Metamodels

Basics (see papers)

Two approaches to metamodeling:

- Traditional
  - Modeling of modeling languages including the languages'concrete syntax (notations), abstract syntax, and semantics.
  - Metamodels determine the set of valid models that can be defined with models' language and behavior in a particular domain.
  - Generic functions in model-based design such as model building, model transformation, and model management are supported by metaprogrammable tools.
  - The tools' core functions are independent from the particular DSMLs and can be instantiated using metamodels.
- Models of Computation-based

Why is Metamodeling Important?

- **Advantages of Domain-Specific Modeling**
  - **Familiar, relevant modeling concepts, relationships, and presentation**
  - **Customized modeling constraints**
  - **Tailored Scope**
  - **Custom analysis capabilities and system artifact generation**
  - **Correctness-by-construction**
  - **"The right tool for the job"**

- **BUT, it is expensive and time-consuming to create new modeling languages and tools from scratch!**
  - **E.G., a custom modeling environment for co-designing the hardware and software for a specific type of missile**
  - **E.G., a custom modeling environment for documenting the architecture of one particular system**

How Does Metamodeling Help?

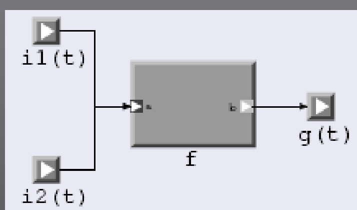
- **Metamodeling Language**: A modeling language used to specify other modeling languages
- Applies the benefits of domain-specific modeling to the design of modeling languages
  - Concepts and relationships key to specifying language syntax
  - Constraints prevent users from building "non-sensical" languages
  - Domain-specific modeling environments can be automatically generated from metamodels
  - APIs for parsing, querying, and manipulating models can be generated as well
  - Metamodels can easily be revised to update the language as program needs change

What is a Metamodel?

- **A model of the syntax of a modeling language**
  - **Formal language specification artifact**
  - **Domain concepts**
  - **Domain relationships**
  - **Domain-specific visualizations**
  - **Domain-specific system design constraints**
- **Analogy: A metamodel is to a graphical modeling language what a BNE Grammar is to a textual language.**
- **Terminology: GME uses *metamodels* to generate *paradigms*, which configure GME into a domain-specific modeling environment (*DSME*).**

Model-Integrated Computing

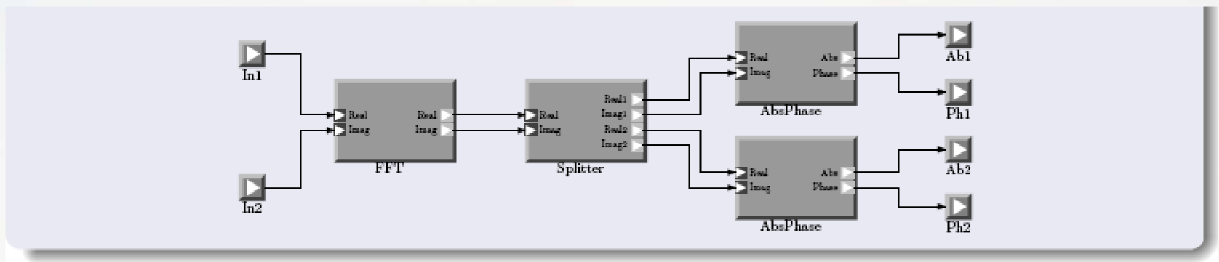
The structural semantics excludes semantically meaningless models.



No operator was provided for composition of values, so this merge model is semantically meaningless in this domain.

Key Idea: Capture intrinsic domain concepts with *domain-specific modeling languages* (DSML-s) and partition DSML-s into *structural* and *behavioral* semantics.

- The structural semantics views a model as a structure, and provides a means for calculating which structures are well-formed.



- The behavioral semantics defines what the structures do.

1. A block f represents an n-ary map over some value domain. $f : \mathcal{V}^n \to \mathcal{V}^m$.
2. A connection c represents an projection operator: $\pi_{i,m} : \mathcal{V}^m \to \mathcal{V}$, where $\pi_{i,m}(v_0, v_1, \ldots, v_{m-1}) \mapsto v_i$
3. composition is by function composition: $splitter(\pi_{0,2} \circ fft(in1(t), in2(t)), \pi_{1,2} \circ fft(in1(t), in2(t)))$.

Specification of Structural Semantics of DSML-s

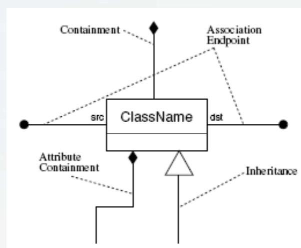
Abstract syntax of DSML-s are defined by metamodels. Metamodeling languages provide structural semantics.

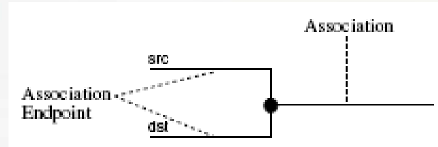
A metamodeling language is one of the DSML-s: the same tool can be used for modeling and metamodeling.

- Metamodels define the structural semantics of DSML-s:

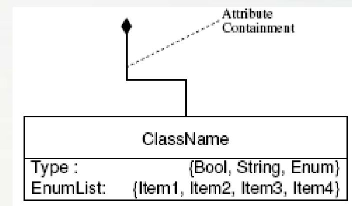
$$L = \langle Y, R_Y, C, ([\;]_i)_{i \in J} \rangle$$
$$D(Y, C) = \{ r \in R_Y \mid r \models C \}$$

- GME, the metaprogrammable modeling tool of ISIS, supports rapid construction of metamodels and DSML models.

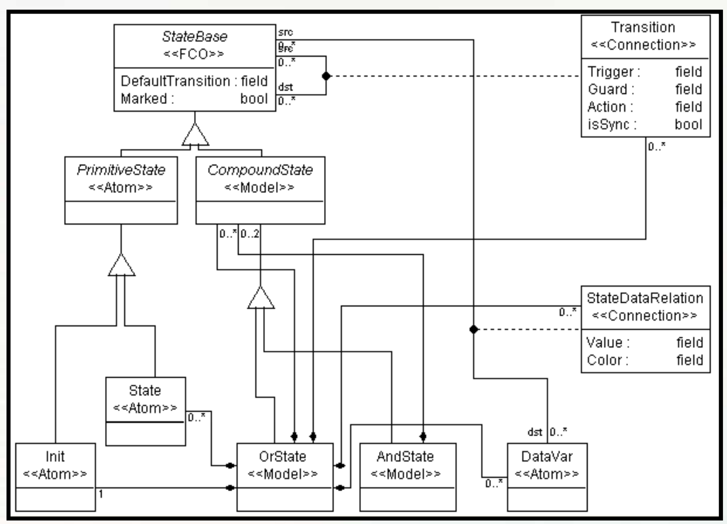




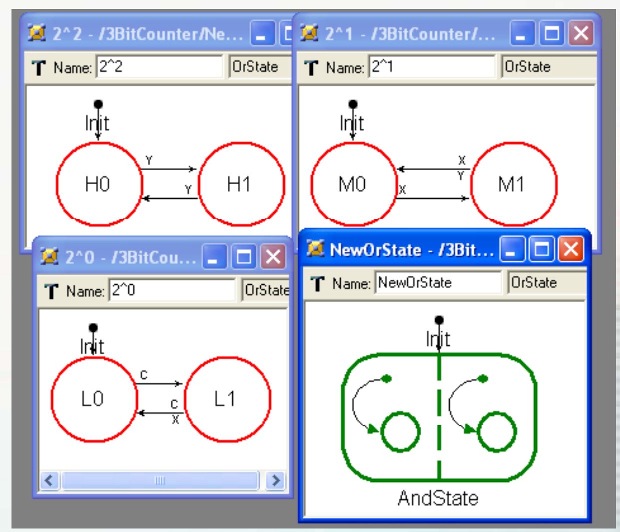
OCL Constraints:
```
self.transTo->forAll(s | s <> self)
```



Basic metamodeling notation: UML Class Diagram + OCL



MetaGME metamodel of simple statecharts



Model-editor generated from metamodel

Specification of Behavioral Semantics of DSML-s

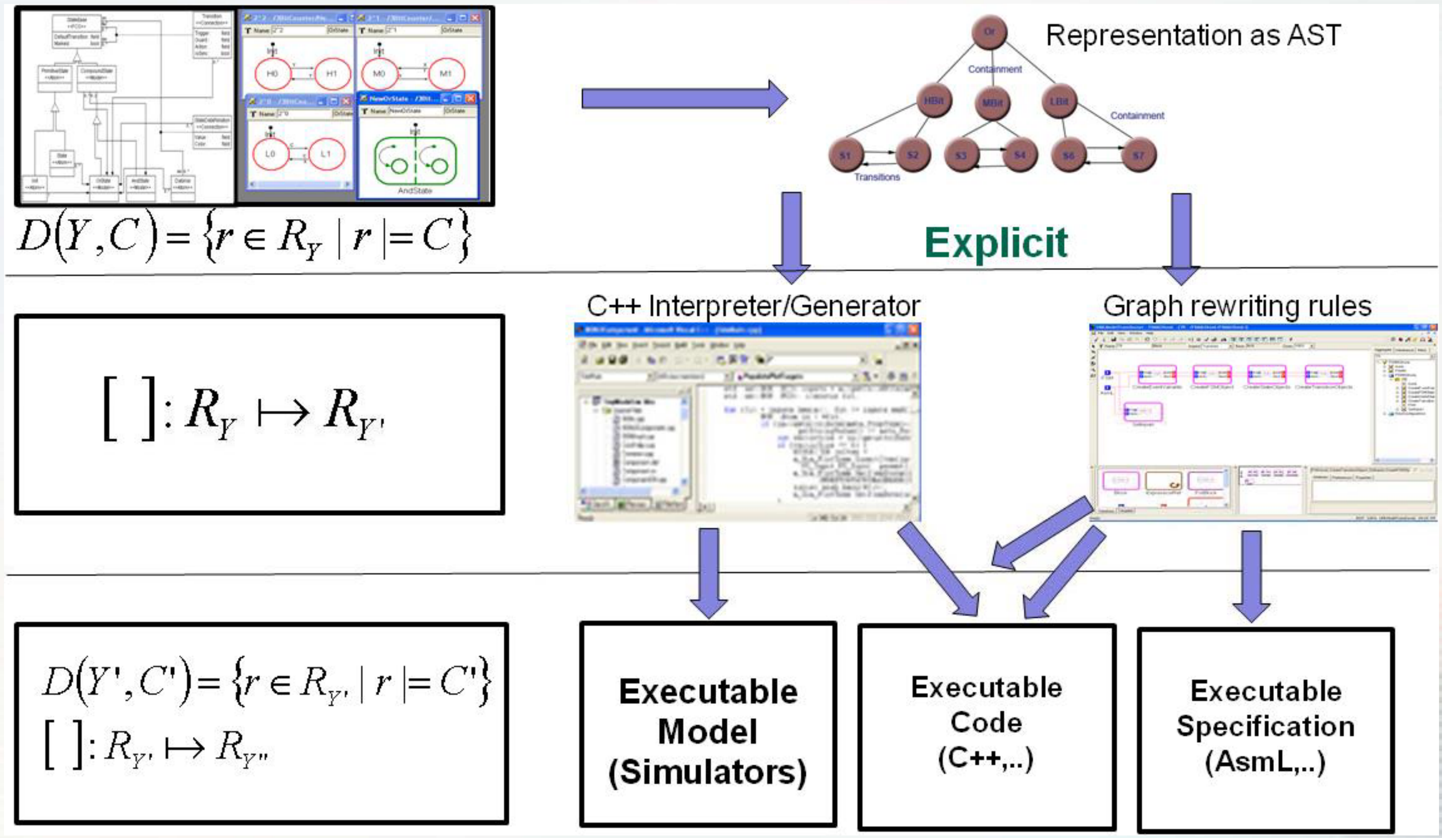
C++ coding permits complex behavioral semantics, but the "specifications" are cluttered with C++ details.

