Transaction-Level Modeling and Electronic System-Level Languages

Steven P. Smith

SoC Design

EE382V
Fall 2009

Overview

- Motivation: Why have ESL languages?
- Transaction-Level Modeling
- Levels of abstraction in modeling
- Basic requirements of ESL languages
- ESL languages and environments: trade-offs
- An overview of a sampling of ESL languages
- What's missing from current ESL languages?
- Conclusions

Motivation

- Why use transaction-level modeling and ESL languages?
  - Manage growing system complexity
  - Move to higher levels of abstraction
  - Enable HW/SW co-design
  - Speed-up simulation
  - Support system-level design and verification

Increase designer productivity

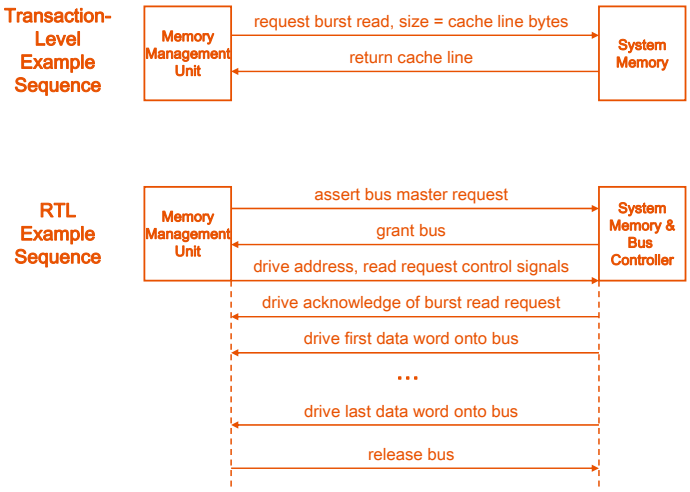
Reduce development costs and risk

Accelerate time-to-market & time-to-money

Transaction-Level Modeling

- Communication among modules occurs at the functional level.
  - Each transaction is a coherent unit of interaction
  - Data structures and object references are passed instead of bit vectors
- Goals of TLM
  - Higher level of abstraction
  - More comprehensible high-level system models
  - Greater simulation speeds
- Advantages of TLM
  - Natural way to think about high-level communications
  - Object Independence
  - Abstraction Independence

Transaction-Level Modeling



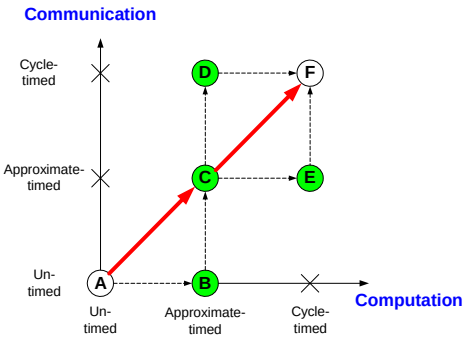
Elements of Transaction-Level Modeling

- Transaction-Level Modeling =
  < {objects}, {compositions} >
- Object = {computation object} | {communications object}
- Composition
  - Computation objects send and receive abstract data via communications objects.
- Advantages of TLM
  - Object Independence
  - Abstraction Independence

* Definition from Gajski and Cai, UC Irvine

Levels of Abstraction

- Consider models as a function of their time-granularity



A. **Specification Model**
   "'Untimed' Functional Models"

B. **Component-Assembly Model**
   "Architecture Model"
   "'Timed' Functional Model"

C. **Bus-Arbitration Model**
   "Transaction Model"

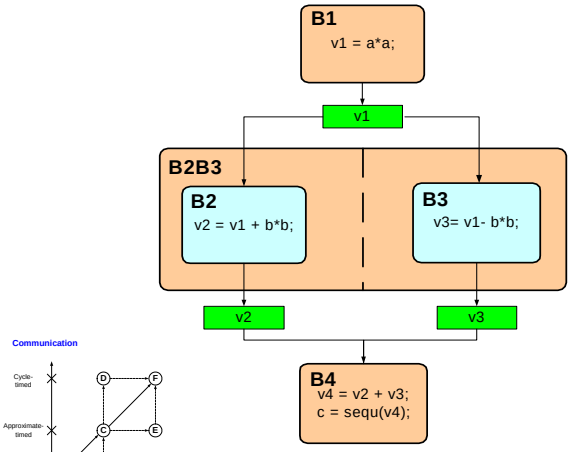
D. **Bus-Functional Model**
   "Communication Model"
   "Behavior-Level Model"

