Today we find that embedded systems touch almost every aspect of our daily lives. Such applications can comprise thousands of lines of code, large heterogeneous collections of microprocessors, VLSI components, and array logics. Such collections may include significant legacy components. Can’t afford to redesign each new system from ground up. Mixture of different forms of control. Event driven. Data flow constructs. Mixture of different technologies. Components or IP from a variety of different sources. May not have access to internals. May be distributed. Around an office or around the world.

Systems on a chip are becoming increasingly more common - SOC. Moving towards networks on a chip - NOC.

Our designs are often subject to contradictory constraints. On cost. Size, Power, Speed. Safety, Reliability. At the same time. Designs becoming increasingly sophisticated and powerful.

The ubiquitous microprocessor. Coupled with our advances in design and development of large scale integrated circuits. ASICs and programmable logics. Supported by our progress and developments in the software fields. Giving us opportunities to explore designs that push the edges our current science.

Today the successful deployment of embedded applications requires new approaches and new tools. That can deal with these complexities.
Yet the typical developer
Continues to use methodologies and principles
That can be quickly overwhelmed by
Complexities and demands of modern systems
The problem is growing worse

The modern creative design and development process
Begins with
Abstracted notion of the system to be built
Then moves through an iterative series of transformations to the final product
Computer based tools and methods are essential to that process

**Hardware – Software Co-Design**
Traditional design approach dictates that we
- Design the hardware components
- Design the software components
- Bring the two together
  Often the software components
    Significantly lag the hardware components
    Are forced to fit the hardware
Error in hardware design
  May be discovered late in design process
  Today mask costs are in the millions of dollars
  Cost of error can be significant
    Time and dollars
- Spend time testing and debugging
  Later in development
    Where cost of change is higher

Co-design
Combined and ‘simultaneous’ design of
Hardware and software components
With objective of meeting system-level requirements
  Through trade-offs between these components
Key points
Specify and (iteratively) design and develop both aspects of system concurrently
Interactive design of hardware and software
  To meet required performance and functional objectives

Goals
As with any design or design process
- Increase productivity – reduce design cycle time
- Improve product quality
For tomorrow’s designs
  • Boundary goal of one pass silicon

Process Overview
  Hardware / software co-design
    Suggests tools and approaches / methodologies to system design that integrates
    Design and development of both hardware and software components

Co-design focuses on major areas of design process
  • Ensuring a sound hardware and software specification and input to the process
  • Formulating the architecture for the system to be designed
  • Partitioning the hardware and software
  • Iterative approach to the design of hardware and software
    Includes iterative process or
    Synthesis, simulation, and verification

Co-design process comprises number of sub processes
  Included among these are
  • System specification
    Develop and validate the specification for both sets of components
    This is an essential component
  • Modeling - Architecture
    Develop system, hardware and software models
    Take into consideration both aspects
    Work together to
    Model complete system
    Validate the models
  • Co-design
    Partitioning the system into hardware and software components
    Refine interprocess communication
    Synthesize hardware – software interfaces
    Co-synthesis
    Synthesize both components from higher level models
    Software synthesis
      Target specific tasks to hardware components
    Hardware synthesis
      Decompose computation steps into clock cycles
      Bring two pieces together
    Co-simulation
  • Co-verification
    Verify that models meet specification
  • Repeat the process

Such processes require
  Computer support
  Computer based tools
Advantages

Co-design permits both hardware and software
To influence the design during early stages
There is a continual verification of the design
Throughout the development cycle
Models and interoperability of hardware and software design tools
Throughout process
 Particularly during architecture definition
Permits
 Greater exploration of design trade-offs
 More interesting architectures
Reuse is integral part of process
 Component’s behaviours
 Previously verified and co-verified as part of existing design
Enable one to reduce integration and test times
Quicker to market
Approach the boundary of one pass silicon

History

Before going too deeply into the codesign process
Let’s begin our study with a look at
Where we came from

Independent of which design methodology we examine
We have what we call a product life cycle
Life cycle comprises necessary steps or processes
 Necessary to execute a design using the methodology

We’ll begin with these

Life Cycle Models

The life cycle of the development of a system
 Is purely descriptive representation
 Breaks development process into series of interrelated activities

Over the years
 Number of models have been suggested for
 Implementing development process
At the end of the day
 All have same basic goal
 Have similar phases
 Better described as needs
 Specification or definition
 The design
 The implementation
 Manufacture
 Application to intended problem
Several of historically more common
Waterfall
Contractual
V cycle
Spiral
Rapid Prototype

**Waterfall**

Waterfall model represents cycle
Series of steps
Appearing much like waterfalls
Sequentially
One below the next
Steps given as
✓ Specification
✓ Preliminary Design
  Design review
✓ Detailed Design
  Design review
✓ Implementation
  Review

Each step
Linked to next
Chained manner
Has verification phase
Tends to say
Complete each phase
Go on to next
Recognition of problems
Tends to be delayed until later states of development
Limited in sense
Does not consider iterative
nature of real design

**V Cycle**

Similar to the waterfall model
Except it emphasizes the importance of
Addressing testing activities up front
Instead of later in the life cycle
For each stage
Associates each
development activity
With a test or validation at the same level

Each test phase is considered in its matching development phase
Requirements ↔ System/Functional Testing
High-Level Design ↔ Integration Testing
Detailed Design ↔ Unit Testing

Each development activity
Builds a more detailed model of the system
Than the one before it

Each validation
Tests a higher abstraction than its predecessor

The specification and design procedure
Top down model
Implementation and test
Utilize bottom up model

Spiral Model
Developed by Barry Boehm
A Spiral Model of Software Development and Enhancement

A risk-oriented software life cycle
Each spiral addresses major risks that have been identified
After all the risks have been addressed
The spiral model terminates as a waterfall software life cycle
Like waterfall model
Beginning with good specification of requirements
Iteratively completes a little of each phase
Start small
Explore the risks
Develop a plan to deal with the risks
Commit to an approach for the next iteration
Until product complete

Each iteration involves six steps
1. Determine objectives, alternatives, and constraints
2. Identify and resolve risks
3. Evaluate alternatives
4. Develop deliverables-verify they are correct
5. Plan the next iteration
6. Commit to an approach for next iteration
Boehm’s model contains a lot more detail
  Graph presents general concepts

Spiral model is an improvement on the waterfall model
Provides for
  Multiple builds
  Several opportunities for customer involvement
However it is
  Elaborate
  Difficult to manage
  Does not keep all developers occupied during all phases.

Concepts anticipate iterative approach
  That is central to co-design

Rapid Prototyping - Incremental
Used to develop a quick implementation of
Software and hardware
  Prior to or during requirements phase

Allows developers
  To construct hardware and software
  In incremental stages
  Each stage adds additional functionality

Each stage consists of
  Design, code and unit test, integration test and delivery

Prototype used several ways
  Developer
    Enables rapid development of major functionality of system
    Establishes structural architecture and flow of control
    Permits one to identify major problems early
  Customer
    Permits one to put functional unit
      Into the hands of the customer
      Much earlier than either the
      Waterfall or V cycle model

Customer uses the prototype
  Provides feedback to the
    Developers as to its strengths and weaknesses

Feedback is used to refine or change prototype
  To meet the real needs of the customer.
Prototype
- Can either be evolutionary or throw away

Has advantage of having working system
- Early in development
- As noted
  - Problems can be identified earlier
  - Provides tangible measures of progress
- Requires careful planning both at
  - Project management level
  - Designer’s level.

