processors
    ♦ communication: Buses, cross-bars, cache, DMAs, SDRAM, …
    ♦ coordination: Synchronous, Asynchronous (event driven, time driven)

◆ Goals:
  ♦ Design methodologies:
    ♦ abstraction levels: design capture, mathematics for the semantics
    ♦ design tasks: cache size, address map, SW code generation, RTL generation, …
  ♦ Tool set:
    ♦ synthesis: data transfer scheduling, memory sizing, interface logic, SW/HW generation, …
    ♦ verification: property checking, static analysis of performance, equivalence checking, …
Outline

◆ What can be done in Metropolis?

A case study for a multi-media application:

- 4 levels of abstraction
- binding between adjacent levels of abstraction
- tool support for verification and synthesis

This is just one example; different methodologies developed for different application domains.
Evaluate the methodology with formal techniques applied.

• Function
  – Input: a transport stream for multi-channel video images
  – Output: a PiP video stream
    - the inner window size and frame color dynamically changeable

60 processes with 200 channels
**Multi-Media System: Abstraction Levels**

- Network of processes with sequential program for each
- Unbounded FIFOs with multi-rate read and write

- Communication refined to bounded FIFOs and shared memories with finer primitives (called TTL API):
  - allocate/release space, move data, probe space/data

- Mapped to resources with coarse service APIs
- Services annotated with performance models
- Interfaces to match the TTL API

- Cycle-accurate services and performance models
Binding Adjacent Levels of Abstraction

Example: a unbounded FIFO v.s. a bounded FIFO with the finer service.

- Implement the upper level services using the current services
- Bounded FIFO API, e.g. release space, move data
- FIFO width and length parameterized

- Metropolis represent both levels of abstraction explicitly, rather than replacing the upper level.
  - essential for specifying the refinement relation.
- Refinement relation is associated with properties to preserve through the refinement.
  - the properties can be formally specified, and verified either formally or through simulation.
Refinement Verification

Two properties checked for the refinement:
1. Deadlock
2. Data consistency:
   Always: \( \forall i: \text{Writer.data}[i] = \text{Reader.data}[i] \);

Both were verified for the refined protocol:
1. Automatically applied SPIN in Metropolis:
   - Translate the protocol description to SPIN
   - Found a deadlock in the original algorithm provided by Philips
2. Automatically applied LoC Monitor in Metropolis
   - Translate the property formula to executable code (simulation monitor)
   - Simulate the monitor together with the original processes and refined protocol
   - Found two bugs in the protocol description (a piece of the writer's data was conditionally ignored in the reader's side).
Architecture Exploration

• Configure the resources, e.g. the size of an internal memory, width of a bus.

• Bind the processes to the resources.

• Compose the resource services, e.g. schedules of data transfers, compilation of basic blocks.

Once done, performance analysis and simulation can be carried out, using the performance models associated with the resource services.

But, to do so, we need models for the resources: services and performance.
Resource Modeling

What to model/abstract depends on what you are concerned about.

Example:

- Where should my data be located?
- When should my data be transferred?
- How often do my peripherals generate interrupts?

These all influence my decisions on sizes of internal memories, bus configuration, data transfer schedules.

Resource models based on these concerns:

- Memory: internal (single-access, dual-access), external
- DSP: bus read/write(), check/serve_Interrupt(), execute()
  performance: latency per byte transfer
- DMA control, CPU: similar
- peripheral: non-deterministic interrupt generator
  could be restricted wrt supported behavior (under development)
**Architecture Exploration**

- Configure the resources, e.g. the size of an internal memory, width of a bus.
- Bind the processes to the resources.
- Compose the resource services, e.g. schedules of data transfers, compilation of basic blocks.

Performance analysis and simulation can be carried out, using the performance models associated with the resource services.

Effective scheduling of process operations mapped to a CPU is a key issue:
- reduce the context-switching between tasks for efficient execution
- increase data coherency among processes for efficient memory usage
Outline

How are designs represented in Metropolis?