E. **Cycle-Accurate Computation Model**

F. **Implementation Model**
   "Register-Transfer Level (RTL) Model"

* Figure and taxonomy by Gajski and Cai, UC Irvine

Specification Model



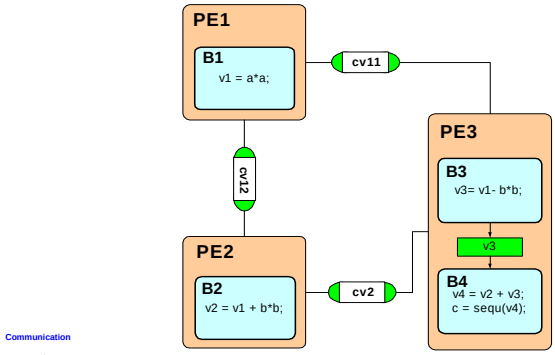
Objects:
- Computation:
  - Behaviors
- Communication:
  - Variables

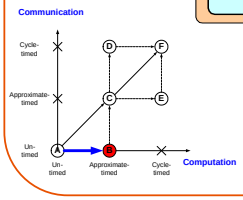
Composition:
- Hierarchy
- Execution Order
  - Sequential
  - Parallel
  - Pipelined
  - States

Synchronization:
- Notify/Wait

* Figure and taxonomy by Gajski and Cai, UC Irvine

Component-Assembly Model





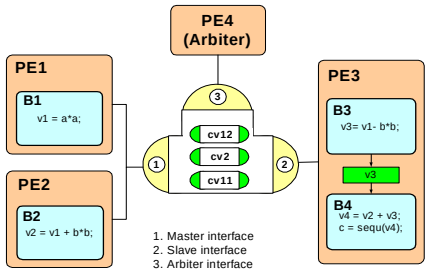
Objects:
- Computation:
  - Processors
  - Memories
  - IP
- Communications:
  - Variable Channels

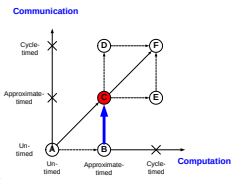
Composition:
- Hierarchy
- Execution Order
  - Sequential
  - Parallel
  - Pipelined
  - States

Synchronization:
- Notify/Wait

* Figure and taxonomy by Gajski and Cai, UC Irvine

Bus-Arbitration Model





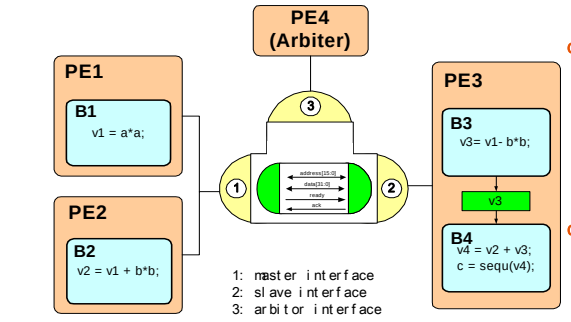
Objects:
- Computation:
  - Processors
  - Memories
  - IP, arbiters
- Communications:
  - Abstract Bus Channels

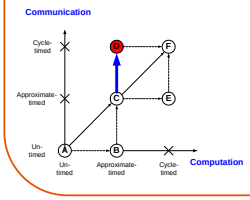
Composition:
- Hierarchy
- Execution Order
  - Sequential
  - Parallel
  - Pipelined
  - States