Caution
- Prototypes should never turn into product

As noted
- Each has certain strengths and weaknesses
Objectives of co-design approach
- Begin to attack some of these
- Simplify the development of systems
  - That are becoming increasingly complex

Co-Design
- Early computers based upon CISC architecture
- Work at IBM plus other places
- Led to RISC
  - Simplifying design
    - Both hardware and software
    - Increasing speed of computation
    - Simplify and improving performance of
      - Flow of control through machine
        - In particular flow associated with context switching

Simultaneously work continued on compilers for such machines
Developed ‘independently’ or serially
- Led to two designs – hardware followed some time later by software
  - Neither of which was optimal

Thought
- If development and optimizations
  - Could be executed simultaneously
- Could improve both

Concepts behind co-design
- Have been around for approximately 20 years
Started to gain credibility about 10 – 15 years ago
  First international workshop
Evolved from thoughts and research in several areas
  Recognition that microprocessor based systems
    Becoming increasingly important area
    For traditional IC designers
Software engineering
  And recognition that software
    Integrated and essential component in future chip designs
Work in synthesizing design from behavioural model

Objectives of co-design
  Gain control of design of hardware and software based systems
    Ensure greater predictability in meeting initial
      Goals and requirements
Objective to give designers tools to
  Assess and verify delivered system meets
    Speed, power, cost, performance, complexity goals
Evaluate many alternative designs
  At detailed level
  Without having cost of full implementation

Essential to co-design methodology
  Use of tools
    Use of computer based tools
We have computer aided
  Hardware and software design
    In reality co-design

Let’s now look at the embedded systems development process
Traditional Embedded Systems Development

As we know

Most embedded systems
Share a common structure
Also share a common development cycle
Through such a cycle
Develop the embedded design

Adjacent figure
Gives high level flow
Identifies major elements of such cycle

Specifically

Hardware design
Involves design of
Components
Printed circuit boards
System

Software design
May entail design of
High level language code
Assembly
Work in assembly
Requires detailed knowledge of
Microprocessor architecture
Its register structure
Mixture of assembler and high level

Current approach
Immediately following high level architecture
Immediately break system into hw and sw components
Develop each component separately
Generally hardware development
Lagged software development
Lead times on parts and fabrication of custom circuitry
But had to be finished first – before sw introduced

Approach limited hw-sw trade-offs
Interactions determined – discovered
Later in development
Late integration led to
Philosophy of let sw fix hw problems
Poor quality designs
Last second modifications
Difficult to maintain
Slipped schedules
   Possibly cancelled projects
Costly design changes

Diagrammed development process reflects
✓ High degree of concurrent hardware and software development
   Following specification of system architecture
   Such concurrency demands
       Greater co-operation between all designers

✓ Shorter times to market
   Mitigate against an extended trial and error approach
       Actually worked quite successfully during Soviet space program
   Increase demand that errors be identified and corrected early
   Requirements be very well
       Established and understood
       Prior to start of design
   Predictable schedule essential

✓ Increased reuse of legacy components
   Helps to reduce design and development time

Such observations naturally lead to a co-design approach

Let’s now take a look at the various elements of the co-design process
   In a bit more detail

**The Co-Design Process**

**Ideals**

   Transparent methods of modeling hardware and software
   Transparent design and analysis techniques
   Interoperability of tools and methods
   Seamless migration of pieces of functionality between hardware and software
   Iterative design approach
   Ability to quickly evaluate number of different hw-sw partitions
   Ability to evaluate system performance in unified design environment
   Ability to segue real hw and sw into models and simulations
   Ability to seamlessly move from high level models into real hw and sw
   Ability to work at multiple levels of abstraction
Process

As with the design of any kind of well conceived and developed system today Must begin with a specification Such a recognition holds Independent of any approach used

As a result Recognized that specification is necessary Major focus of co-design On the design and synthesis aspects Its domain is indicated in Boxed portion of the accompanying figure

If we consider design and development of Serious large scale systems Must consider System to be designed Environment in which it must operate

First view appears as

We see that our system comprises three aspects Hardware Software Gray area that may be either

For our system We have the necessary decomposition into modules and subsystems Components that must or should be either hardware or software Generally clear We say This part must be hardware or This part must be software

Power supply, display, communications port Necessarily hardware
Operating system, associated drivers
Can agree are necessarily software

Graphically we have situation expressed as seen in following diagram
We have a gray area
Between hardware and software
Where implementation approach not precisely defined

Co-design focuses on this area
Those components that may be either
In such cases we are making engineering decisions or trade-offs
Speed, cost, size, weight, and may others

Co-design emphasizes models – working from abstract to concrete
It is inherently a top-down process that flows as follows

Recall that the major elements of co-design process include
Specification
Modeling
Co-design
Partitioning
Co-synthesis
Co-simulation
Co-verification

In our discussions
We will go through each process step
Examine the critical points
Let’s naturally begin with the specification
Specification
Objective of specification process
Capture description of complete system
Such description
  Structured
  Understandable
  Verifiable

Our first focus must be on the world or environment
In which system is to operate

We follow with increasingly detailed description
Role played by the system in the application
At each step we add to the specification

We see our application must contain
  Model of containing environment
  Description / definition of inputs and outputs
  Description of necessary behaviour
  Description of how system is to be used

We start at a high level of abstraction
  With outside view of the system
  We develop the model(s) appropriate for that level

Through progressive refinement
  Move to lower level of abstraction
  More detailed model

Our specification must
• Be available before we can start formal design
  Ideally it should be executable
    Such an executable specification
      Serves as basis for validation of system

Some difficulties with (executable) specification
  Including non-functional constraints
  Integrating legacy components into an abstract
  Potential need to combine
    Different domain specific languages and semantics

• Be unencumbered by plans for implementation
• Focus on behaviour of system at high level
Formulating the Specification

✓ For the environment
  We must establish
  Description of all relevant entities
  Description of behaviour of all activities
  We must know
  How environment is interacting with our system
  Effect on environment of system output

✓ For the system
  Must have
  Description of all inputs and outputs
  Description of
    Functional behaviour
    Operational Behaviour
    Technological Constraints

We can naturally ask at this juncture
  How can we model our system and environment
  Without describing or knowing internal solution
  Internals are inherently unknown at this point
  How do we capture the desired behaviours

Starting point
  Requirements description and definition
  Such a definition describes the customer’s need

➢ The analysis of the environment
  Leads to a synthesis of reality
  In form of a model

➢ For the system
  We formulate the design from three perspectives

  • Functional view
    Defines system’s internal functions
    Relationships between those functions
  • Operational view
    Capture and express behaviour of those functions
  • Technological view
    Formulate the hardware solution to problem
    Identifies the components comprising solution
    Implementation of functional behaviour on the hardware
    Consistent with identified constraints
At this stage for our system we develop three types of specification

- **Functional Specification**
  Describes functions or operations
  To be performed by system on environment

  Specification
  Enumerates system functions for the application
  These are the external functions
  In contrast to internal implementation
  Description of the behaviour of the environment
  Under control of system for these functions

Pose and answer the question
How does the system affect the environment

✔ We view the system from the point of view of the user’s
  Needs and requirements

✔ We must observe or hypothesize
  What the system must do in its environment
  This is done in UML through Use Cases

✔ We must observe how the system interacts with
  Objects in its environment
  Such a view is purely external
  Knowing the environment means
  Modeling the objects without the system
  Understanding and describing the relationships between them

Gives us the bulk of our high order requirements

- **Operational Specification**
  Focuses on
  Behaviour
  Performance
  Information details
  Methods and approaches
  To be used in system
  Leads ultimately to the Design Specification

- **Technological Specification**
  Includes
  High level timing and timing constraints
  Geographic distribution constraints
  Characteristics of the interface
  Implementation constraints
Our model now takes on the following form

The Environment
We begin by describing world in which system must operate
Environment is temporal world
Heterogeneous collection of entities
Initial goals
Identify all such entities
Characterize their affects on system
Must have all necessary information about such entities
Must be in sufficient detail to support solving problem

Characterizing External Entities
Each external entity can be quantified by
Using method similar to CRC cards
Class – Responsibilities – Collaborators
Can view entities as characterized by
Name
Public interface
Inputs and outputs
Functional behaviour
We now have
• Name - Definition
  Data, Event, State variable, information, etc.
  • Responsibilities – Activities
    Activities, Actions, etc.
  • Relationships
    Relationships between
      Entity and responsibilities or activities
      Causal, responding, producer, consumer…
    System I/O and environment
The System

Characterizing the System

Characterization of system begins with
Identification of inputs and outputs

- System Inputs and Outputs
  System interacts with real world
  Through entities described and defined
  In environmental characterization
  Inputs to our system
  Outputs from environmental entities
  Outputs from our system
  Inputs to environmental entities

We can thus see that system I/O has already been characterized
In environmental entity specification
For each such I/O variable we know
  Name
  Use
  Input or output
  Nature
  Event, data, state variable, etc.
  Functional definition
  Structure, domain of validity, physical characteristics
  Technical constraints

- Functional Specification
  Again referring back to
  Objects
  UML use cases
  Before system is designed
  Appears as a black box
  Can only be viewed from external point of view
  Functional specification
  Defines external behaviour of system
  How system is used by user
  Equivalent to developing a model of it
  Functional specification therefore gives
  Characterization of affects of
  System outputs on environmental entities
  System response to inputs from environmental entities
  One such model can be a UML state chart and activity diagram
  Must ensure
  Modeled states
  Relevant to application
  Actions associated with system I/O
  Necessary to express functional specification
Conditions
Use only
System inputs
Specification states
Internal events
Appropriate time demarcation
Relative or absolute

