Other Tools and their relevance to Metropolis

Rong Chen
Trevor Meyerowitz
Roberto Passerone
Alberto Sangiovanni-Vincentelli

Outline

- Fresco
- Mescal
- Ptolemy II
- VCC
- SystemC
- UML
- Conclusions
The FRESCO Group

- Formal REaltime Software COMponents
- Methodology
  - Masaccio
    - Top level formal specification of continuous time and discrete time systems
  - Pure Giotto
    - Abstract time triggered programming
  - Annotated Giotto
    - Mapped and constrained Giotto
  - The E-Machine
    - A tool for supporting giotto

Masaccio

- Formal framework for embedded system description
  - Hardware and Software
  - All components are non-deadlocking
- Continuous and Discrete Time
- Hierarchical
  - Parallel composition
  - Serial composition
- Provable refinement
  - Supports Assume-Guarantee reasoning
**Pure Giotto**

- Time Triggered Programming (MOC)
- Abstract Description
- Instantaneous Communication
- Time-Deterministic Computation
- Different Components
  - Modes
  - Tasks
  - Ports
  - sensors, actuators, mode switch, private
  - Drivers – connections between ports

**Methodology**

- Masaccio
- Pure Giotto
- Annotated Giotto
- The E-Machine

---

**Refined Giotto & the E-machine**

- Refine from pure Giotto down to an implementation
- Generate code runnable on the specified platform.
- 3 phases of refinement
  - Giotto-P – Platform is specified. With hosts, nets, and worst case execution times.
  - Giotto-PM – Mapping is specified.
  - Giotto-PMS – Scheduling of the resources is added.
- The E-machine (the embedded machine)
  - Scheduling machine for control tasks
  - Based on timers/counters and a stack
  - Interface to an RTOS and Giotto

**Methodology**

- Masaccio
- Pure Giotto
- Annotated Giotto
- The E-Machine
Fresco Conclusions

◆ Pro’s
  ▶ Formalism (Provable performance), Hierarchy, Composition, and Refinement.
  ▶ Tools are mostly completed (e-machine still new)
  ▶ Addresses discrete + continuous time domains

◆ Con’s
  ▶ Restricted to developing time triggered control software
  ▶ Assumptions made may be too restrictive

◆ Relevance to Metropolis
  ▶ Formalism and Hierarchy are nice
  ▶ Great work on provable WCET

The Mescal Group

◆ Modern Embedded Systems: Compilers, Architectures, and Languages
◆ Focus on highly programmable heterogeneous multiprocessor embedded system platforms
  ▶ Currently driven by network processors
◆ Main views
  ▶ Application
  ▶ Architecture
  ▶ Programmer’s Model

The Methodology Picture*

*Portions taken from 12/00 Mescal GSRC presentation
**Mescal Architecture**

- Design exploration
- Single Processing Element
  - VLIW blocks
  - Formal model
  - Automatic generation of compiler and simulator
- Memory Architecture
  - Based on database access model
- Communication Architecture
  - OSI protocol stack
- Visualization and Simulation currently uses Ptolemy II
  - In the future use Liberty (Princeton)

**Mescal Application**

- Extract Maximal Parallelism
  - Process Level
  - Thread Level
  - Instruction Level
  - Bit Level
- Driven by Network Processor Applications
  - Super exponential growth
  - Many features being added
  - Application Profiling (currently MPLS)
    - Evaluate various processors and determine what arch. features are useful
**Mescal Programmer’s Model**

- **What the programmer sees**
  - Hide unnecessary details
  - Reveal useful details

- **Model of computation**
  - Not necessarily formal
  - Application Dependent
    - Currently working on the Click Network Router and its semantics
      - Push/Pull semantics
  - Much discussion ongoing

---

**Long Term Mescal Goals**

- **Flexible Architectural Environment**
  - Support Multiple Levels of Concurrency

- **Automatic generation of SW environment**
  - Simulators
  - Retargetable Compilers/Assemblers/Debuggers
  - Run time environment