Graph transformations provide a transparent mechanism to attach semantics. However, not all behavioral semantics ca be specified this way.

Semantic anchoring with ASM captures the best of both worlds: Simple graph transformations and simple behavioral specifications.

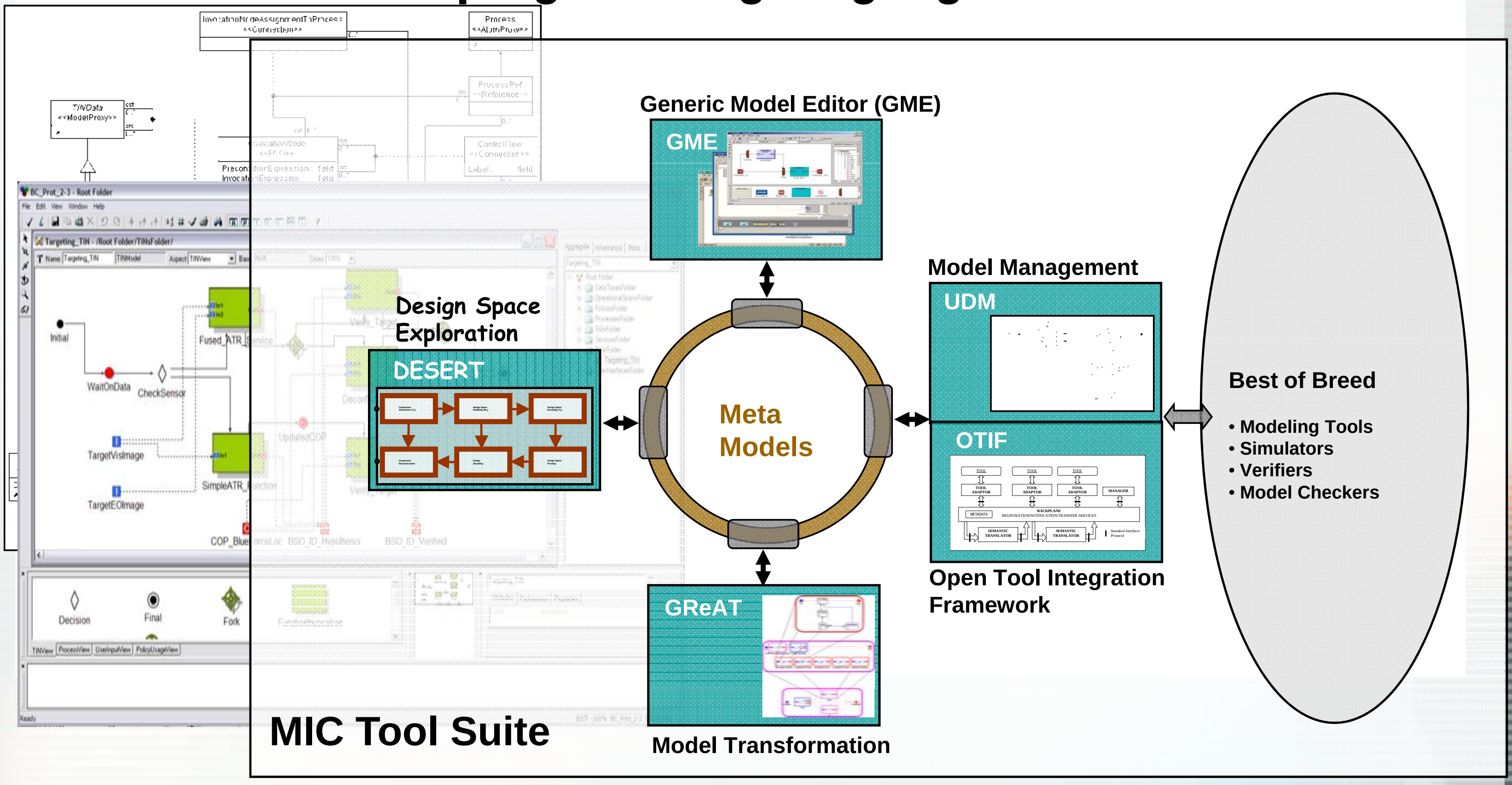
- Behavioral semantics are defined with model transformations and semantic anchoring.

$$[\![\ ]\!]^T : R_Y \mapsto R_{Y'}$$



Metaprogrammable Tools

- Model-based development is practical!
- Domain specific abstractions are not only desirable; they are affordable
- DSML-s are not programming languages

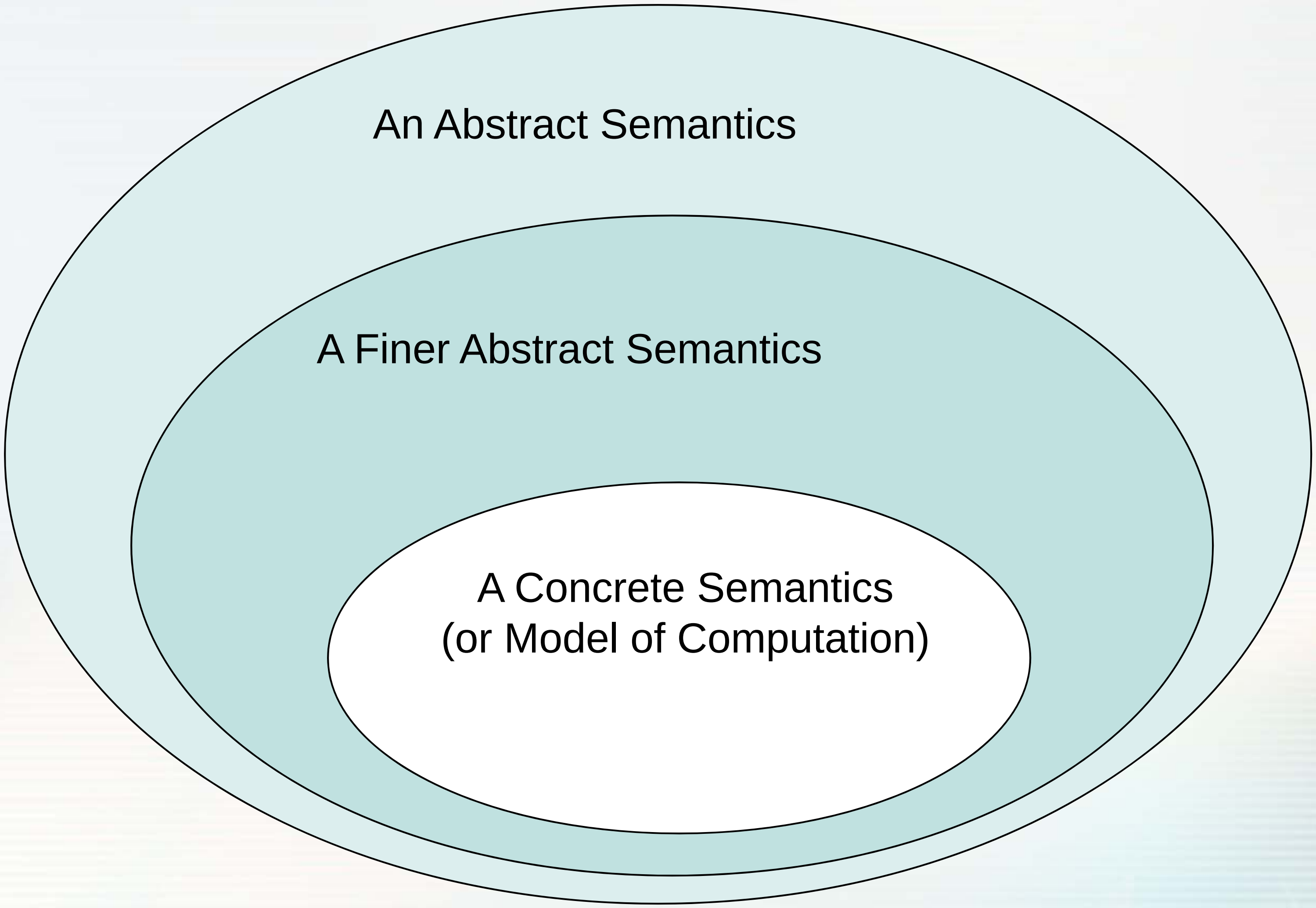


Semantics Metamodel

- MoCs are powerful in capturing specific designs, embedded electronic systems are inherently heterogeneous.
- Modeling requires multiple MoC-specific models, thus making the overall system's analysis problematic because its behavior is not a priori expressible in a mathematical formalism that can be inferred from the components' MoCs.
- Semantics Metamodeling is a way to uniformly abstract away MoC specificities while consolidating MoC commonalities in the semantics metamodel.
- It results in a mechanism to analyze and design complex systems without renouncing the properties of the components' MoCs.

Metropolis Metamodel

Where We Are Headed



Tagged Signal Abstract Semantics

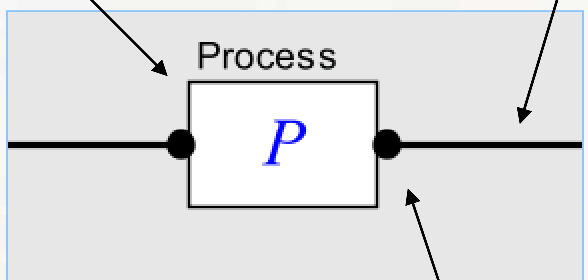
Tagged Signal Abstract Semantics:

a “process” is a subset of the signals with which it interacts.

signal is a member of a set of signals, where the set depends on the model of computation and resolved data type of the connection.

$$P \subset S_1 \times S_2$$

$s_2 \in S_2$



$s_1 \in S_1$

port may be an input or an output, or neither or both. It is irrelevant.

This outlines a general *abstract semantics* that gets specialized. When it becomes concrete you have a
model of computation.