• Operational Specification
  With respect to the operational specification
  Operational means
    Manner in which a function must operate
    Conditions imposed on the operation
    Range of operation

  Such a spec must consider
    Concrete numbers –precisions and tolerances- quantifying
      All variables in functional specification
      All operating conditions
      All ordinary and extraordinary operating modes

  Known information
    May contribute to producing and evaluating a design
    May include domain specific knowledge
      Proprietary or heuristically known to customer

• Technological Specification
  Technological specification
    Includes all specifications relevant to
      Hardware and software design

  Can easily identify 7 areas that should be considered
    1. Geographic constraints
      For distributed applications must consider items such as
        Topologies
        Communications methods
        Restrictions on usage
        Environmental contamination
2. Electrical considerations for interface signals
   Characteristics and constraints on any electrical I/O signals
   These are driven by external environment
   May be beyond control of designer

3. User interface requirements
   System may have interface to external world
   Medical or instrumentation devices
   Must consider
   Presentation method(s)
   Protocols

4. Temporal considerations
   Systems may have real-time constraints imposed
   Hard or soft
   Constraints may specify delays on
   Signals originating from external entities
   Responses to system outputs by external entities
   Internal system delays

5. Maintenance, Reliability, Safety
   System may have requirements for
   Diagnostic tests
   Remote maintenance
   Remote upgrade
   Must have concrete numbers for
   MTTF and MTBF
   Environmental and safety issues
   Must address performance
   Under partial failure
   Full failure

6. Electrical considerations
   Electrical characterization of internal signals and behaviour
   Power consumption
   Supplies
   Tolerance to degraded power
   Characterization of signal
   Levels, times, frequencies, etc.

Modeling Tools and Languages for Codesign
   Specifying and modeling complex contemporary embedded designs
   Draws upon wide variety of tools and methods

   Solid models as well as modeling and specification languages
   Essential for comprehending, expressing, managing, validating
Important to keep in mind
No one tool does everything for all phases of the development process
Codesign process utilizes
Variety of models
At different levels and for different reasons

For structural and hierarchical aspects
We utilize system description languages like
 SDL, SpecCharts, VHDL, Verilog, Esterel

Digital signal processing requirements may be met with
C/C++ or Matlab

Control aspects are addressed by
Variations on FSMs

Have already encountered system level model
Such a model is necessarily behavioural

From system level we move to an architectural level model
Subsequently to detailed level design models

Will look briefly at
Motivation for modeling
What we are modeling
Essential characteristics for modeling method
Examine several different models and modeling/specification languages
Try to point out utility and limitations of each

List is neither comprehensive nor complete

Why are We Modeling?
Primarily we use models to represent a description of
Real system or one that will become real
When it is designed

Models give us different views of our system
External, internal, abstract, behavioural, structural…

Model gives us means to describe characteristics of system to be designed
Provides basis for later verification

Models are cheaper than building complete system
To test design concept

Models allow us to execute test that may be too hazardous to run
During preliminary development

In design process
Model precedes actual design
Provides opportunity to quickly explore variety of alternative approaches
Cheaply
Quickly

What are We Modeling?
To effectively formulate a good model
Must understand what we are modeling

Our target is embedded applications

We know that embedded systems are
- Reactive
  System runs continuously
  Responds or reacts to signals from external environment
- Often real-time
  Time constraints imposed on behaviour
- Heterogeneous
  Composed of hardware and software pieces
- Supported by different development environments

We need to distinguish
Model
Language used to express the model

Qualifying the Model
Restating – Expresses an abstraction of the real world
Intended to give an abstract representation
Portion of real world
Allows us to temporarily ignore certain details
As we gain understanding of problem

To be useful
We can hypothesize some essential general capabilities
- Abstraction
  Must allow us to express and examine behaviour
  Of complete system
  Unburdened by details of sub-components
- Refinement
  Must allow us to express and decompose behaviour of system
  At different levels of granularity
- Structure
  Must be able to express system as set of interconnected modules
• Communication
  Must support inter module communication method
• Should support synchronization method
• Easy to interpret
  Must express anticipated behaviour or aspect being modeled
  In comprehensible format

Two classifications of model are particularly useful
   Conceptual and analytic

Conceptual
   Precedes analytic
   Allow us to work at high level of abstraction
       Uses a symbolic means
       To capture qualitative aspects of problem
   Useful during early stages of design
       Formulating specification
       High-level architecture
       Early stages of partitioning the system
   Allow us to grasp and work with complexities of a design
       To focus on essential details while ignoring others
   Are behavioural in nature

Analytic
   Permits analysis at lower levels of detail
   Use mathematical or logical relations
       Express quantitative physical behaviour

   Useful during middle and later stages of design
       Later stages of partitioning
       Modeling and analyzing detailed architectures
       Verifying detailed performances
       Making performance trade-offs
   Are more structural in nature

A Look at Some Models
   To be effective
   Models should give us ability to express
       1. Modularity and hierarchy
          Should be able to express
          Static and dynamic behaviour
          Structural and functional construction
       2. Relationships among subsystems
          Should be able to express
          Sequential and concurrent flow of control
          Inter subsystem synchronization and communication
3. Communication amongst tools  
4. Use of legacy designs or behaviours  
Models should be executable  
Later this is how we verify the system throughout design process

Let’s look at some of the more commonly used models and languages

**Finite State Machine**

Gives simple behavioural description  
Hardware or software system  
Its input / output function or relationship  
Computed by a finite automaton  
Expressed as directed cyclic graph  
Nodes in graph  
Define discrete the states in the modeled entity  
Arcs or edges  
Labeled with input / output data pairs  
Reflect changes in state in the modeled entity  
Typically one of comprising states  
Defined as the initial state

Two models – Mealy and Moore  
Distinguished by output function  
Mealy – output function of current state and input  
Moore – output function of state only

We can express the basic machine as follows

**Limitations**

There is a theoretical limit on computational power  
Has limited useful memory  
Using states not efficient  
State space explosion for large problems  
Impractical for large numbers of inputs

When it’s necessary to express concurrent activity  
Must use combination of several machines  
Coupled by transition conditions between them

Under such conditions  
Once again we get a state space explosion

The nature of the FSM  
Makes it difficult to continuously refine
FSMs are inherently synchronous
System is in single global state at each time instant
If we design complex system
By interconnecting one or more machines
Outputs to inputs
Must have some from of synchronism
Common clock
Handshake

There are a number of variations of basic FSM
At core of different models and modeling languages
Each addresses one or more of the limitations with this model

Communicating Finite State Machines
To begin to address some of limitations of FSM
Examine several examples of what are called
Communicating finite state machines

Extended FSM
A first example of a communicating FSM
Called Extended Finite State Machine

EFSM model expressed as
Network of FSMs with
Communication mechanism
Used to orchestrate operation of model

Communication channel is non-destructive
Event can be read many times by receiver
Simplest form is familiar shared variable

Communication is synchronous

Modifications to basic FSM include
1. Add a (set of) variables or (FIFO) queues
   Variables have name and can hold abstract objects
   Type of abstract object restricted to integer
2. Restrict use of queues to transferring integers
3. Add collection of logical or arithmetic operators
   Operate on contents of queues

We see
Next state is function of present state, inputs, and variables
Output also becomes function of variables
Variables computed as function of present state, inputs, and variables
The extended model appears as follows

Descriptive power of EFSM is equivalent to Turing machine

Limitations

Too low level to easily support specification
Generally formal specification languages more effective
Esterel, SystemC, VHDL, Verilog, Statecharts

Codesign FSM

Second such model is called Codesign FSM
To address issues with the synchronous model
Underlying concurrent FSMs
Such a model implies all comprising machines
Change state at same time
Software implement in single processor environment
Requires interleaving actions in time

Codesign FSM addresses issues of timing and coordination

Similar to EFSM
CFSM model expressed as
Network of FSMs with
Communication mechanism
Used to orchestrate operation of model

State Changes
State changes in comprising FSMs
Asynchronous with respect to each other
Time to compute next state change
Can be different in each machine
Can be unbound in limit if implementation not known
Events