Synchronization:
- Notify/Wait

* Figure and taxonomy by Gajski and Cai, UC Irvine

Bus-Functional Model





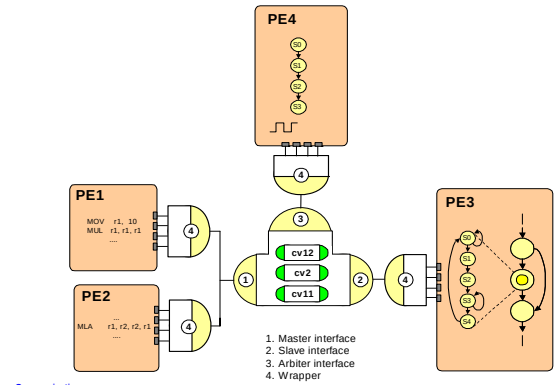
Objects:
- Computation:
  - Processors
  - Memories
  - IP, arbiters
- Communications:
  - Protocol Bus Channels

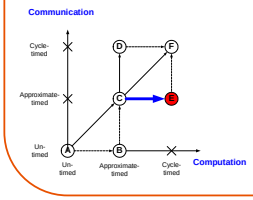
Composition:
- Hierarchy
- Execution Order
  - Sequential
  - Parallel
  - Pipelined
  - States

Synchronization:
- Notify/Wait

* Figure and taxonomy by Gajski and Cai, UC Irvine

Cycle-Accurate Computation Model





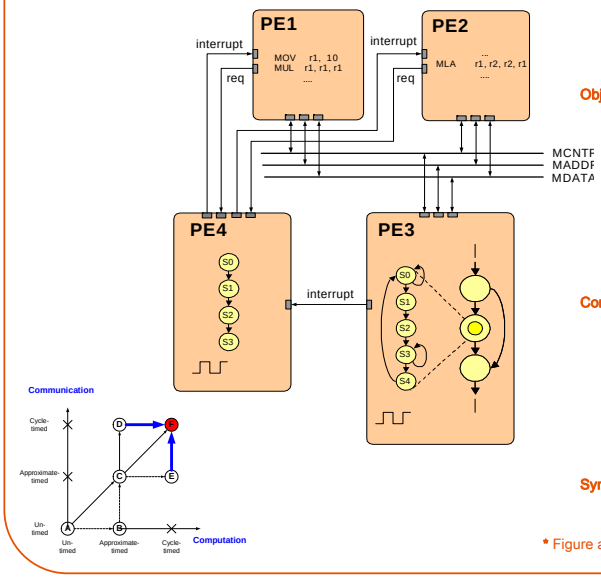
Objects:
- Computation:
  - Processors
  - Memories
  - IP, arbiters
  - Wrappers
- Communications:
  - Abstract Bus Channels

Composition:
- Hierarchy
- Execution Order
  - Sequential
  - Parallel
  - Pipelined
  - States

Synchronization:
- Notify/Wait

* Figure and taxonomy by Gajski and Cai, UC Irvine

Implementation Model



Objects:
- Computation:
  - Processors
  - Memories
  - IP, arbiters
  - Wrappers
- Communications:
  - Buses/Wires

Composition:
- Hierarchy
- Execution Order
  - Sequential
  - Parallel
  - Pipelined
  - States

Synchronization:
- Notify/Wait

* Figure and taxonomy by Gajski and Cai, UC Irvine

Characteristics of the Different Models

Models	Communication time	Computation time	Communication scheme	PE interface
Specification model	no	no	variable	(no PE)
Component-assembly model	no	approximate	variable channel	abstract
Bus-arbitration model	approximate	approximate	abstract bus channel	abstract
Bus-functional model	time/cycle accurate	approximate	protocol bus channel	abstract
Cycle-accurate computation model	approximate	cycle-accurate	abstract bus channel	pin-accurate
Implementation model	cycle-accurate	cycle-accurate	bus (wire)	pin-accurate