- **Analysis Tools**
Mescal Conclusions

◆ Pro’s
  ▲ Automatic SW environment generation
  ▲ Architectural exploration techniques
  ▲ Seems great for programmers

◆ Con’s
  ▲ Still in early stages of development
  ▲ Takes a limited view of hardware

◆ Relevance to Metropolis
  ▲ Mostly Bottom-Up approach (vs. Top-Down-ishness of Metro)
  ▲ Software oriented approach
  ▲ Nice complement to Metro

Ptolemy II (UC Berkeley)

◆ Component-based simulator
  ▲ components interact with producer-consumer semantics

◆ Supports several models of computation

◆ Hierarchical Heterogenity
  ▲ each level of the hierarchy must use a single model
  ▲ heterogeneity only possible at the boundaries

◆ Methodology
  ▲ choose a model for each object
  ▲ mix models hierarchically
  ▲ create reusable models (polymorphism)
**Hierarchical Heterogeneity**

**Ptolemy Meta-Model**

- Abstract syntax based on a network of components
  - Actors are connected through ports and relations
- Actors expose an execution API:
  - preinitialize();
  - initialize();
  - prefire();
  - fire();
  - postfire();
  - wrapup();
  - terminate();
- Composite actors characterized by a director
**Ptolemy execution model**

- The *director* defines the execution semantics of a composite actor (one level of hierarchy)
- A lower level director is treated as an actor with the additional transfer of data to/from the higher level
- An execution is a sequence of iterations preceded by initialization followed by wrapup
- During each iteration the director calls the fire methods of the controlled entities
  - (prefire(), fire(), postfire())*

**Ptolemy data transfer**

- Ports expose a data communication API
  - put();
  - get();
  - hasRoom();
  - hasToken();
- A *receiver* in each port defines the semantics of the communication
  - by sequencing the calls to the communication API
  - by suspending the execution when appropriate
- A director and a receiver define the model of computation
Ptolemy II Conclusions

◆ Pro’s
  ▲ Robust software infrastructure
  ▲ Heterogeneous systems
  ▲ Flexible meta-models supports several domains (including fixed-point semantics)
  ▲ Separation of communication protocols from the actor description

◆ Con’s
  ▲ Limited to hierarchical model composition
  ▲ Domains not completely formally defined
  ▲ Lack of architectural exploration, refinement, implementation

◆ Relevance to Metropolis
  ▲ Environment for experimentation on heterogeneous design
  ▲ Possible underlying simulation engine

VCC (Cadence)

◆ Virtual component integration
◆ Architecture exploration
◆ Separation of functionality and architecture
◆ Functional simulation
◆ Performance simulation through mapping
◆ Underlying discrete event model of computation
VCC functionality

◆ Blackbox
  ▲ simulation model

◆ Whitebox
  ▲ performance estimation
  ▲ simulation model

◆ Clearbox
  ▲ synthesis
  ▲ performance estimation
  ▲ simulation model

VCC architecture

◆ Predefined architecture blocks for computation and communication
  ▲ ASIC
  ▲ CPU
  ▲ OS
  ▲ data bus, interrupt bus

◆ Predefined communication patterns
  ▲ interrupt, polling, shared memory, etc.

◆ Flexibility using Architectural Services
  ▲ provide performance simulation model
  ▲ ability to model at several levels of details
  ▲ dynamic binding for mapping independence
Performance modeling

- **Delay scripts**
  - Relative timing of input to output, including some data dependency
- **Annotated blackbox/whitebox**
  - Flexible in-line annotation for higher accuracy
- **Software estimation**
  - High-level processor models for fast performance estimation
- **Specialized architectural services**
  - Flexible architectural modeling tool for arbitrary trade-off of speed vs. accuracy
- **Not just timing models!**

VCC Links to Implementation

- **Customizable exporters**
- **Automatic synthesis of communication infrastructure**
  - register mapping
  - memory allocation
  - interrupt service routine
  - operating system calls
VCC conclusions

◆ Pro’s
  ▲ Architectural exploration with performance estimation
  ▲ Fast simulation
  ▲ Software and communication synthesis