Communication mechanism based upon timed events

*Event* defined by triple
- Name of the event
- Value
- Positive integer denoting time of occurrence

Event transfer uses unacknowledged protocol
Receiver does not acknowledge receipt

Two kinds of events defined
- **Trigger events**
  - Provide basic synchronism mechanism in CFSM
  - Used one time
    - Called *destructive* read
    - Cause state transition is target machine
- **Pure Value events**
  - Cannot cause state change directly
  - Can be used to select from alternate possibilities
    - Involving same set of trigger events

Operation

Operation of CFSM comprises 4 phases
1. Idle
2. Detect input event
3. Execute transition according to
   - Which events are present and
   - Match associated transition relation
4. Emit output event

Consequently
- State transition triggered by input signal
- Reflected sometime later in output change

Timed trace of events is
- Ordered sequence – in time
  - Time is monotonically non-decreasing
- No two events with same *name*
  - Can occur simultaneously
    - Implies that no communication channel
      - Can carry two *values* at same time

Globally asynchronous nature
- Facilitates partitioning system model
  - Into hardware and software components
Low level structure of individual CFSMs
Makes synthesis of hardware or software rather straight forward

Limitations
Communication mechanism very specific
Does not easily support
Complex data types
Data transformations
Can model other schemes
Blocking messages
Shared variables
Can be cumbersome however
Similar to EFSM
Too low level to easily support specification
Generally formal specification languages more effective
Esterel, SystemC, VHDL, Verilog, Statecharts

Program State Machines
Program state machine extends FSM by integrating
Hierarchical concurrent FSM
Program language concepts

State
State represents distinct mode of computation
Only subset of program states can be active at any one time
Carrying out associated computation or operation

Program state can be
Compound
Made up set of program states
Such states may be executing in sequence or in parallel
Similar to statechart
Elementary
Modeled as a sequential algorithm in imperative language
Assignments are only statements in such a language
Responsible for change in state

State Transitions
Transitions between sequential program states of two types
1. Transition immediate arcs
   Transition occurs as soon as condition becomes true
   Independent of state of sub-program states
   Active – inactive
   Computation complete or not
Intended to address
  Reset or exception conditions

2. Transition on completion arcs
  Transition occurs when state has finished activity

  Similar to signal-wait and signal-continue semantics in monitors

Limitations
  Communication and synchronization between concurrent states
    Modeled through shared memory
  No other means provided

Statechart
  Statechart can be viewed as graphical specification language
    Powerful tool for modeling and specifying
      Dynamic behaviour of reactive objects

  Extends basic FSM by supporting
    Hierarchical decomposition or refinement
    Concurrency
      Very useful for expressing
        Simultaneous sequential behaviours in control systems
    Notion of delays and timeouts
    A history mechanism
      Allows one to re-enter state at same point left

  Discussed in detail in earlier work on UML

Petri Nets
  Petri net useful for expressing and validating behaviour of system
    Like statecharts they support concurrency thus can be used to express
      Sequential or parallel behaviour
    Support behaviour hierarchy

  Petri net is a bipartite graph
    Comprised of
      Nodes or places and
        Represented by a circle
        Correspond to conditions that may hold in the system
        Place may or may not have token
      Transitions
        Represented by a rectangle
        Express events that may occur
      Tokens
        Signify assertion of associated condition
Expressed as a small solid circle or dot within a place

Following illustrates several basic structures

**Operation**

Petri net is marked by placing tokens on the various places
When all incoming arcs to a transition have a token
   Transition fires
      Corresponds to occurrence of associated event
   All tokens on input arcs removed
   Token placed on all outgoing arcs
   Result is change in state of system

**Limitations**

Like FSM
   Petri net is flat
      Difficulty in expressing structural hierarchy
   Complexity rapidly increases
   Communication between parallel threads
      Only through shared variables

**Kahn Process Network**

Model is similar in structure to
   Extended FSM and Codesign FSM

**Structure**

Processes like finite automata
   Perform mapping from input sequence to output sequence
   Networked collection of processes
   Supports distributed control
   Supports distributed memory
Operation

Processes run autonomously
Communicate via tokens through unbounded FIFOs
  Like CFSM
    Token can only be written or read once
  Writes are nonblocking
  Reads are blocking
    Synchronize through a blocking Read operation
Process is either
  Executing
    Execute command
  Communicating
    Send or Get command
Behaviour is deterministic

Control Flow – Data Flow – CDFG Graphs

Control and data flow graphs
  Provide another means to express
    Control and data flow in a system
Have explored already under discussion on structured design

Basic organization comprises
  Control and Data nodes

Control nodes interconnect to capture
  Expected control flow operations
    Sequences, branches, concurrent operations, loops
  Each linked to data flow block

Data flow block
  Encapsulates
    Set of data computations
  Each computation sequence expressed as
    Directed acyclic graph

Data node expresses
  Arithmetic, logical, relational operation or
  Read or Write to memory or external port

As with statecharts
  Data and control flow graphs were extensively covered earlier
Languages

Well beyond current scope to delve too deeply
  Into numerous modeling and specification languages available

Will introduce and comment on several of more common ones

System Level

Will frequently be using a
  Graph of one form or another with communicating processes

Problems will be
  Reactive real-time systems
  Often have strong control component
  May include DSP

We are focusing on transactions

Hardware Languages

Signal Processing Worksystem
  Cadence Design Systems
    Intended for design and development of
      Signal processing circuits and systems
      Supports hierarchical block diagram approach
    Can import models written in variety of different languages
      Matlab, C/C++, SystemC, Verilog, VHDL

Supports reuse of legacy code

Statecharts
  Have already discussed these

Java
  Should be familiar

SDL

Specification and Description Language
  Object oriented formal language
  Defined by International Telecommunications Union-Telecommunications Sector (ITU-T)
  Language is
    Standardized internationally
    Graphical and symbol-based
      Simplifies use

Intended for specification of
  Complex, event driven, real-time applications
    Typically involving
Large number of concurrent activities
Communicating using discrete signals
Target applications may be local or distributed

Works well with other modeling and specification languages
OMT/UML
CORBA
Code generator available for C/C++

System Studio
Formerly COSSAP
Block diagram oriented tool for DSP design
Capability focused towards SoC designs emphasizes
  • Algorithm design
    Targets the DSP type applications
  • Architecture design
    Targets higher level design of
      Processor, memory, bus combinations on silicon
Supports high level design
Verilog, VHDL, SystemC

Software Languages
At system level on software side we have

SA/RT
Structured Analysis for Real-Time
Adds real time extensions to structured analysis/design methods
Considered to be an informal language
  Notations are imprecise and ambiguous
  In formal sense
Most commonly used version defined by
Hatley and Pirbhai
Intended to add support for
  Concurrency, synchronization, interrupts, etc
  Missing from traditional structured methods

SA component supports
Data flow diagrams
Process hierarchy
Means to express process specification
  Decision tables or decision trees
Structured text
Formulas
Diagrams
Labeled graphics
Data dictionary
RT component adds
  Control flow diagrams
  Means to represent a finite automaton
    State transition diagrams
    State transition tables
    Decision tables
  Means to represent time

Esterel
  Is an executable specification language
    It’s a programming language
      Intended for programming reactive systems
    Can generate either a software or hardware implementation

  Also is compiler used for translating
    Esterel programs into FSM

Synchronous Languages
  Esterel belongs to family of synchronous languages
    Signal, Lustre, and StateCharts
  Synchronous modules
    Perform read, write, compute
      In single time instant

    At conceptual level synchronous languages assume
      Computation and information exchange
        Occur in zero time

Esterel Model
  Esterel follows synchronous model
    Except incorporates time model
  Application modeled as concurrently executing modules
  Components communicate with each other and external world
    Through signals
  Denoted *Globally Asynchronous Locally Synchronous*
    Consistent with earlier discussion
    Inter module communication loosely coupled
    Intra module communication tightly coupled
Supports number of essential model characteristics

1. Decomposition
   System behaviour can be decomposed to the algorithm level

2. Structural hierarchy
   Can express system as set of interconnected modules

3. Behaviour hierarchy
   Can decompose a behaviour into distinct sequential or concurrent sub-behaviours

4. Communications
   Supports communication structure between processes

5. Exceptions
   Supports instantaneous communication with the system and can abort on occurrence of a specified condition

6. Synchronization
   Can support need to synchronize two processes uses

   \[ \text{await} \ <\text{signal}> \]