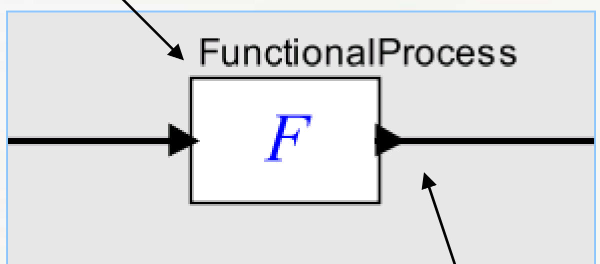
A Finer Abstraction Semantics

Functional Abstract Semantics:

a process is now a function from
input signals to output signals.

$$F : S_1 \to S_2$$

$s_2 \in S_2$



$s_1 \in S_1$

port is now either an
input or an output (or both).

This outlines an *abstract semantics* for deterministic producer/consumer actors.

Uses for Such an Abstract Semantics

- Give structure to the sets of signals
  - e.g. Use the Cantor metric to get a metric space.
- Give structure to the functional processes
  - e.g. Contraction maps on the Cantor metric space.
- Develop static analysis techniques
  - e.g. Conditions under which a hybrid systems is provably non-Zeno.

Another Finer Abstract Semantics

Process Networks Abstract Semantics:

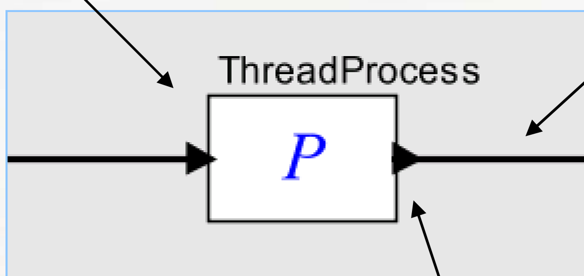
A process is a sequence of operations on its signals where the operations are the associative operation of a *monoid*

sets of signals are *monoids*, which allows us to incrementally construct them. E.g.

- stream
- event sequence
- rendezvous points …

$$P \subset S_1 \times S_2$$

$s_2 \in S_2$



$s_1 \in S_1$

process is not necessarily functional (can be nondeterministic).

port is now either an input or an output or both.

This outlines an abstract semantics for actors constructed as processes that incrementally read and write port data.

Concrete Semantics that Conform with the Process Networks Abstract Semantics

- Communicating Sequential Processes (CSP) [Hoare]
- Calculus of Concurrent Systems (CCS) [Milner]
- Kahn Process Networks (KPN) [Kahn]
- Nondeterministic extensions of KPN [Various]
- Actors [Hewitt]

Some Implementations:

- Occam, Lucid, and Ada languages
- Ptolemy Classic and Ptolemy II (PN and CSP domains)
- System C
- Metropolis

A Finer Abstract Semantics

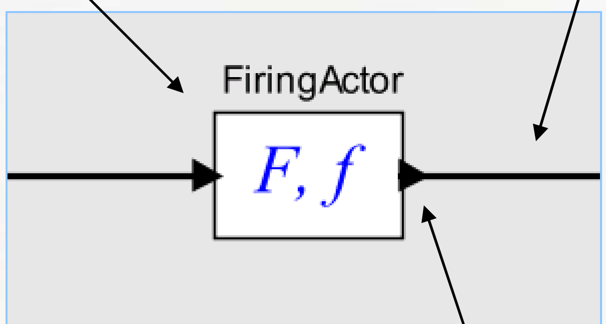
Firing Abstract Semantics:

a process still a function from input signals to output signals, but that function now is defined in terms of a firing function.

signals are monoids (can be incrementally constructed) (e.g. streams, discrete-event signals).

$$F : S_1 \to S_2$$

$s_2 \in S_2$



$s_1 \in S_1$

port is still either an input or an output.

The process function *F* is the least fixed point of a functional defined in terms of *f*.

Models of Computation that Conform to the Firing Abstract Semantics

- **Dataflow models (all variations)**
- **Discrete-event models**

In Ptolemy II, actors written to the *firing abstract semantics* can be used with directors that conform only to the process network abstract semantics.

Such actors are said to be *behaviorally polymorphic*.

A Still Finer Abstract Semantics

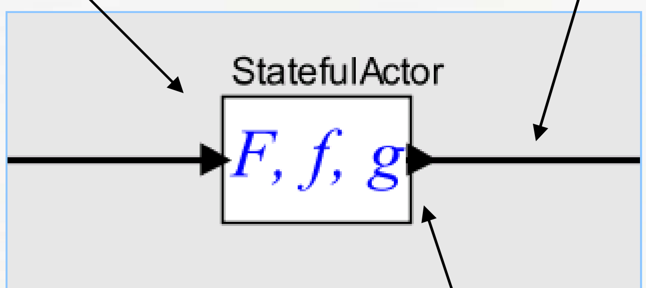
Stateful Firing Abstract Semantics:

a process still a function from input signals to output signals, but that function now is defined in terms of two functions.

signals are monoids (can be incrementally constructed) (e.g. streams, discrete-event signals).

$$F : S_1 \to S_2$$

$s_2 \in S_2$



$s_1 \in S_1$

$$f : S_1 \times \Sigma \to S_2$$

$$g : S_1 \times \Sigma \to \Sigma$$

state space

port is still either an input or an output.

The function *f* gives outputs in terms of inputs and the current state. The function *g* updates the state.

Models of Computation that Conform to the Stateful Firing Abstract Semantics

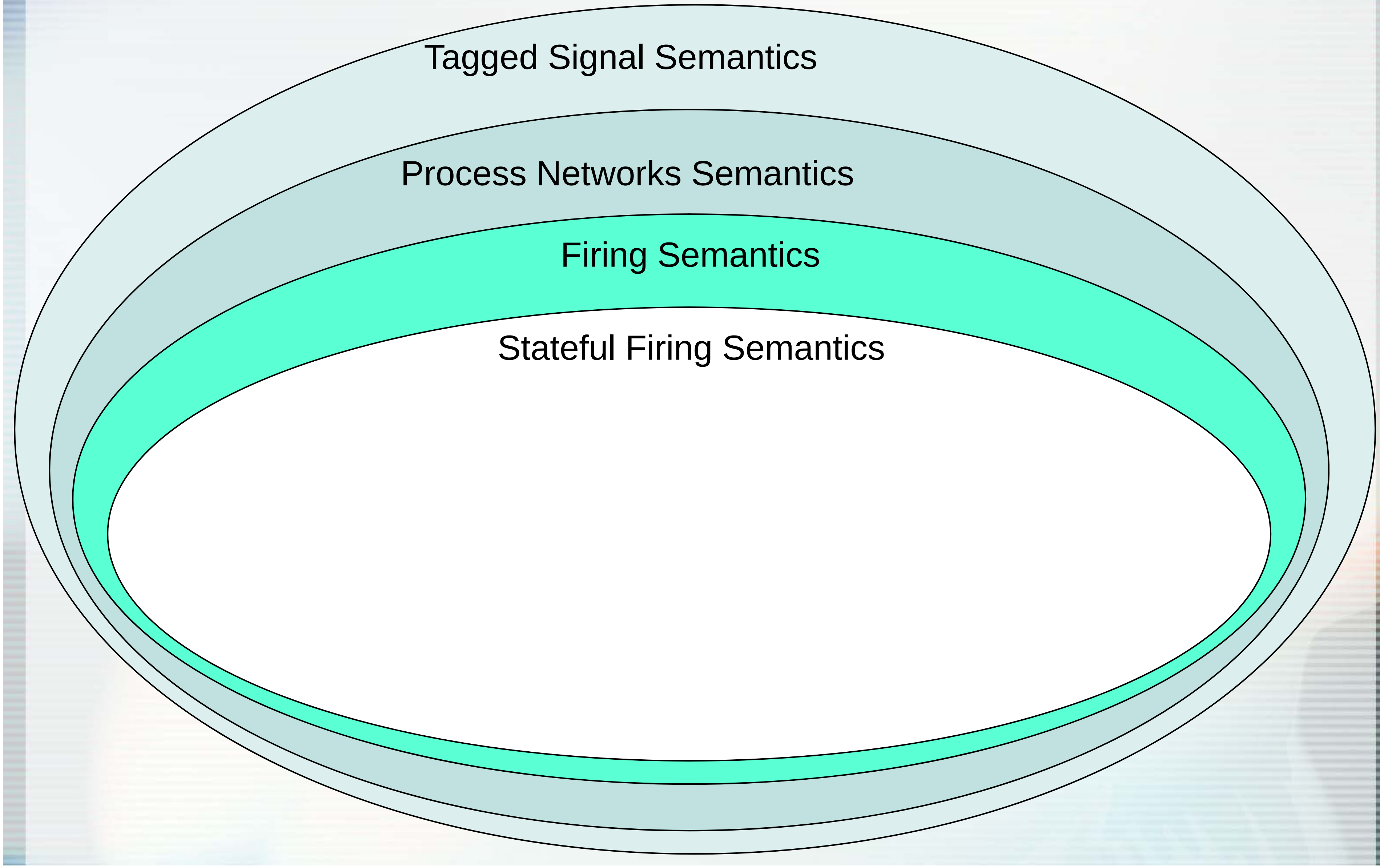
- Synchronous reactive
- Continuous time
- Hybrid systems

Stateful firing supports iteration to a fixed point, which is required for hybrid systems modeling.