Metropolis meta-model: a language + modeling mechanisms

- represents all key ingredients: function, architecture, refinement, platforms
- parser and API to browse designs, interact with tools
Meta-model : function netlist

MyFncNetlist

process P{
    port reader X;
    port writer Y;
    thread()
    while(true){
        ...
        z = f(X.read());
        Y.write(z);
    }
}

interface reader extends Port{
    update int read();
    eval int n();
}

interface writer extends Port{
    update void write(int i);
    eval int space();
}

medium M implements reader, writer{
    int storage;
    int n, space;
    void write(int z){
        await(space>0; this.writer ; this.writer)
        n=1; space=0; storage=z;
    }
    word read(){ ... }
}
Meta-model: execution semantics

◆ Processes take *actions*.
  ◆ statements and some expressions, e.g. 
    \[ y = z + \text{port.f()};, \quad \text{z+port.f()}, \quad \text{port.f()}, \quad i < 10, \quad \ldots \]

◆ 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.
**Meta-model: architecture components**

An architecture component specifies services, i.e.

- what it *can* do : interfaces
- how much it *costs* : quantities, annotation, logic of constraints

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

medium Bus implements BusMasterService {...
    port BusArbiterService Arb;
    port MemService Mem; ...
    update void busRead(String dest, int size) {
        if(dest== ... ) Mem.memRead(size);
        [[Arb.request(B(thisthread, this.busRead));
        GTime.request(B(thisthread, this.memRead),
        BUSCLKCYCLE +
        GTime.A(B(thisthread, this.busRead)));
    }
}
...
```

```java
interface BusArbiterService extends Port {
    update void request(event e);
    update void resolve();
}

scheduler BusArbiter extends Quantity
    implements BusArbiterService {
    update void request(event e){ ... } 
    update void resolve() { //schedule }
}
```
**Meta-model: architecture netlist**

Architecture netlist specifies configurations of architecture components.

Each constructor

- instantiates arch. components,
- connects them,
- takes as input *mapping processes*.

![Diagram of architecture netlist]

- **CPU + OS**: Master
- **Bus**
- **Mem**
- **Arbiter**
- **Slave**
- **MyArchNetlist**
- **OsSched**
- **Bus Arbiter**
**Meta-model: mapping processes**

---

**Function process**

```plaintext
process P{
    port reader X;
    port writer Y;
    thread(){
        while(true){
            ...
            z = f(X.read());
            Y.write(z);
        }
    }
}
```

**Mapping process**