SDL
We’ve discussed already

Matlab
We’ve discussed already

Java,
We’ve discussed already

Algorithmic Level
At algorithmic level
Interested in steps in computation
Model often based on some form of
Data or control flow diagram

Hardware Languages
In addition to familiar HDLs
Verilog and VHDL

SystemC commonly used
Has become industry driven de facto standard
Language is C++ class library
Permits design of mixed hardware and software systems
At different levels of abstraction
Structure of the language
Appears as shown

Software Languages
Software support languages at algorithmic level
Familiar C, C++, and Java
Techniques of HW/SW Codesign

We’ve looked at the first several of steps in the codesign process

Let’s examine how we execute the design itself

Our concerns now are

- Partitioning the system into hardware and software components
- Refine inter process communication
- Synthesize hardware – software interfaces
- Co-synthesis
  - Software synthesis
    - Target specific tasks to hardware components
  - Hardware synthesis
    - Decompose computation steps into clock cycles
- Co-simulation

Let’s begin with the partitioning problem

Partitioning

Our objective is to

Allocate operations comprising system behaviour to
  - Hardware or software

Can view system being designed as comprising following sets

Due to Kumar
- Set of available software functions
- Set of hardware resources
- Set of communications between the hw and sw units
- Set of functions to be implemented
  - Such functions can be assigned to
  - Hardware, software, or communications

To these we add
Due to Vahid
- Hardware size specification
- Set of performance constraints
  - Functions to be designed and implemented
  - For real time systems
    - Such constraints are temporal

Goal

Find a partition that
- Satisfies performance constraints
- Minimizes hardware size
Such an assignment
- Determines the speed of the operation
  Can also be viewed as delay through the associated function
- Potentially induces additional delays
  Resulting from communications overhead
  As we learned earlier in discussions of traditional system design
  Goal is to decompose system into
  Loosely coupled – highly cohesive modules
  Same goal holds as we explore alternate partitions of our system

- Affects system performance
  As more software tasks allocated to processor
  Increases processor utilization

  Thus one measure of performance as we saw earlier
  Affects available bandwidth of bus
  Between processor and
  Peripheral hardware
  Accelerator hardware
  Memory system

Partitioning Methods

Variety of different methods to attack partitioning problem
We consider two extremes

*Software Oriented Model*
  Initially put everything into software
  Move time critical pieces of functionality to hardware as necessary
  To meet time constraints

*Hardware Oriented Model*
  Initially put everything into hardware
  Move non-time critical pieces of functionality to software as appropriate
  To meet time constraints

Let’s begin to codify the various approaches
  According to main properties or characteristics

- The model used
  Typically system expressed as a graph
  Derived from abstraction of system specification
- Cost Function
  One metric for assessing a partition
  How does such a partition impact system performance
Suggests devising a partition *cost function*

Partitioning cost function
Guides partitioning algorithm towards
Minimizing such a function
In theory best partition is one that yields such a minimum

Such a function should measure
Consequences of hw/sw allocation
Bounded by two extremes
All hardware and all software
Affects on
System timing behaviour
Power consumption
Memory loading

Partitioning in
Software
Looking at statistical behaviour of executing program
To drive the algorithm
Have some flexibility
Hardware
Looking at static allocation of functionality

- **Granularity**
  Usually based upon elements of the specification language
  Processes, tasks, loops, etc.

- **The Method**
  How is the partitioning executed
  What is the algorithm
  Aggregating or grouping elements
  Iteration
  Greedy heuristics
  Builds solution sequentially
  Picks ‘best’ local decision
  No consideration for consequences
  Best is subjective
  Mathematical relations

Let’s now look at several different partitioning methods
Functional Partitioning
Due to Mahapatra and Vahid
Numerous codesign tools exist for addressing partitioning problem
*Functional partitioning* is one such approach

Functional partitioning
- Refines system’s functional specification
  - Into multiple sub-specifications
- Each such sub-specification
  - Expresses functionality of a component within the system
    - Either hardware or software component
- Resulting components
  - Synthesized into gates or code for target machine

Top level view
Viewed at abstract level
- System to be partitioned interpreted as
  - Collection of procedures
  - Can model as set of procedures with single top level procedure
    - \( F = \{f_0, f_1, \ldots f_{n-1}\} \)
- Procedures structured into call graph
  - Each node is a procedure
  - Each arc is procedure call
- Procedure call
  - Modeled as simple processor with read and write capability
    - Recall our earlier discussion of FSM based models
- Execution of set \( F \)
  - Modeled as procedures executing sequentially
    - No concurrent operation

Objective is to map procedures to
- Software program
- HDL program

Functional partition operation
- Creates partition \( P = \{p_0, p_1, \ldots p_{m-1}\} \)
  - Containing \( M \) parts or groups \( p_k \)
    - \( M \) is of size 2 or more
    - Each part or group may be
      - Hardware processor
      - Software processor
      - ASIC or other programmable logic device

Such that each \( f_i \) is mapped to one and only one \( p_k \)
- As we see in the following figure
We have
\[ F = \bigcup p_k \quad \text{and} \quad p_i \cap p_j = 0 \quad \text{for all } i, j, i \neq j \]

Each \( p_k \) implemented on single abstract processor
Abstract processors
Are mutually exclusive
May exist on same
Physical hardware device or software procedure

System operation after partitioning
Same as prior

Subsequent to partition
No restriction placed on synthesis process
May implement process’ procedures in parallel
Provided that data and timing constraints satisfied
Procedures not mutually exclusive after partitioning
Processors remain so

Tasks
Tasks appropriate to sound partitioning
Create the model of the system
Instantiate appropriate processors
Partition the system among the processors
Estimation
Gives information used for design metrics
Prior to and during partitioning

A Three Step Partitioning Method
Vahid proposes a three step partitioning method
Given in accompanying diagram
Prior to application of partitioning algorithm
Useful to guide process
To reduce design time or runtime of partitioning algorithm

Goal in preliminary decomposition
1. Enter with the grain as large as possible
   Supports
   Maximum pre-estimation of partition
   Application of heuristics for segregation

2. Components of system included in procedure
   Only if segregation would yield inferior solution
Granularity Selection

Key component in partition trade-offs is Granularity of procedures
 Entered into N-way partitioning algorithm

Clearly fine grained procedures
 Give greater
 Visibility into system under design
 Flexibility in allocation to blocks comprising the partition
 Can be very computationally intensive
 Potentially can limit heuristics that require iterative estimations
 Of partition

Course grained procedures
 Can reduce run-time complexity
 For sophisticated partitioning algorithms
 Grouping behaviours into atomic unit
 Eliminates partitions that may benefit from separating them

There are some transformations
 That can help to guide the granularity selection problem

- Inlining
  Replace procedure call by the contents
  Can make granularity coarser
  Inlined procedures may be eliminated
  Increases code size

- Exlining
  Converse of inlining
  Sequences of statements
  Replaced by procedure containing only those statements
  Distinguish two types
  Redundancy
  Two or more nearly identical procedures
  Replaced by one
  Computation
  Large sequence of statements
  Refined into several smaller procedures
  Statements within a procedure
  Highly correlated
  Ensures that group of related statements
  Not decomposed during N-way assignment

Moves towards finer granularity
• Cloning
  Make a copy of a procedure
  For use by specific caller
  We make a trade off between
  Code space
  Communication needs
  Efficiency

Pre-Clustering

Next step is similar to but different from defining a granularity

Goal now
  Reduce number of procedures that need to be allocated to groups
  Merge those procedures whose separation
  Would never lead to good solution
  That is those that should never be separated
  In a good partition
Consider following
  Let procedure A contain 25 statements
  Let A be called from 10 separate locations in procedure B
    Significant calling overhead between potential groups
  Inlining A in B
    Will add 225 new lines to program
  Cloning leads to similar problems
  Better to combine into same cluster prior to partitioning

Pre-clustering different from granularity definition
  Procedures under consideration to be clustered
  May not be able to be exlined
  Calls to such procedures not adjacent

Pre-clustering also different from the N-way assignment
  Each cluster does not represent a
    Hardware, software, or other group
  Thus not clustering operation not guided by normal design metrics

General algorithm for pre-clustering
  After granularity selection
    Each procedure expressed as node in a graph
    Graph is fully connected
      Each edge weighted with closeness of associated nodes
        Similar to routing in telecoms networks
    Closest pair merged into new node
    Merging repeated until no pair has closeness measure
      Below specified minimum
Closeness is an empirical measure
Examples include
  Communication bit density between nodes
  Shared hardware
  Shared procedures
    One in each set for example