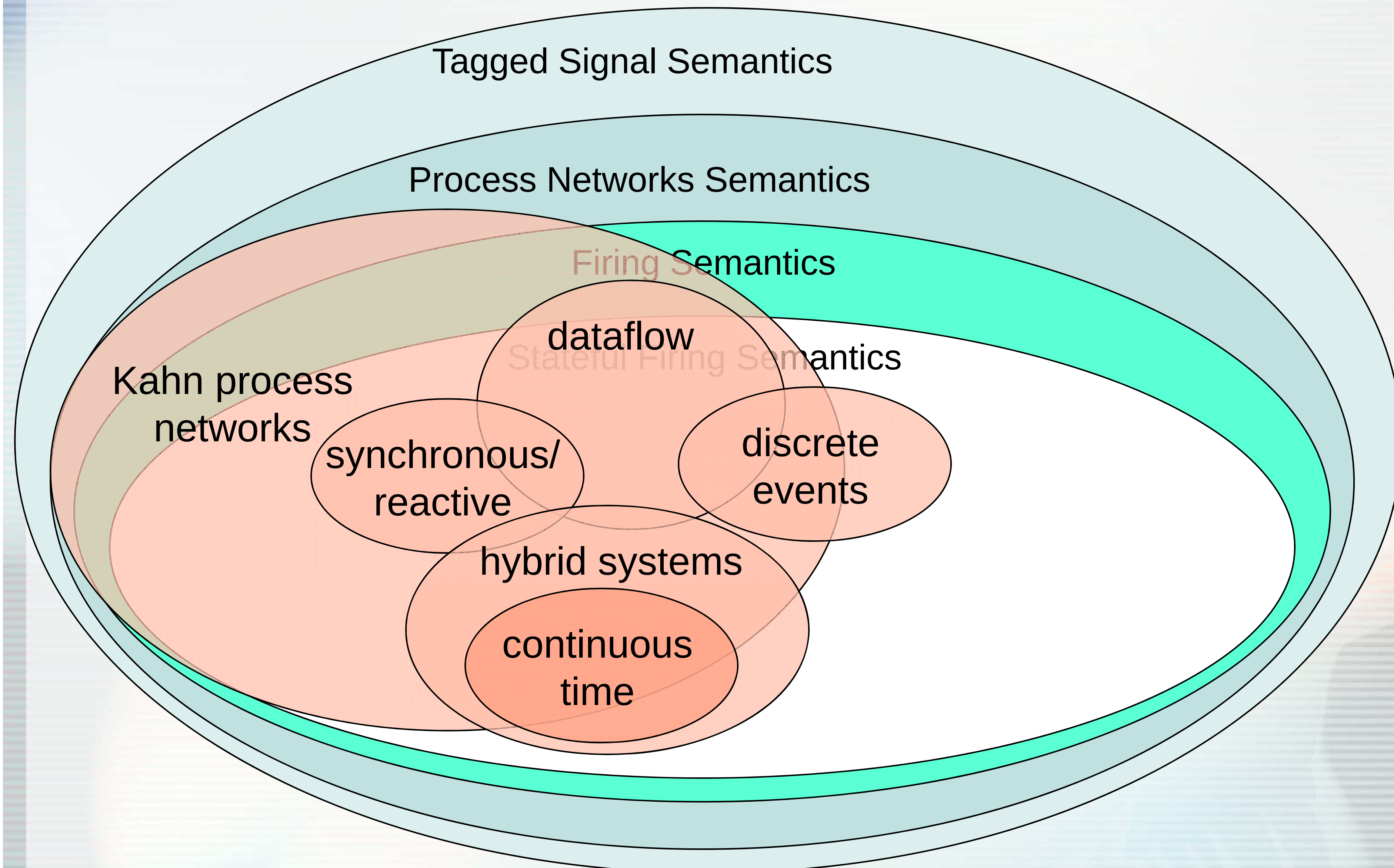
In Ptolemy II, actors written to the stateful firing abstract semantics can be used with directors that conform only to the firing abstract semantics or to the process network abstract semantics.

Such actors are said to be *behaviorally polymorphic*.

Where We Are

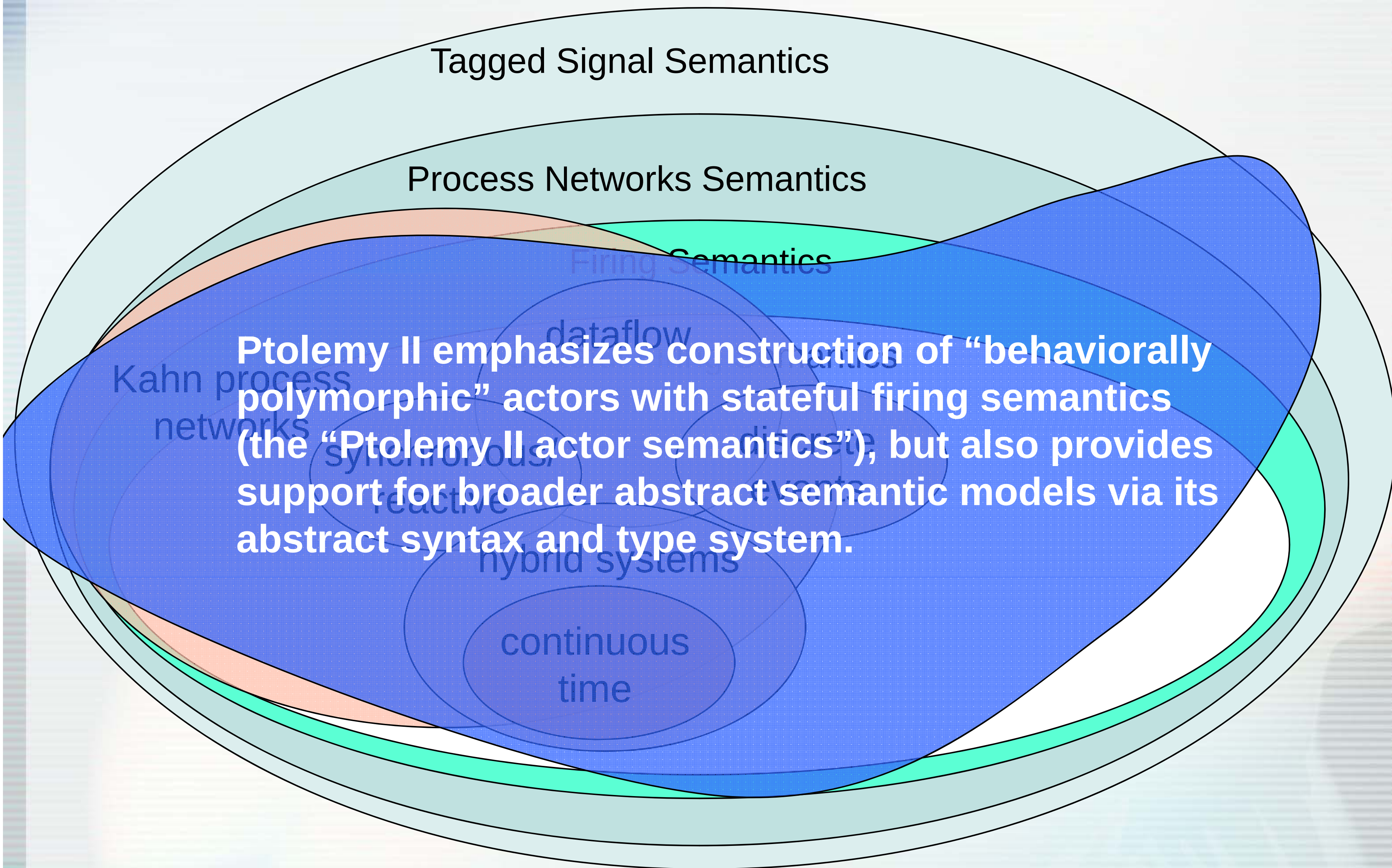

Tagged Signal Semantics
Process Networks Semantics
Firing Semantics
Stateful Firing Semantics

Where We Are



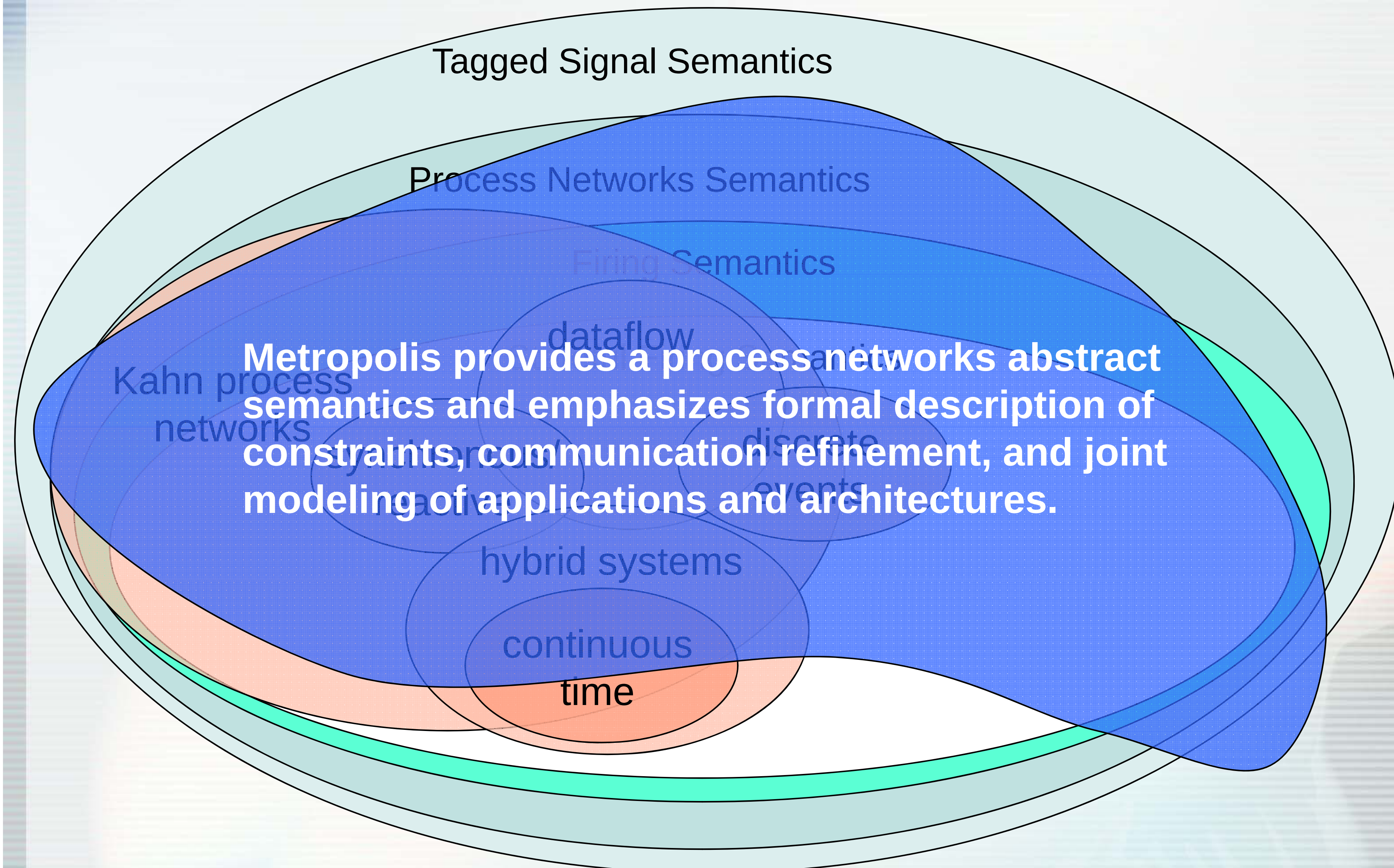
Tagged Signal Semantics
Process Networks Semantics
Firing Semantics
Stateful Firing Semantics
Kahn process networks
dataflow
synchronous/ reactive
discrete events
hybrid systems
continuous time

Meta Frameworks: Ptolemy II



Ptolemy II emphasizes construction of “behaviorally polymorphic” actors with stateful firing semantics (the “Ptolemy II actor semantics”), but also provides support for broader abstract semantic models via its abstract syntax and type system.