◆ Con’s
  ▲ Single model of computation
  ▲ No hardware synthesis

◆ Relevance to Metropolis
  ▲ Separation of functionality and architecture
  ▲ Advanced architectural modeling capabilities
  ▲ Communication patterns
  ▲ Support for Operating System modeling

SystemC 1.0

◆ Extension of C++ with modules, ports and signals
◆ HW-like data types, including fixed-point types
◆ Model
  ▲ discrete event with delta cycles
  ▲ processes react to changes in signals (sensitivity list)
◆ Abstraction close to that of RTL hardware description languages
SystemC 2.0

- An attempt to raise the level of abstraction to the system level
- Introduces new concepts
  - channels: container for communication and synchronization
  - interfaces: set of access methods for a channel
  - events: low-level synchronization primitives
- Can model HW signals, queues, semaphores, memories, busses, etc.

SystemC 2.0 model

- 64-bit unsigned integer absolute global time
- Supports static sensitivity list as in 1.0
- Supports event based synchronization
  - wait( event || time stamp );
  - event.notify( <time> );
**SystemC 2.0 execution model**

- Execution phases:
  - initialization, ( ( evaluate*, <update> )*), advance *)
- Based on co-routines and independent stacks for each process
- Order of execution of processes in the evaluate phase is unspecified
- Processes can only be preempted when they call wait( )
  - Atomicity of execution is guaranteed between wait( )'s
- To model preemption, must include effects of delays in the architecture using wait( )

**SystemC 2.0 comm. model**

- Interfaces define a set of access methods
- Channels implement one or more interfaces
  - primitive: have no structure and do not access other channels
  - hierarchical: are modules that can access other modules and channels
- The interface method is executed by the channel in the context of the caller
- Use the “request-update” scheme for complex synchronizations
- Communication refinement
  - Adapter insertion: convert the interface from abstract and refined
  - Interface refinement: merge adapters into processes (aka “protocol in-lining”)
SystemC 2.0 Conclusions

◆ Pro’s
  ▲ Industry standard (potential for wide adoption and tools)
  ▲ Supports communication refinement

◆ Con’s
  ▲ No architectural modeling
  ▲ No software modeling

◆ Relevance to Metropolis
  ▲ Target simulation environment
  ▲ Similar concepts of separation between processes and communication media

---

UML

◆ A standard adopted by OMG for SW specification
◆ A language designed to model, NOT to implement
  ▲ good for system specification
  ▲ limited in defining contents of actions
  ▲ insufficient to support testing, simulation, code generation

◆ Diagrams
  ▲ Use case diagrams (graphical/textual)
    ▼ represent functional requirements
  ▲ Class diagrams
    ▼ show the static structure of the system
  ▲ State diagrams
    ▼ show how objects transit from one state to another
  ▲ Other diagrams
    ▼ activity diagrams, physical diagrams, interaction diagrams, package diagrams ...
**UML**

- **Examples**
  - Use Case Diagram
    - ![Use Case Diagram](image)
  - Class Diagram
    - ![Class Diagram](image)

- **Pro’s**
  - Clearly specified syntax
  - Expressed in multiple visual/textual diagrams
  - Extensible mechanism

- **Con’s**
  - Vaguely defined semantics
  - Translation from syntax to semantics not precisely defined

- **Relevance to Metropolis**
  - Can be applied to a wide range of applications
  - Can serve as a very good front end description of Metro
    - Well defined syntax
    - Can describe syntax for any MOC
    - UML -> Metro IP needs to be done
Summary

- FRESCO
  ▲ Formal design with an emphasis on time-triggered software
- Mescal
  ▲ Focus on software for highly programmable embedded platforms
- Ptolemy II
  ▲ Modeling of heterogeneous systems
- VCC
  ▲ Architecture exploration, performance estimation
- SystemC
  ▲ Separation of communication and computation
- UML
  ▲ Specification and documentation of system structures

Questions Raised

- How can we get these different tools to work together?
- Should we get them to work together?
- What are the strengths and weaknesses of these tools? Of Metropolis?
- Have we missed anything?
  ▲ Tools?
  ▲ Observations?