*N-Way Partition*
Objective
  Allocate procedures to set of processors

Variety of heuristics may be used to create initial solution
  Statistical techniques
    Random distribution
    Simulated Annealing – Hill Climbing
      Avoid local minima with long runtimes
    Genetic algorithm methods
  Greedy heuristics
    Linear time heuristic
      Moves nodes to reduce cost function
  Port calling
    Seeks to balance access to shared ports

Guiding heuristic tries for balanced allocation

*Graph Based Models*

*Due to Ernst and Henkel*
Model associates nodes with elementary operations
  Such operations derive from statements in associated specification

Process begins with all software implementation
  Initial step is to identify bottlenecks that can be eliminated
    By migrating selected operations to hardware
  Initial step followed by closeness estimation between operations
    Data – explores number of common variables among operations
      Loop index has
        Initialization
        Increment
        Compare
    Control – examines distance between invocations of same operation
      Immediately successive operations
    Operator – similarities between operators
      Add and subtract, shift left or right
Estimate communications overhead
When elements moved or exchanged among partitions
One measure can be density of data exchange amongst partitions

Partitioning implemented by two nested loops
Inner loop uses statistical methods
   Simulated annealing
Outer loop
   Synthesis based to refine inner loop estimates

*Due to Barros, Rosenstiel, and Xiong*
Method uses a fine grained graph based model
Nodes represent statements in specification language
Nodes are clustered according to variety of metrics
   Concurrent operations
      Control and data flow are independent
   Sequential operations
      Control and data flow are dependent
Mutual exclusion
Aggregation of sequence of related assignments

Clustering minimizes cost of cuts in clustering tree
Partition is iteratively improved by examining
   Pipelining
      Re-evaluating previous assignments
   Resource sharing

Other Partition Schemes
Some schemes include
   Vulcan
      Allocation among packages
         Uses iterative improvement heuristics
   Chop
      Allocation among packages
         Utilizes suite of feasible solutions
            That satisfy overall constraints
   Polis
      Using interactive approach
      Supports variety of environments for
         Formal verification
            Co-simulation
Evaluation of a Partition
With each partition
Must revisit our boundary conditions – we ask
- Is performance improved when piece of functionality
  Moved from software to hardware
- Is hardware complexity and size improved when piece of functionality
  Moved from hardware to software

Co-Simulation
Material based upon
Polis and Ptolemy projects at UCB
Mahapatra at Texas A&M
Lavagno et.al.

In early days of embedded design
Hardware always lagged the software
Design errors and flaws often remained hidden
Until later stages of the project development

Today partitioning and simulation go hand in hand
Using behavioural model simulation conjoined with other simulation tools
Can produce much better simulation earlier on in project development

Simulation – in its many forms – remains one of our best tools
For validating evolving design
Against our incoming specification

Recall that co-design is iterative process
Part of the process involves partitioning system into
Hardware and software parts
To ensure a good partition
Must simulate against initial specification
Use simulation results to modify partition
Repeat process until satisfactory solution reached

We see this in following diagram
Reflects portion of Polis design flow

Let’s examine the co-simulation block first then the synthesis block

If we examine the flow above
Cannot afford to build hardware platform for each iteration
Try out the design
Make adjustments
Try again
Simply too expensive and takes too much time
We want to be able iterate through process number of times
Refining the design with each iteration
Desideratum
  During early stages of design
    Want maximum flexibility
  During later stages of design
    Would like to be able to introduce physical hardware
      Processors
      FPGAs
      ASICs
    Of these ASICs most difficult since take most time to build
    Certainly could have processors and FPGAs earlier

To satisfy our objects
  During early stages we use simulated hardware running real software
  Later we migrate towards real hardware

Problems and Goals
  We want to execute the software as fast as possible
  Want to keep both simulations synchronized
    To ensure proper performance in ultimate target platform

To accomplish such a goal
  One could simulate the target hardware design
    Running on general purpose HDL based hardware simulator
  Execute software solution
    On such a simulation or
    On a host that is faster than the target platform

For larger systems
  Such an approach becomes quickly impractical

Tools
  Co-design tools enable today’s designers
    To work at high level of abstraction as well as at detailed level
  Gain high level of confidence that
    When system ultimately fabricated
      It will perform according to initial specifications

Our typical simulation platform comprises
  The computer
  Hardware and software modeled as a process
Hybrid environments
Add co-processors
Assume part of the computational load
Remainder remains in software

We distinguish between simulation and emulation
With *simulation*
We model the software and hardware
We make best effort to duplicate real system

With *emulation*
We augment with hardware as necessary
To ensure that we meet all timing requirements
Usually done with FPGAs or some such
Emulation becomes essential when trying to identify
Higher speed time critical operations
If we are designing a system to operate in the GHz region
We’re not going to do that with a sw simulation
Under such circumstance
We are using a hybrid or extensible simulation
Emulator based or driven simulators
Have difficulty with behavioural based models

Types
We identify several major types of simulation
Gate level
Cycle based
Event driven
Data flow

We may use several different simulators or simulation methods
During course of design
Always choose the best tool for the job

*Gate Level Simulations*
Tries to replicate as much as possible
Gate level implementation behaviour of CPU
Method is highly inefficient
Impractical for larger systems

*Cycle Based*
Simulate bus interface and associated timing
Program executed on high level instruction set interpreter
Signal state evaluated at clock edges
Positive or negative going
Provides information about
Number of clock cycles necessary for given instruction sequence
   Between I/O operations
Used most often in large complex designs
   Have significant number of tests/ necessary computations
Approximately 10 times faster than event driven approach
Such simulators have difficult with asynchronous designs
   Suffer severe time penalty

**Event Driven**

More accurate than cycle based
   Value of every active signal
      Computed for every device during clock cycle
         As signal propagates
A signal is simulated for
   Value and temporal behaviour
Used most often when
   Timing analysis must be performed
      Test for race conditions and their consequences
Clearly very computationally intensive
   Requires significant computational platform
      If to be completed in reasonable time

**Data Flow**

As name suggests
   Focus is on flow of data
Generally used for high level simulation
   To verify overall functionality
      Correctness / validity of algorithms
Signals expresses as stream
   No notion of time
Functional blocks are linked by signals
   Execute when data is present
      See Petri nets
Simulator scheduler
   Determines order of execution

**Approaches**

There are various approaches we can take to
   Executing co-simulation
In such an execution
   We must address several issues

- How do we simulate the components comprising a mixed environment?
- How do we simulate all components at the same time?
- Software simulator typically runs faster that the hardware one
How do we synchronize the two?

Let’s look at a couple ways to address these issues

**Detailed Processor Model**

Processor components simulated by event driven model
- Includes memory, data path, bus, instruction decoder, etc.
- Interaction between processor and other components
  - Again uses event driven model
- Gate level simulation
  - Done using behavioural model
  - Event model too slow
- Our model looks like the following

![Diagram showing Detailed Processor Model]

**Cycle Based Simulation – Bus Model**

- Uses discrete event shells to simulate bus interface
- Not concerned with executing software associated with processor(s)
- Focus is low level bus – memory types of interactions
- Software simulated using an ISA model
- Provides timing information – in clock cycles
  - For given sequence of instructions
  - Between I/O operations
- Trade speed for accuracy

Detailed model now extended as

![Diagram showing Cycle Based Simulation – Bus Model]
**Instruction Set Architecture Model**

Simulate the instruction set in C
C program then acts as interpreter for embedded application

Software is executed on the ISA model
Model provides timing details for co-simulation

Can be more efficient than detailed model of processor
Don’t need high overhead of discrete event model

Detailed model now modified as

![Diagram of Compiled Model]

**Compiled Model**

Again attacking hardware speed problem
Assuming a slower target processor
If we are trying for more accurate sw timing model
Translate embedded sw specification to native code
For processor upon which simulation being executed
Software execution on host
Provides timing details on interface (to sw) to the co-simulation
Accuracy depends upon interface information

Our model is modified as follows

![Diagram of Hardware Model]

**Hardware Model**

If target processor(s) available or can be modeled using FPGA
Physical hardware can be incorporated into simulation
Ideal model during later stages of development
Substantial improvement in simulation speed
Disadvantage is that such hardware must be available
We now have

We’ve seen several approaches to co-simulation

These work well when focused on one or a few aspects of process

We’ve commented several times earlier
Want simulation to be able to follow design evolution
As design matures
Tools should be able to support evolution