```plaintext
process MapP{
    port CpuService Cpu;
    void readCpu(){
        Cpu.exec();   Cpu.cpuRead();
    }
    void mapf(){ ...
    ...
    thread(){
        while(true){
            await {
                (true; ; ;) readCpu();
                (true; ; ;) mapf();
                (true; ; ;) readWrite();
            }
        }
    }
}
```

\[ B(P, X.read) \iff B(MapP, readCpu); \]
\[ E(P, X.read) \iff E(MapP, readCpu); \]
\[ B(P, f) \iff B(MapP, mapf); \]
\[ E(P, f) \iff E(MapP, mapf); \]

...
Meta-model: mapping netlist

MyMapNetlist

\[
\begin{align*}
B(P1, M.write) & \leftrightarrow B(mP1, mP1.writeCpu); & E(P1, M.write) & \leftrightarrow E(mP1, mP1.writeCpu); \\
B(P1, P1.f) & \leftrightarrow B(mP1, mP1.mapf); & E(P1, P1.f) & \leftrightarrow E(mP1, mP1.mapf); \\
B(P2, M.read) & \leftrightarrow B(P2, mP2.readCpu); & E(P2, M.read) & \leftrightarrow E(mP2, mP2.readCpu); \\
B(P2, P2.f) & \leftrightarrow B(mP2, mP2.mapf); & E(P2, P2.f) & \leftrightarrow E(mP2, mP2.mapf); 
\end{align*}
\]
interface MyService extends Port {
    int myService(int d);
}

medium AbsM implements MyService {
    int myService(int d) {
    ...
    }
}

refine(AbsM, MyMapNetlist);

refine(AbsM, MyMapNetlist1);

refine(AbsM, MyMapNetlist2);

refine(AbsM, MyArchNetlist);

refine(AbsM, MyFncNetlist);

MyMapNetlist1

MyMapNetlist2

MyArchNetlist

MyFncNetlist

B(P1, M.write) <=> B(mP1, mP1.writeCpu);
B(P1, P1.f) <=> B(mP1, mP1.mapf);
E(P1, P1.f) <=> E(mP1, mP1.mapf);
B(P2, M.read) <=> B(mP2, mP2.readCpu);
B(P2, P2.f) <=> B(mP2, mP2.mapf);

B(P1, M.write) <=> B(mP1, mP1.writeCpu);
B(P1, P1.f) <=> B(mP1, mP1.mapf);
E(P1, P1.f) <=> E(mP1, mP1.mapf);
B(P2, M.read) <=> B(mP2, mP2.readCpu);
E(P2, P2.f) <=> E(mP2, mP2.mapf);

B(P1, M.write) <=> B(mP1, mP1.writeCpu);
B(P1, P1.f) <=> B(mP1, mP1.mapf);
E(P1, P1.f) <=> E(mP1, mP1.mapf);
B(P2, M.read) <=> B(mP2, mP2.readCpu);
E(P2, P2.f) <=> E(mP2, mP2.mapf);
Meta-model: platforms

A set of mapping netlists, together with constraints on event relations to a given interface implementation, constitutes a platform of the interface.
Meta-model: recursive paradigm of platforms

\[
\begin{align*}
B(Q_2, S.cdx) & \iff B(Q_2, mQ_2.excCpu); \\
E(Q_2, M.cdx) & \iff E(mQ_2, mQ_2.excCpu); \\
B(Q_2, Q_2.f) & \iff B(mQ_2, mQ_2.mapf); \\
E(Q_2, P_2.f) & \iff E(mQ_2, mQ_2.mapf); \\
\end{align*}
\]
Metropolis design environment

- Load designs
- Browse designs
- Relate designs (refine, map etc)
- Invoke tools
- Analyze results

Metropolis interactive Shell

Meta model language

Meta model compiler

Front end

Abstract syntax trees

Back end\textsubscript{1}
Back end\textsubscript{2}
Back end\textsubscript{3}
...Back end\textsubscript{N}

Simulator tool
Synthesis tool
Verification tool
Verification tool
Outline of this talk

Architecture and Resource Modeling Proposal

- What architecture models are needed and why Xilinx
- What resources models are needed
- Potential modeling strategies
- Architecture modeling as refinement
- Tying it all together
Need Models

As mentioned, keys to the Metropolis design methodology are:
- Architecture Exploration
- Resource Modeling

Need non-trivial set of architectures
- Different topologies
- Different services

Explore different granularities of architectural services
- Read/write v.s. stb/lmw
- Execute
Why FPGA

◆ Current FPGA platforms are ideal architecture platforms
  ◆ Can realize many different designs quickly
  ◆ Heterogeneous components
  ◆ Various granularities
    ♦ IP blocks, embedded processor cores, CLBs
  ◆ Well established tool flow

◆ Key is to establish diverse model set allowed by reconfigurability.
  ◆ Spatial vs. Temporal computation models
  ◆ Bit-Level vs. High Level models
  ◆ IP Block vs. custom models
  ◆ Static vs. Dynamic models
Resource Models Needed

◆ Computation Elements
  • These should provide services associated with computation
  • PowerPC, MicroBlaze

◆ Communication Elements
  • These should provide services associated with communication
  • CoreConnect Bus, Embedded Memory elements

◆ Quantity Annotation
  • Identify appropriate performance annotation information.
    ♦ Power, throughput, latency, cycle count, computational density, etc.

◆ Ideally establish a common set of interfaces so that combinations of these elements is possible
Modeling Strategies

MyArchNetlist

Cpu

OsSched

Bus

Mem

SelectRAM+

Off Chip RAM
**Architecture Modeling as Refinement**

- Various stages of development result in various architecture models
  - Initial models tend to be abstract while later models are typically considered refinements