Meta Frameworks: Metropolis



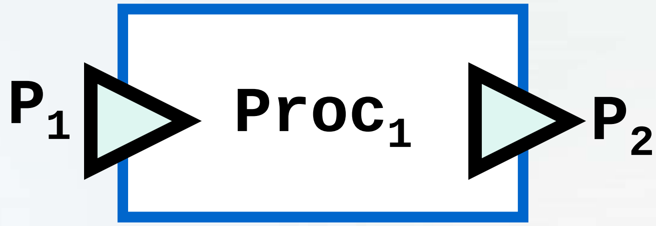
Metropolis provides a process networks abstract semantics and emphasizes formal description of constraints, communication refinement, and joint modeling of applications and architectures.

Metropolis Metamodel

Metropolis Objects

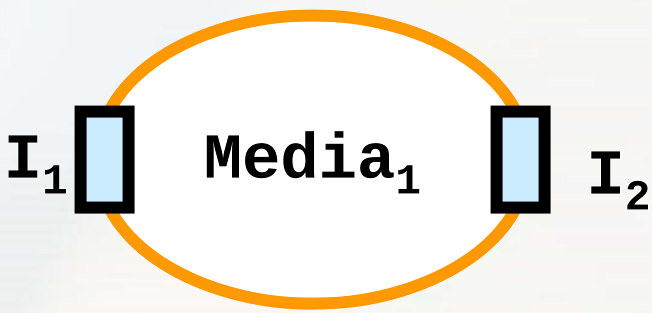
- Metropolis elements adhere to a “separation of concerns” point of view.

- Processes (Computation)



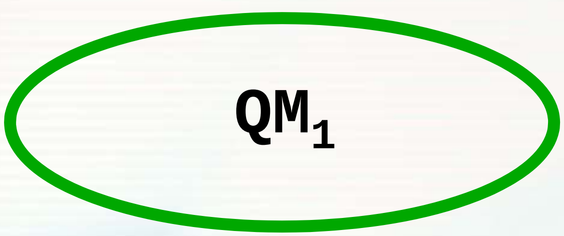
Active Objects
Sequential Executing Thread

- Media (Communication)



Passive Objects
Implement Interface Services

- Quantity Managers (Coordination)



Schedule access to
resources and quantities

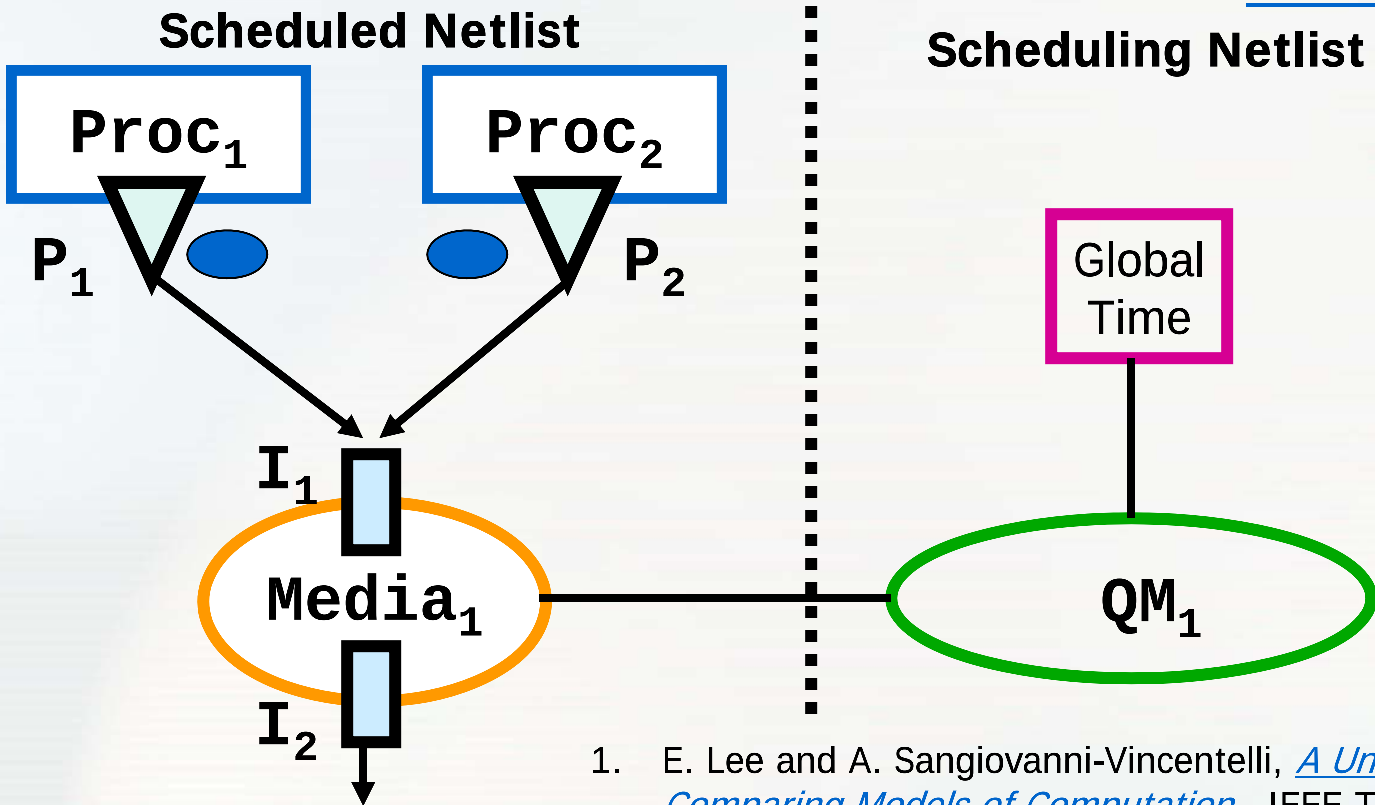
Metro. Netlists and Events

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**.

Related Work



Event[1] – represents a transition in the action automata of an object. Can be annotated with any number of quantities. This allows performance estimation.

1. E. Lee and A. Sangiovanni-Vincentelli, *A Unified Framework for Comparing Models of Computation*, IEEE Trans. on Computer Aided Design of Integrated Circuits and Systems, Vol. 17, N. 12, pg. 1217-1229, December 1998

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

1. E. Lee and A. Sangiovanni-Vincentelli, *A Unified Framework for Comparing Models of Computation*, IEEE Trans. on Computer Aided Design of Integrated Circuits and Systems, Vol. 17, N. 12, pg. 1217-1229, December 1998

Action Automata

- Processes take *actions*.
  - statements and some expressions, e.g.
    y = z+port.f();, z+port.f(), port.f(), i < 10, …
  - 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(port.f()),
    the end of an action, e.g. E(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.

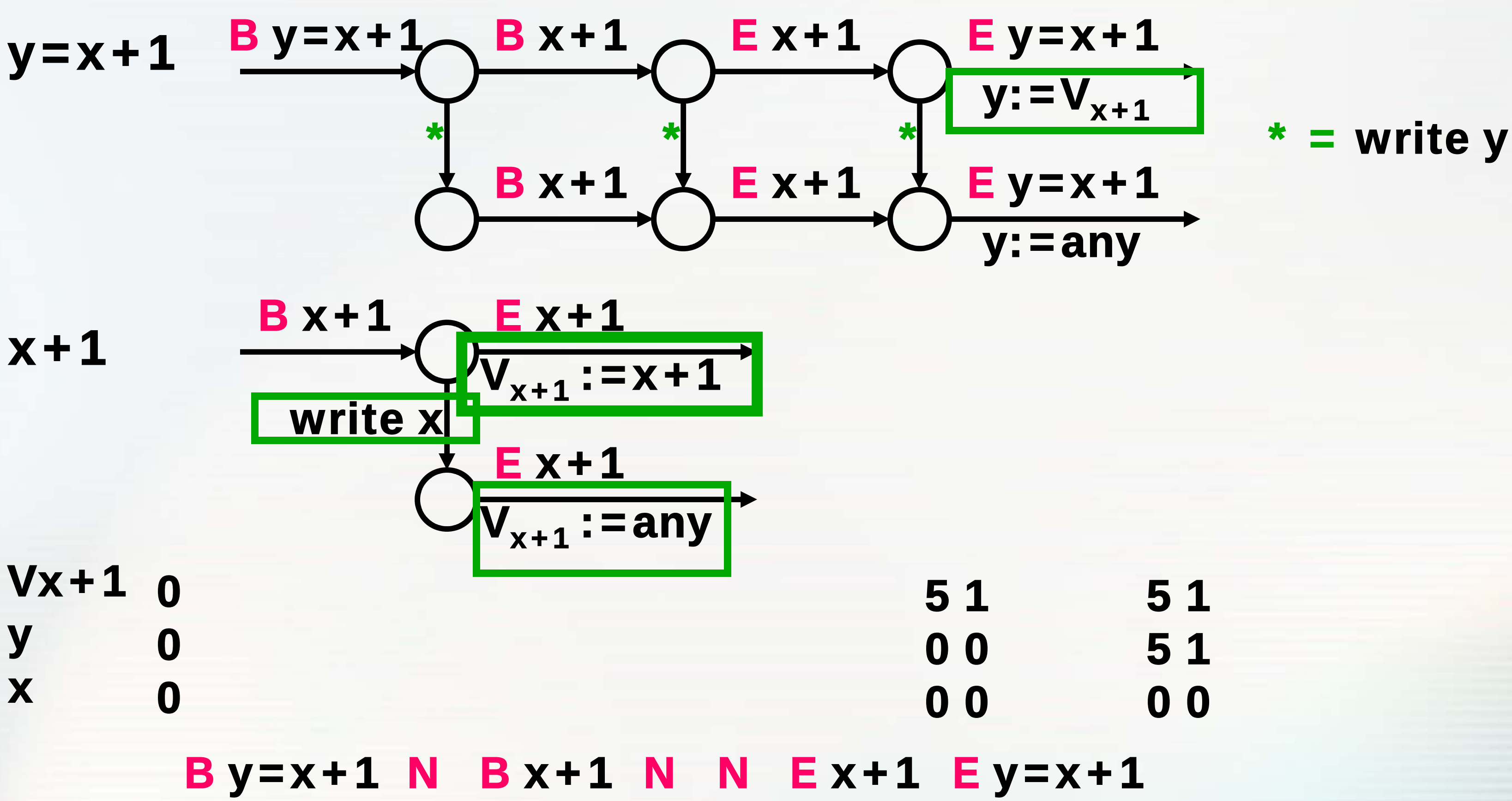
Execution semantics

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.