* Figure and taxonomy by Gajski and Cai, UC Irvine

Transaction-Level Formalisms

- Rigorous definition of elements and operators in a transaction-level model
- Precision in modelling aids comprehension of designs
  - But only if the notation is easily understood by designers
- Key goal is to enable synthesis from ESL level
  - There is a fundamental tension between representations that are easily understood by designers and those that are easily "understood" by tools.
  - More work in early stages of design

* From Gajski and Cai, UC Irvine

Model Algebra

- Algebra = *< {objects}, {operations} >* [ex: a * (b + c)]
- Model = *< {objects}, {compositions} >*
- Transformation t(model) is a change in objects or compositions.
- Refinement of a model is an ordered set of transformations, < tm, … , t2, t1 >, such that:

  model B = tm(… (t2(t1(model A))) …)

- Model algebra = *< {models}, {refinements} >*
- Methodology is a sequence of models and corresponding refinements

* From Gajski and Cai, UC Irvine

Model Definition

- **Model** = < {objects}, {composition rules} >
- **Objects**
  - **Behaviors** (representing tasks | computation | functions)
  - **Channels** (representing communication between behaviors)
- **Composition rules**
  - Sequential, parallel, pipelined, FSM
  - Behavior composition creates hierarchy.
  - Behavior composition creates execution order.
  - Rules define the relationships between behaviors in the context of the formalism.
- Relationships between behaviors and channels
  - Data transfer in channels
  - Interface between behaviors and channels

* From Gajski and Cai, UC Irvine

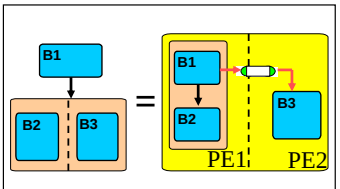
Model Transformations (Rearrange and Replace)

- Rearrange object composition
  - Distribute computation over components.
- Replace objects
  - Import library components
  - Develop more detailed behaviors
- Add or remove synchronization
  - Parallel -> sequential
  - Sequential -> parallel
- Decompose abstract data structures
  - Map data transactions to a specific bus structure
- …

a*(b+c) = a*b + a*c

Distributivity of multiplication over addition

analogous to……



Distribution of behaviors (tasks) over components

* From Gajski and Cai, UC Irvine

Model Refinement

- Definition
  - A refinement of a model is an ordered set of transformations, < tm, … , t2, t1 >, such that:

    model B = tm(… (t2(t1(model A))) …)
- Derives a more detailed model from one more abstract
  - Specific sequence of steps for each model refinement
  - Not all sequences are relevant
- Equivalence verification
  - Each transformation maintains functional equivalence
  - The refinement is thus "correct by construction."
  - Not always (typically?) possible
- Refinement-based system-level methodology
  - Methodology is a sequence of models and refinements

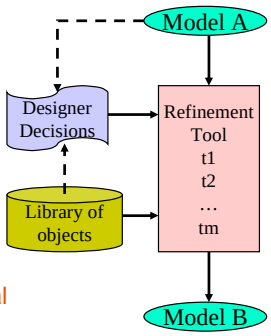
* From Gajski and Cai, UC Irvine

Verification by Equivalent Transformations

Transformations can be made to preserve equivalence

- Same partial order of tasks
- Same inputs and outputs for each task (unknown value handling aside)
- Same partial order of data transactions
- Same (or covered) functionality in the replacements

- Refined models "equivalent" to the input model
  - Still need to verify first model using traditional (i.e., simulation) techniques
  - Still need to verify equivalence of replacements
  - In practice, this is not always possible.