- Models which correspond to refinements of abstract models should be able to be substituted into the system and maintain correct functionality of the overall system.
  - Various properties held by the abstract model should be preserved in refinement

- Platform Based Design aided by various architecture targets for the “bottom up” portion of the design phase where various architecture instances are considered for platform mapping and performance estimation.
PiP Architecture
Vertical Refinement

◆ Design may call for the introduction of new services; ASICs, Dedicated Memory, Peripherals
  ♦ These are currently media with associated schedulers

◆ The services will be added “vertically” into the scheduled netlist
  ♦ Existing services not interacting with new services remain unmodified
  ♦ Existing services interacting with new services require port changes and changes to the services they provide in order to reflect the new hierarchy
    ♦ Main effort in this area

◆ The service schedulers will be added into the scheduling netlist
  ♦ One-to-one correspondence between new services and new schedulers

◆ New objects must be instantiated in the top level netlists and connections made to reflect the new topology.
Vertical Refinement
**Horizontal Refinement**

- Designer can introduce more scheduling service type implementations in the scheduled netlist
  - Would like to move work from the more “virtual” scheduling netlist elements into the more service oriented scheduled netlist
- This is a “Horizontal” movement of quantity managers into the scheduled netlist and a conversion of their functionality into media services.
  - Tasks services are now first requested to the schedulers which then request actual services from original media
  - Original media now interact with global time quantity manager directly.
Horizontal Refinement

Scheduled Netlist

Task1

TaskN

CPU RTOS

Bus

Mem

CPU Service Scheduler

Bus Service Scheduler

Mem Service Scheduler

Scheduling Netlist

Global Time

CpuScheduler

BusScheduler

MemScheduler
**Depth Refinement, Hybrid Refinement**

**深度细化、混合细化**

**深度细化**
- 这是指内部过程的变化
  - 内部数据结构
    - 原子（数组，类）
  - 函数定义
    - 函数内部工作
  - 函数调用序列
    - 线程体

**混合细化**
- 自然地您可以结合这些细化技术

**关键将是识别细化方法的优缺点**

- 属性与风格相关
  - 变更您可以获得所需的效应
  - 细化粒度
    - 您正在工作的抽象级别是什么，可用服务是什么？

### 例子

<table>
<thead>
<tr>
<th>Abstract</th>
<th>Refinement</th>
</tr>
</thead>
<tbody>
<tr>
<td>Check()</td>
<td>Read()</td>
</tr>
<tr>
<td>Read()</td>
<td>Write()</td>
</tr>
</tbody>
</table>

**Abstract**
- Check()
- Read()
- Check()
- Write()

**Refinement**
- Read()
- Write()
Tying it all together: Modeling

- Model several “CPU” type resources
- Model several “Bus” type resources
- Model several “Mem” type resources
- Determine appropriate type and granularity of services provided
- Put through the “PiP application flow”
**Tying it all together: Refinement**

- Create Baseline Xilinx architecture
- Add/Remove services
- Introduce new scheduling protocols
- Key is to identify properties and behaviors in each refinement
  - Which are maintained?
  - Which are transformed?
  - Consistency? Predictability?
  - Methodology Recommendations?
Metropolis Summary

• Concurrent specification with a formal execution semantics
• Feasible executions of a netlist: sequences of event vectors
• Quantities can be defined and annotated with events, e.g.
  - time, power, global indices, composite quantities.
• Concurrent events can be coordinated in terms of quantities:
  - logic can be used to define the coordination,
  - algorithms can be used to implement the coordination.
• The mechanism of event coordination wrt quantities plays a key role:
  - architecture modeling as service with cost,
  - a mapping coordinates executions of function and architecture netlists,
  - a refinement through event coordination provides a platform.
• Metropolis design environment:
  - meta-model compiler to provide an API to browse designs,
  - backend tools to analyze designs and produce appropriate models.
Modeling Proposal Summary

- Determine appropriate architecture services and resources for the Xilinx Virtex II family of devices
  - Computation, Communication, Coordination
  - Various granularities and interfaces

- Create Metropolis models

- Create native Virtex II Pro PiP application

- Run models through PiP Architecture Flow
  - Compare results to native implementation
  - Refine models to better reflect implementation while still preserving useful abstraction level