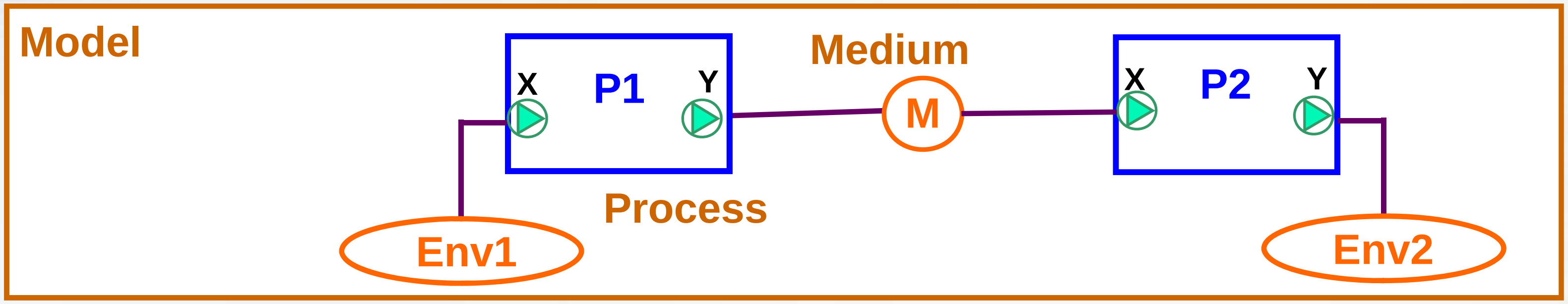
Action Automata

- y=x+1;



Return

Process Network Abstract Semantics in Metropolis




```
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();
}
```

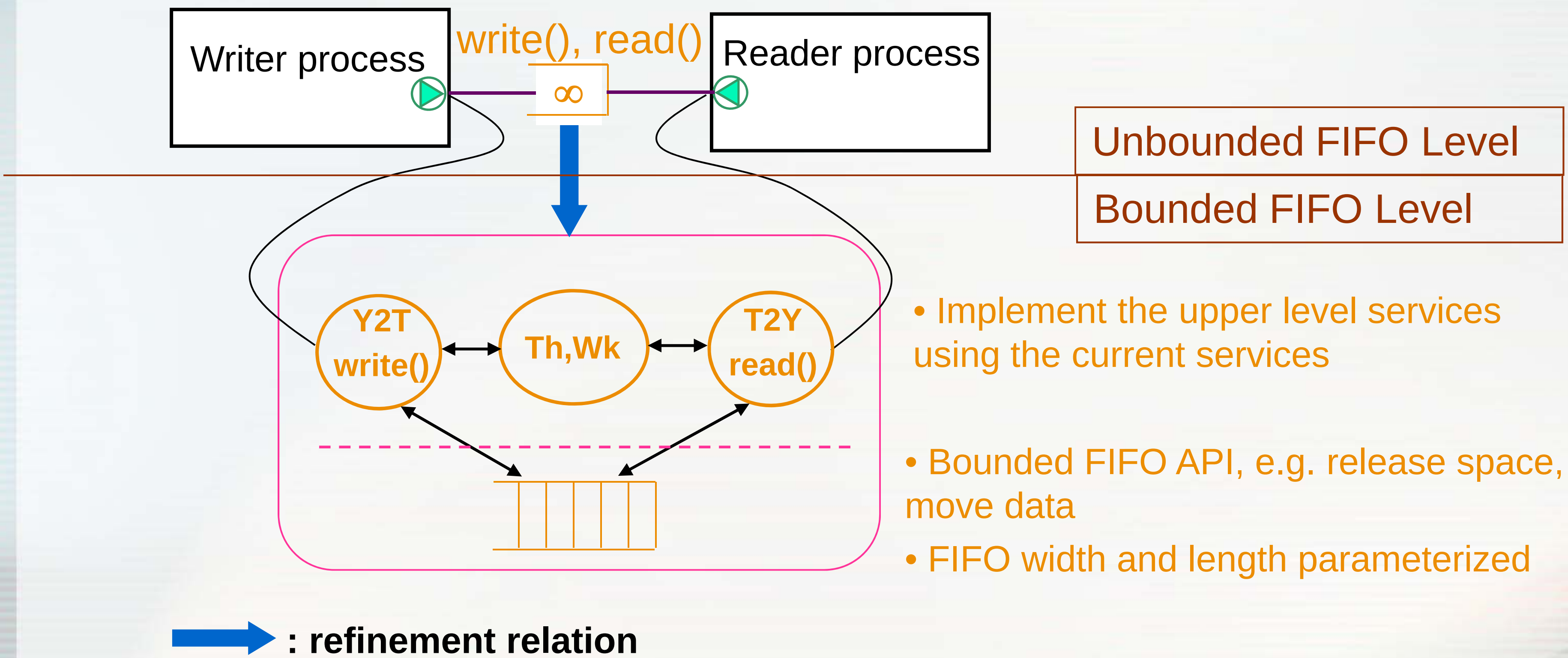
```
interface writer extends Port{
    update void write(int i);
    eval int space();
}
```

```
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(){ ... }
}
```

Thanks to
Doug Densmore

Leveraging the Abstract Semantics for Refinement Verification in Metropolis

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.
- Refinement relation is associated with properties to preserve through the refinement.

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:

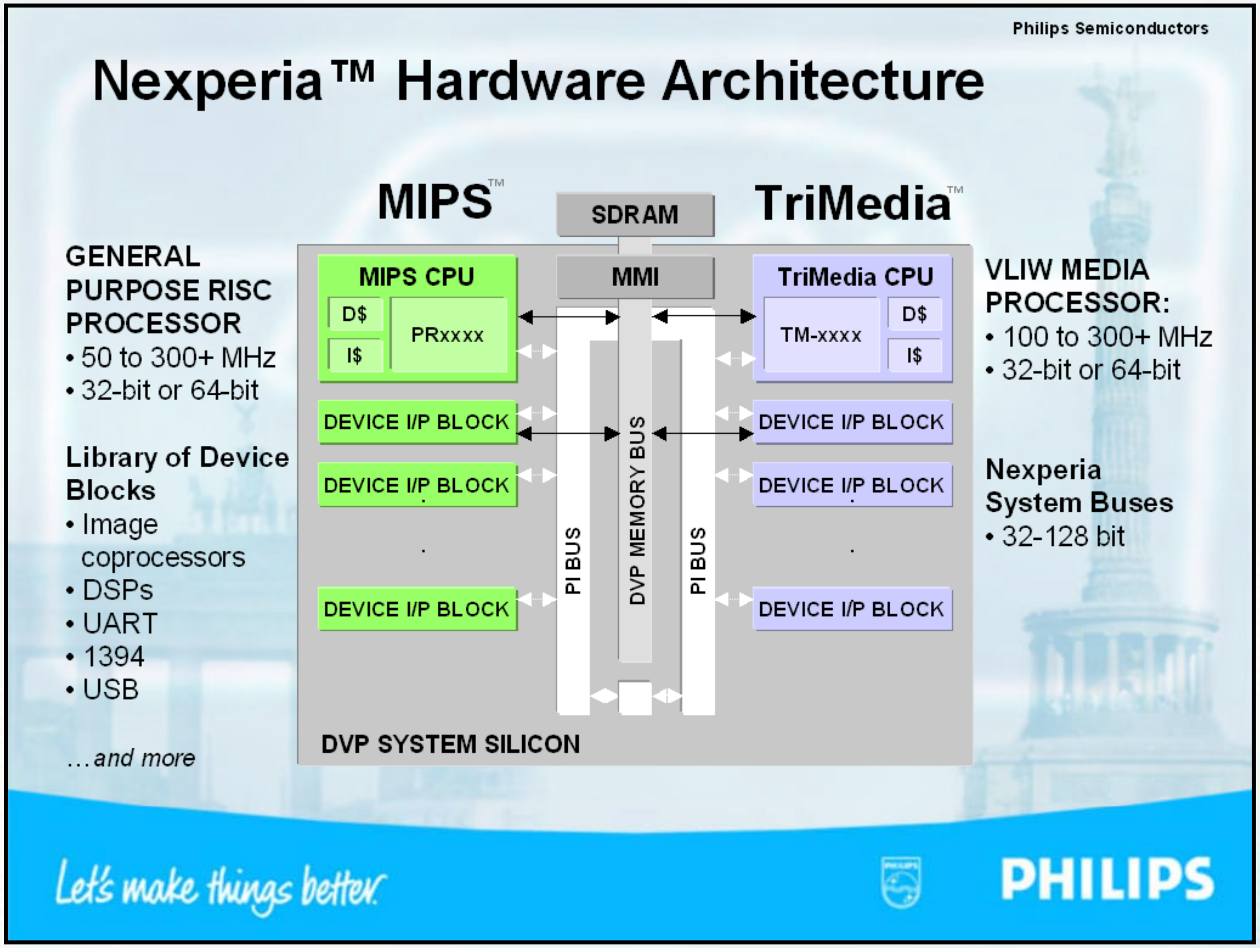
    concurrency, synchronization (awaits and constraints)

Metropolis Architecture Representation

Architecture components

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

```
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);
  }

  …
```

Meta-model: quantities

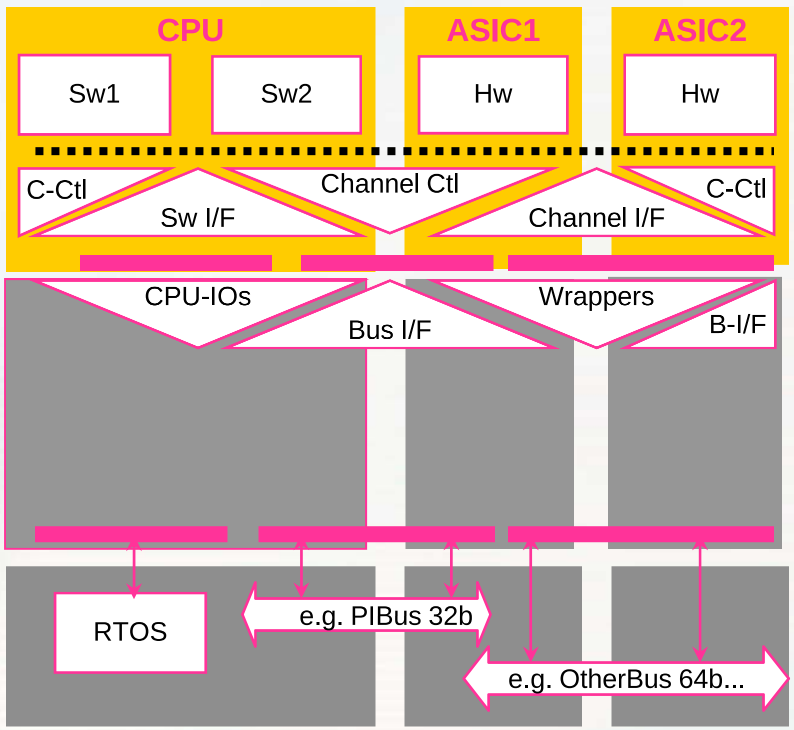
- The domain D of the quantity, e.g. *real* for the global time,
- The operations and relations on D, e.g. subtraction, <, =,
- The function from an event instance to an element of D,
- Axioms on the quantity, e.g.

  the global time is non-decreasing in a sequence of vectors of any feasible execution.

```
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)));
}}
```

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

Quantity resolution

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

1. **quantity requests**

   for each process *Pi*, for each event *e* that *Pi 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

Architecture Modeling Related Work

1. David C. Luckham and James Vera, *An Event-Based Architecture Definition Language* , IEEE Transactions on Software Engineering, Vol. 21, No 9, pg. 717-734, Sep. 1995.