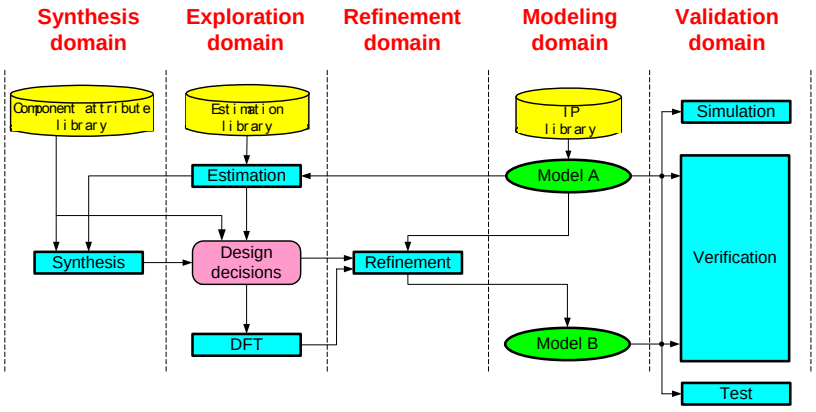
* From Gajski and Cai, UC Irvine

Synthesis

- Set of models
- Set of design tasks
  - Profile
  - Design-space exploration
  - Select components / connections
  - Map behaviors / channels
  - Schedule behaviors/channels
  - ...
- Each design decision results in a model transformation.
- Detailing is a sequence of design decisions.
- Refinement is a sequence of transformations
- Synthesis is detailing and refinement.
- The challenge, of course, is to define the "right" sequence of design decisions and transformations.

* From Gajski and Cai, UC Irvine

Design Domains



* From Gajski and Cai, UC Irvine

Transaction-Level Modeling Conclusions

- In TLM, computation and communication objects are connected through abstract data types.
- TLM enables modeling each component independently at differing levels of abstraction.
- A major challenge is to define, obtain, or develop the necessary and sufficient set of models for the design flow.
- Another major challenge is to define the model algebra and its corresponding methodology to make the design flow as efficient as possible (e.g., synthesis).
- In practice, assembling the system model is no small feat either, especially when models come from different sources (e.g., third-party IP, embedded processor vendor, etc.).
- **The potential payoff is truly enormous.**

* From Gajski and Cai, UC Irvine

Basic Requirements of ESL Languages

- Support for Transaction-Level Modeling
  - Objects can be modeled independently.
  - Objects can be modeled at different levels of abstraction.
- Object Independence
  - Black-box objects
  - Third-party objects (IP)
- Abstraction Independence
  - Assists in verification of the sequence of refinements
  - Flexibility in development methodologies.
- Support all models of computation
- Enable high-speed simulation

ESL Language and Environment Design Trade-Offs

- Object-oriented?
  - A natural way to think of system behavior
  - Easy to build component and data abstractions
- General-purpose language extensions?
  - Easier to support third-party tool, test-bench and model interfaces, although doing so may require significant expertise and effort
  - Generally more open and flexible
- Precise representation of software modules?

More ESL Language and Environment Design Trade-Offs

- "Platform-based" environment?
  - System-level model "stitching" may be greatly simplified through the use of a single model library…
  - …until that library doesn't have what you need, and you are forced to import or develop models or tools.
- Well defined third-party tool and model interfaces?
  - Resorting to "pure" C or C++ features is often an unsatisfying and complex recourse when problems are encountered.
    - System model assembly quickly becomes an extremely challenging task.
- Black-box models often embody their own simulation semantics
  - May require a "simulator of simulators."

ESL Languages: SpecC

- Extension of ANSI-C
  - Every C program is a SpecC progam
  - SpecC type extensions for HW (minimal by design):
    - Boolean
    - Bit vectors
    - Events
  - Basic structure consists of behaviors, channels, interfaces, variables, and ports
  - Focus on automated transformations and synthesis
    - Arguably somewhat “hardware-centric”
  - Not widely adopted by industry or EDA community

ESL Languages: System Verilog

- Standards-based successor to Superlog, a language combining Verilog and C previously developed by Co-Design Automation (now part of Synopsys)
  - Extends Verilog 2001 (IEEE-1364-2001) with complete interface to C
  - Verilog inside “comfort zone” of today’s hardware designers (where SystemC clearly is not)
  - Bluespec has released an ESL Synthesis tool based on “Bluespec System Verilog.”
    - Higher than RTL
    - But still obviously (and intentionally) close to the hardware *structure* and not purely its behavior