We are suggesting that co-simulation support
Follows several levels of abstraction
During early stages
Hardware and hardware synthesized models not available
We use functional model to make hw/sw tradeoffs
During middle stages
Functional model segues into netlist model
Later
Hardware confirmed
Can migrate back to higher level model for hw

Master Slave Model
One approach to attack problem
Decompose simulation into pieces
As we might do with a network
We architect a simulation comprised of
Master simulator
Several slave simulators

Master invokes slave by procedure call
As necessary
With such a scheme
We compromise concurrent simulation

Such a model appears as
Distributed Co-Simulation

Alternate approach utilizes more distributed simulation
Could easily envision widely distributed model
Pieces of simulation residing at different
  Vendors
  Universities
  Companies
All conjoined into sophisticated simulation
  For very complex problems
Architecture utilizes co-simulation (software) bus
  Transfers data between simulators
    Uses procedure call
      Number of such schemes available
Bus can take many forms
  ‘Local’ bus
  Ethernet
  Wireless
Can support concurrency amongst simulators
Managing time is critical element
  How do we ensure that simulation is synchronized

Such a model appears as

Heterogeneous Modeling – The Ptolemy Project

Current incarnation of this environment is Ptolemy II
Designed as an environment for
  Simulation and prototyping of heterogeneous systems
  Uses strengths of object oriented methodology
    For modeling each subsystem efficiently and naturally

System uses Java for its object capabilities
Designed to take advantage of several capabilities built into Java
  Network integration
    Can support distributed simulations
  Migrating code
  Built-in threading
  User interface capabilities
Uses XML for persistent data representation
    Designs are expressed in XML

Sophisticated type system
    Supports type inference and data polymorphism
    Components can operate on multiple different data types

Behaviour types
    Components and domains have interface definitions
Support static and dynamic structure

Ptolemy environment introduced
    Notion of domain polymorphism
    Components can be designed to
    Operate in multiple domains
    Interact with other components in wide variety of domain

Domains
    Key to heterogeneous approach
    Support for wide variety of different design styles
    Real-time and distributed computing
    Distributed discrete events
    Timed multitasking

Such styles called Domains

Domain implements computational model or model of computation
    For each particular type of subsystem comprising a design
    Referred to as MOC
    MoC is semantics of interaction between
    Modules or components
    Provide means to support concurrency and time
    In different ways

Supported domains include
    Synchronous data flow
    Data driven and statically schedule
    Dynamic data flow
    Data driven and dynamic schedule
    Discrete event
    Digital Hardware Model
    Uses a functional simulator developed by Stanford
    Supports models ranging from
    Detailed gate (structural) to behavioural
Classes of MoCs

Component Interaction
Models systems that mix
   Data drive and demand driven
      Styles of computation
Client- server can be viewed as demand driven
Push-pull interaction between producer and consumer
   Exhibits data driven behaviour

Communicating Sequential Processes
   Ptolemy expresses such processes as Java threads

FSM
   Enumerates states
   Rules for transitioning between states
   Execution is ordered sequence of state transitions
      Utilizes the notion of guards
      Constrains when transition taken
   Work well in control type applications

Dataflow
   Modules react to data available on inputs
      Perform computation on data
      Emit data on output

Process Networks
   Communication is via streams of tokens
      Each token is arbitrary data structure
         Operated on by the MoC
   Models processes that communicate via
      Buffered message channels
   Models are loosely coupled
      Provides easy support for
         Concurrent or distributed operation

Synchronous data flow
   Special case of more general process networks
      Restrictions are on understanding of
         Deadlocks and bonds on operations
   Modules execute sequentially to complete task
   Supports computations that operate on streams of data token
   Schedule of operations is statically determined
Discrete event

Modules react to events occurring at instant of time
Event comprises
Value and time stamp
Applicable to systems that
Respond to input events
Produce output events
Either at same time
Future time instant

Continuous time
Models components that interact
Through continuous time signals
Provides support for analog aspects of some circuits

Co-Synthesis
Material based upon
Mahapatra at Texas A&M
Lavagno et al

As we saw in earlier diagram
Partitioning, simulation, and synthesis
Work in cooperation
Also at the end of the day
Must implement the system we are designing

Let’s now look at the synthesis component

Synthesis is a wide open problem
Many different commercial as well as research based tools
Many of the commercial tools focus on the hardware

We’ll look at general considerations
Without delving into details of specific tools

Synthesis entails
Turning HDL code into hardware
Software into firmware

System
Our system comes with
Specification
Set of hardware and software resources
Mapping – via the partition
Onto an architecture
Our target architecture comprises collection of black boxes

These black boxes charged with executing the hardware and software tasks
Within the evolving design

Hardware comprises
- Microprocessor, Microcontroller, Microcomputer
  - Certainly could have multiple copies of each
    - For now consider single copy
Set of hardware modules including
- Memories
- I/O devices
- CPLDs, ASICs, FPGAs

Software or more appropriately ultimately the firmware includes
Collection of firmware processes
Operating system
Device drivers
Flow control
- Co-routines
- Function / procedure calls
- Concurrent operations
Error / exception management

In addition we have
The data and control busses interconnecting these
The interface between the hardware and software
Schedule of all operations

Specific implementation of these black boxes
Defined during the synthesis portion of the co-development
Basic implementation takes on following form
Constraints

Mapping to physical implementation
Encounters number of constraints

Such constraints and evaluation criteria
Similar to those we encountered when
Proposing and evaluating partitions

Of particular concern
• Ability to meet specification requirements
  At the end of the day
  One must satisfy requirements specification

• Ability to meet time constraints
  Time performance of constituent algorithms
  Real-time deadlines

• Ability to meet communications costs
  Both in time
  Complexity of implemented solution

• Size constraints
  Holds for both software and hardware
  Hardware
    Ability to fit design within hardware components
  Software
    Memory loading

Today
Current state of the art imposes some restrictions
On hardware architecture
Arises from
Our successes in hardware arena
Current state of hw synthesis tools
  Far ahead of those in software domain
  Many of these perform very well
Synthesis from
  VHDL, Verilog, SystemC
  Rather straight forward

Desire to make problem computationally more tractable
As power of tools continues to increase
  Such simplifying methods will become less necessary
Thus to minimize cost of particular partition
   We have a contemporary bias towards using
   Off-the-shelf processors
   Easily synthesizable logics such as FPGAs
   In such an environment we have
   Libraries of proven components
   Today FPGA technology advanced to point
   Processor core can be dropped into middle of
   array
   Mapping from HDL to array logic
   Well understood
   None the less there remains much room for
   Research and improvement

   ASICs and full custom ICs are expensive in time and money
   Once again we have a bias towards ASICs
   That can be implemented from legacy libraries
   Although more expensive than FPGA
   Less costly than full custom implementation

Thus while certainly improvements can always be made
Interesting areas are in software synthesis and developing associated tools

Once again there have been many advances in this area
   Code generators and template based sw development
      No longer novelties

Herein we’ll focus can be on software component of the problem

**Software Synthesis**

   Translating sequence of behavioural statements into
   Standard implementation language such as C or C++
      Relatively straight forward task

More difficult is formulating and meeting
   Scheduling requirements and real time constraints
      For the application

   Typically application built on single processor
   Accompanied by collections of programmable logics
   Implication is that with single CPU in reality
      We only have single real thread of control
Consequently
Concurrent threads of execution which may work well in model
Must be flattened into linear or sequential execution order
Thus in simple case 4 threads executing concurrently in model
Only receive one fourth of CPU rather than all
Can increase CPU speed by four times
Often impractical
Extend time constraints

System Characterization
Systems we are considering are
Reactive
Real-time
During modeling and partitioning
Such systems naturally decompose into
Concurrent pieces of functionality
Mapping of model to software / firmware
Must respect such a decomposition
Significant elements of modeled concurrency that must be considered
When executing the mapping
Communication
Synchronization
Computational flow
Timing constraints

Scheduling
As is evident with migration to physical implementation
Scheduling of sw execution on hw can be difficult problem

Have already examined scheduling algorithms
In context of study of operating systems

There we saw
Scheduling algorithms fall into several categories
- Static – Schedule determined at compile time
- Quasi-static – Some schedule decisions are made at compile time while others are made at runtime
- Dynamic – Schedule determined at runtime by dedicated piece of software
  Such schedulers may rely on
  Pre-determined information to establish execution order
  Runtime information
Such schedulers may be
Non-preemptive
Pre-emptive
The same rules and objectives apply now