2. Ingo Sander and Axel Jantsch, *System Modeling and Transformational Design Refinement in ForSyDe*, IEEE Transactions on CAD, Vol. 23, No 1, pg. 17-32, Jan. 2004.

3. Paul Lieverse, Pieter van der Wolf, Ed Deprettere, and Kees Vissers, *A Methodology for Architecture Exploration of Heterogeneous Signal Processing Systems*, IEEE Workshop in Signal Processing Systems, Taipei, Taiwan, 1999.

Return

	Metropolis	Rapide[1]	ForSyDe[2]	SPADE[3]
Mapping	x	x	x	x
Quantity Managers	x	No	No	No; collectors in bldg blocks
Event Based	x	x	x	No
Pure Architecture Model	x	x	No; Functional tied to Arch.	x

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)

Task After Mapping
Read (0x34, 8, 10, 4)

Programmable Arch. Modeling

- **Coordination Services**

PPC Sched

MicroBlaze Sched

PLB Sched

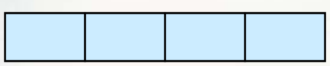
OPB Sched

BRAM Sched

General Sched

Request (event e)

-Adds event to pending queue of requested events

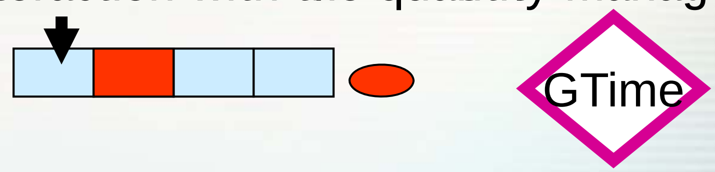


PostCond()

Resolve()

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

-Uses algorithm to select an event from the pending queue



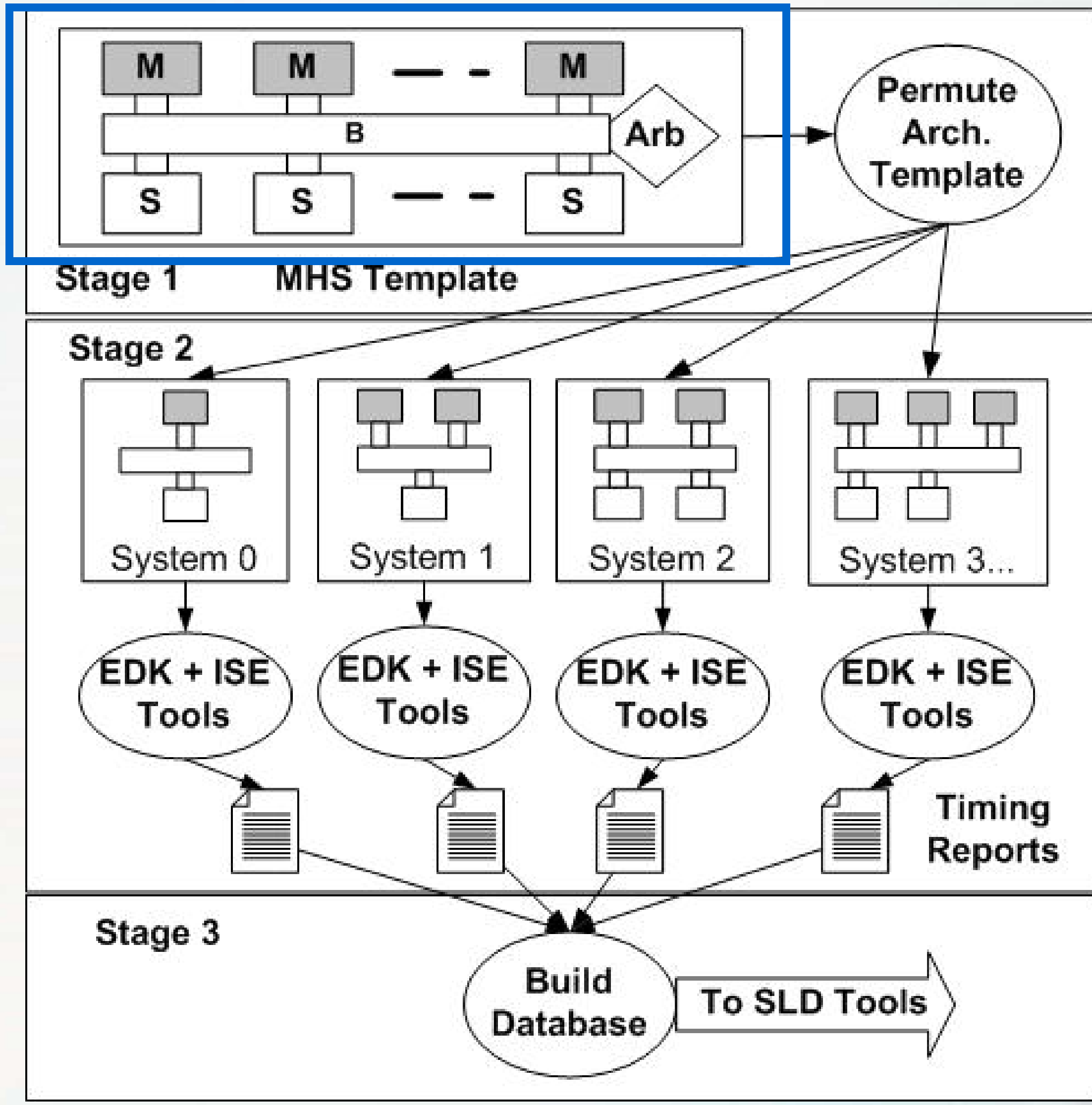
Prog. Platform Characterization

Need to tie the model to actual implementation data!

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.



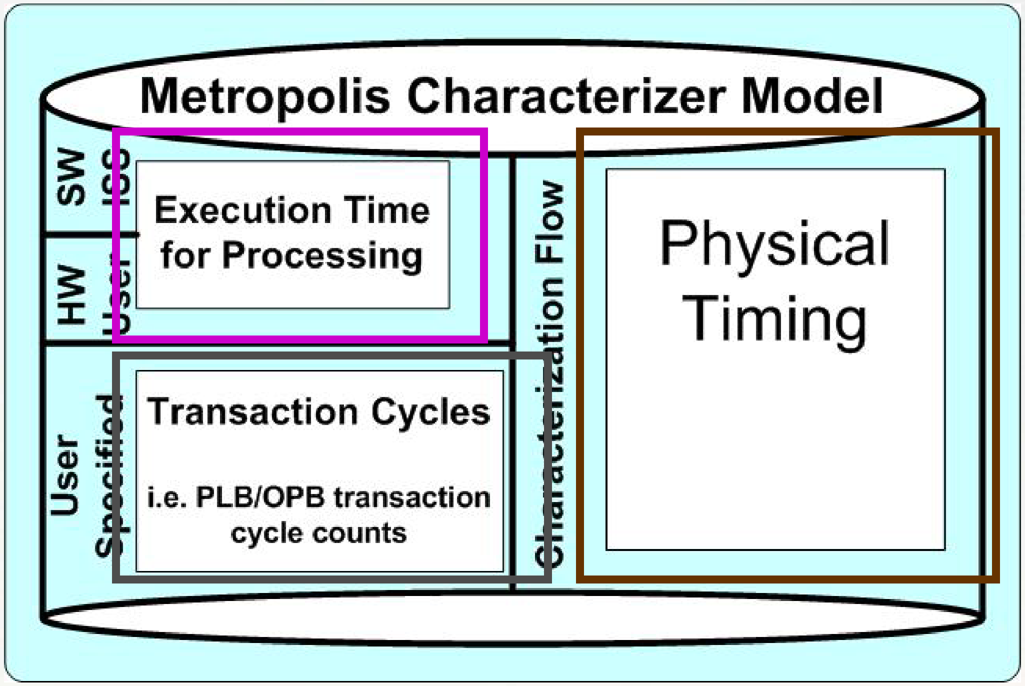
Prog. Platform Characterization

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.



From Char Flow Shown

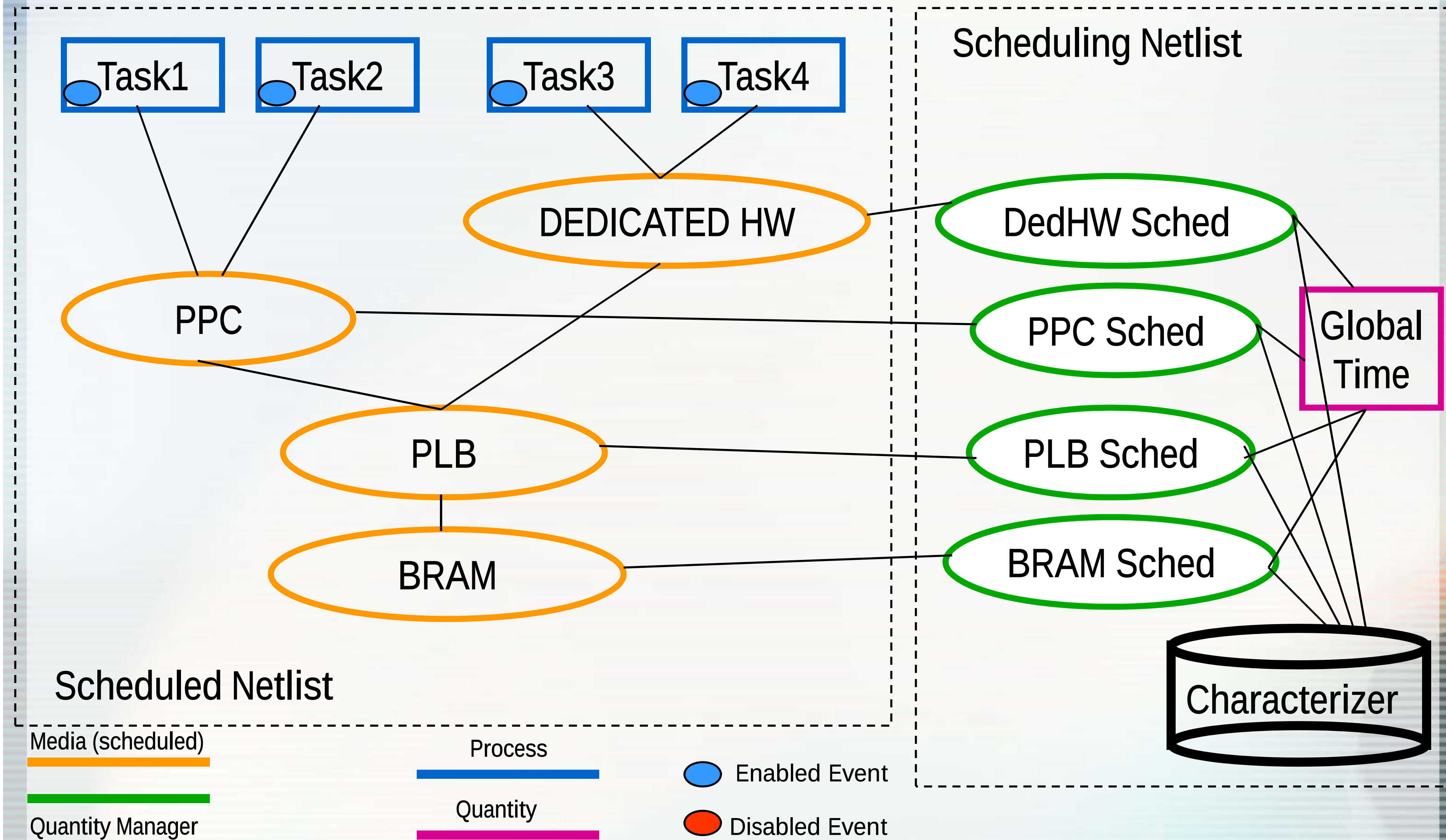
From Metro Model Design

From ISS for PPC

Work with Xilinx Research Labs

1. Douglas Densmore, Adam Donlin, A.Sangiovanni-Vincentelli, *FPGA Architecture Characterization in System Level Design*, Submitted to CODES 2005.
2. Adam Donlin and Douglas Densmore, *Method and Apparatus for Precharacterizing Systems for Use in System Level Design of Integrated Circuits*, Patent Pending.