ESL Languages: SystemC

- Class library extension to C++
- Recently extended to support verification-specific constructs
- C++ can be intimidating to HW designers trained in Verilog or VHDL
- Software developers find it easier to integrate their programs and tools than with other ESL languages.
- Open standard effort through the Open SystemC Initiative (OSCI)
- Synthesis tools emerging in the marketplace

SystemC Advantages

- SystemC is well-matched to the development of application-specific SoC's that start from a working base of application software.
  - Media processors typify this class of SoC.
  - Develop from the application code down to the hardware.
    - Comparatively simple (depending on code structure) to partition and map software modules to hardware elements during design-space exploration
    - Verification at each step of the refinement process uses the original (typically regression) test-bench.

AADL: Architecture Analysis and Design Language

- Adopted as standard by SAE
  - Originally developed specifically for mission-critical avionics
  - Part of RTCA* DO-254 and DO-178B standards for mission-critical hardware and software, respectively
- Supports rigorous definition of both software and hardware models and their interfaces
  - Enables automated generation of software builds
  - Notation limited to module interfaces
- Distinguished from hardware-centric ESLs
  - Software modules not merely an afterthought

* Radio Technical Commission for Aeronautics

Today's ESL Languages: What's Missing? (*A Few Brief Editorial Comments*)

- In practice, an electronic systems-level design effort encompasses, minimally:
  - Hardware elements, including general-purpose processors, other third-party IP, custom processors, hardware accelerators, memories, analog interfaces, etc.
  - Software elements, including microcode, hardware abstraction layer (HAL) interface code, operating systems (typically an RTOS), application code, etc.
  - Hardware test benches and related tools, scripts, etc.
  - Software test benches and related tools, scripts, etc.

Today's ESL Languages: What's Missing?

- Elements of practical ESL design efforts, continued:
  - Debugging tools for HW and SW
  - Compilers, assemblers, linkers, etc.
  - Sensors of various types, and models for them
- Current ESL languages tend to give short shrift to everything but the hardware elements.
  - Third-party hardware IP issues are often overlooked as well
  - "Growing up the abstraction ladder from RTL"
- Total development effort and cost for software often substantially exceeds that required for hardware.

Today's ESL Languages: What's Missing?

- In effect, current ESL language development has been driven simply by the laudable but narrow goal of improving the productivity of hardware designers.
  - The inescapable conflict between Moore's Law and Brook's Law (*The Mythical Man-Month*)
  - Improved hardware design productivity is an important goal, to be sure, but…
  - … targeting a reduction in the overall system development cost, time, risk, etc., is ultimately the only meaningful goal.
    - At the end of the day, SoC's are still, unavoidably, a business venture, and success depends upon all elements of the development process (among a great many factors).

Today's ESL Languages: What's Missing?

- In practice, constructing and maintaining system models can take many months of effort.
  - The presence of heterogeneous multiprocessor SoC's, often with their own software development tools and debuggers, further exacerbates the problem.
    - Coordinating the execution of all the tools and models is non-trivial, to put it mildly.
    - For example, how do you get two different debuggers to cooperate during multiprocessor debugging?
  - Third-party IP models may encapsulate their own simulation semantics.
    - Thereby requiring a simulator to coordinate the simulators…
    - Merging cycle-based models with event-driven, etc.

Conclusions

- Transaction-Level Modeling is key to exploiting ESL languages and design methodologies.
- Electronic System-Level languages enable the use of higher levels of abstraction in hardware modeling.
  - Improved hardware design productivity
  - HW/SW co-design
  - Transformation and refinement of models through synthesis is emerging.
- Developing operational ESL models of systems remains a very challenging task.
  - We're now only looking at the tip of the iceberg.
- ESL design methodologies must address the entire design flow, not just the hardware.