Modeling & Char. Review



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.**

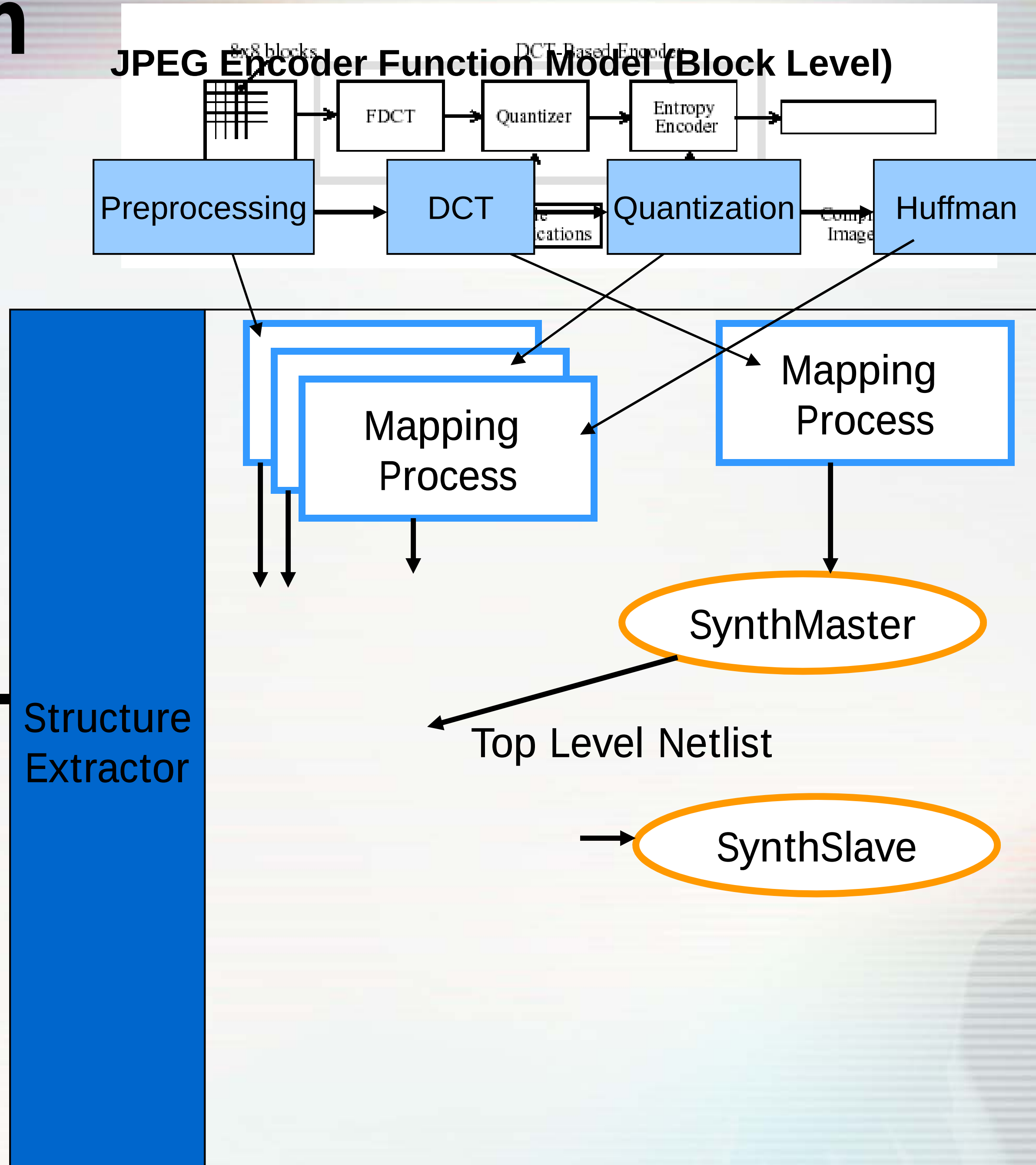
D. Garlan, *Style-Based Refinement for Software Architectures*, SIGSOFT 96, San Francisco, CA, pg. 72-75.

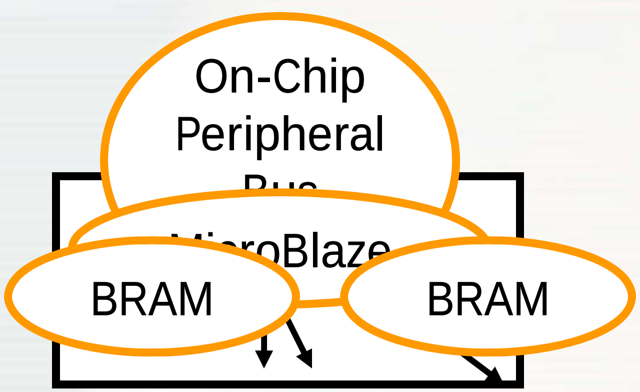
Refinement Technique	**Description**	**Metropolis**
Style/Pattern Based	Define template components. Prove they have a desired relationship once. Build arch. from them.	Potential; TTL YAPI
Event Based	Properties (behaviors) expressed as event lists. Explicitly look for this event patterns.	**Discussed**
Interface Based	Create structure capturing all behavior of a components interface. Compare two models.	**Discussed**

Example Design

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



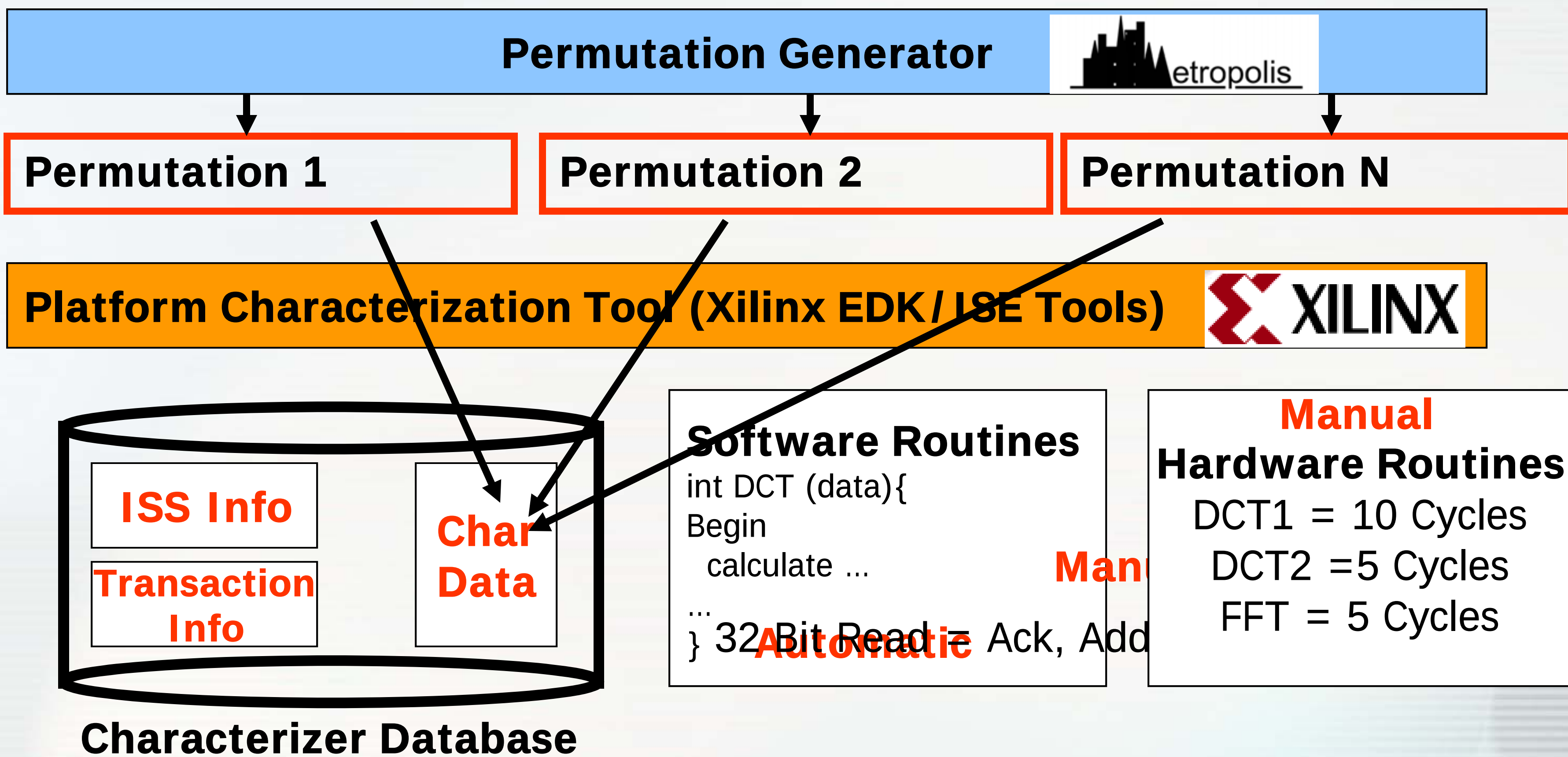


Example Design Cont.

Problem Statement
Approach
Contribution

File for Xilinx EDK Tool Flow

1. Feed the captured structural file to the permutation generator.
2. Feed the permutations to the Xilinx tools and extract the data.
3. Capture *transaction* info for software and hardware services.
4. Provide *transaction* info for communication services.



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