As with all other aspects of co-design process
  Must make trade-offs
  Reliability and predictability
    Now more important as we move towards final implementation
  Scheduling (of processes) flexibility
  Minimize scheduling overhead

For embedded applications with (hard) real-time deadlines
  Prefer static – non-preemptive implementation
  Can relax to quasi-static with limited pre-emption
  Certainly there are cases in which pre-emption necessary

Such an approach
  Moves towards reliability and predictability
  Certainly reduces (runtime) flexibility
  Minimizes scheduling overhead

Synthesis Methods
  Most approaches to synthesizing software component
  Decompose problem into set of cooperating tasks
  Schedule according to
    • Classical algorithms
    • Ad hoc techniques driven by knowledge of Problem and domain
      Limited special purpose systems
      Ideal for such an approach

Let’s look at several different approaches to this problem

*Synonymous Model*
  Approach due to Berry, Benveniste
  Concepts involved originate with synchronous languages
  Used for embedded development

As we’ve seen such languages include
  *Esterel*
    Semantics based upon reactive model
    Synchronous and parallel systems
    Compiles into FSMs
    That can then be executed
**Lustre**
Synchronous data flow language
Targeted towards reactive systems
Specifically control and monitoring systems

**Signal**
Semantics based upon model of
Multiple clocked flows of
Data and events

Synchronous model utilizes *synchronous* interpretation of time
Underlies several contemporary synthesis tools

Synchronous approach
Familiar approach in engineering
Divides continuous time into discrete instances
Utilizes two common models
Event driven
Data or input driven

Schematically models execute as follows

**Underlying structure of author’s approach**
Builds a mathematical framework in which
Model’s time synchronized to one or more clocks
System advances ‘synchronously’ accordingly
System exhibits concurrent deterministic behaviour

Authors propose ideal model of reactive systems such that
System outputs produced synchronously with inputs
Such outputs occur instantly
Intra system communications
Implements an instantaneous broadcast model
Signals visible and reacted upon
At time of emission
Implements global interleaving of input signals
Determines how (asynchronous) inputs managed and effects computed
In time

```
# Data Driven
do forever
  for each clock tick
    get inputs
    compute system outputs
    update internal state
  end for
end do

# Event Driven
do forever
  for each input event
    compute system outputs
    update internal state
  end for
end do
```
Execution
Design of sw system is event driven

Begins with Esterel model

Derives single FSM from collection of concurrent modules

Inter module communication becomes rather complex
   Generated software emulates
      Syntactically derived hardware implementation from
      Esterel program

Synchronous approach avoids using schedule
   With each sensed event
      Reaction by system to all events present is
         Computed and executed
   With such approach we have precise determination of performance
      In (idealized) time
      Real world component accommodated as appropriate
      For context

Most other approaches
   Map concurrent processes onto cooperating tasks
   Formulate a schedule for execution
      Subject to timing constraints given in specification
   One such approach uses the RM scheduler

Rate Monotonic Schedule Base
   Approach due to Cochran
   Scheduling based upon rate monotonic analysis (RMA) algorithm
      Under such a model
         Use static assignment of priorities with pre-emption
         Deadlines equal to invocation period
         Assume system overhead negligible
   Recall that given such hypotheses
      If given set of tasks can be scheduled by static priority
         algorithm
         Can be scheduled by RMA algorithm

RMA is extended to include
   Synchronization constraints
   High priority I/O via ISRs
   Context switching overhead
   Multiple processors
   Deadline management
Author accommodates multiple processors
Evaluate ability to schedule selected allocation of tasks
To processors
Provides feedback on potential
Bottlenecks and deadlocks

Rate Monotonic Extension
Approach due to Chou, Walkup, Boriello

Earlier work – Esterel and StateCharts
Used idealized timing model
As we saw with synchronous model earlier
Assumes simple computations take zero time
Computations that violated the assumption
Modeled as external signals to system
Time constraints on such signals prohibited

Authors extend such a model by adding timing constraints

Method accommodates both
Fine and coarse grained timing constraints
Specification is Verilog based
Uses such constructs to provide structured concurrency
Augmented by watchdog-style preemption
Similar to UML guard or watchdog timer
Associated action invoked with condition met

Timing constraints specified using *modes*
Mode specifies scope or context in which
Specified set of timing constraints must be met
Until one of the watchdog triggers
Either disables or forces an exit from the mode
When such a transition initiated
Each concurrent branch target to be disabled
Allowed to run until safe exit point reached

Similar to StateCharts
Modes express different states for the operation
Initialization
Normal operation
Error recovery

Constraints on min and max inter event separation
Can be defined either
Within mode or
Set of events in several modes
Scheduling performed within a mode

1. Identifying cyclic order of operations
   That preserves I/O rates and timing constraints

2. Each mode transformed into acyclic partial ordering
   Done by unrolling
   Splitting
   If multiple parallel loops

3. Partial order linearized based upon longest path algorithm
   Checked for feasibility
   Start times assigned to all operations

Others

Numerous other synthesis and software scheduling algorithms
COSYMA and Vulcan represent two of these

COSYMA
Cosynthesis for embedded microarchitectures
Following hw/sw partitioning
   C code generation and communication synthesis
   HDL code generation and communication synthesis
Software synthesis
   Uses c compiler
Hardware synthesis
   Braunschweig Synthesis System and Synopsis Design
   Compiler

Vulcan
System behaviour specified in HardwareC
   Another HDL with C like syntax
   Supports both timing and resource constraints
   Including delay operations
HDL implementation
   Compiled into graph model based on data flow graphs
Following constraint analysis and satisfaction phase
   System partitioned into
   Program graph
      Code synthesized into C program
Interface portion
ASIC graph
   ASIC and interface synthesized to structural level
   Targeted towards ASIC implementation
Co-Verification

It’s assumed that verification has followed each iteration
Through the partition-simulate-synthesize cycle
It’s further assumed that we have been doing
All appropriate analysis of our design throughout
We are now in the latter stages of the design
We have completed a ‘final’ synthesis of our design
Must confirm that it satisfies our original system specification

Hardware-software co-verification

Gives us a means for testing and stressing our design
Long before we have a physical prototype available
We endeavour to reduce risk by identifying problems
Earlier rather than later

As today’s designs become increasingly integrated
Visibility into the internals of the devices
Becomes more and more limited
Unfortunately traditional tools such as scopes and logic analyzers
Rely on being able to connect to and monitor important signals
System busses
Other data and control signals
With no places to connect
They loose effectiveness

JTAG port
Developed with intention of giving some access
Difficulty
Must be designed in early in process
Consumes valuable
Internal real estate
I/O pins

Driven by
The high costs of errors in silicon
Increasing demands to reduce time to market
Our desire is to verify that our design is correct
Prior to first silicon
Through co-simulation and co-verification we strive to
Verify the hardware – software interface
Accelerate the firmware debug process
Exercise
Boot code
Hardware and software diagnostics
Device drivers
Board support packages
Some application code
It’s recognized that a good model improves probability of solid design
However models not perfect
To model every hardware and hardware – software interaction
Quickly becomes impractical as we move towards
Smaller and higher performance designs
Circuit that may perform superbly on paper or distributed prototype
May suffer severe noise, crosstalk, and EMI contamination
As geometries reduced in today’s designs

Tools
To enable us to effectively co-simulate and co-verify our designs
Must have good tools at all levels of process

Today there are a great variety of commercially available tools
To help embeded developer
We will not discuss any of these
Information can be found with the various vendors

Let’s look at capabilities such tools have or should have

Most such tools share a common set of basic capabilities
- Control of hardware and software simulation
  At all levels of development process
  Essential are
  Support for initialization
  Breakpoint synchronization
  Set in either hw or sw domain
  Encountering either should halt system
- Ability to pass variables between hw and sw environments
- Support for various levels of abstraction
  Includes high level behavioural models
  HDLs
  C/C++ models
  Detailed component structural level and hardware emulators
  Can range from
  Complete software abstraction
  Hardware stubbed out and interface modeled
  System in silicon

Levels of Abstraction
Let’s examine necessary abstraction levels
Supported by many tools
In a bit more detail

- Software only
  - Complete abstraction
    - Hardware stubbed out and interface modeled
    - Application interfaces with
      - Stubbed out or modeled hardware through device driver
    - Enables functionality of software to be tested
    - Excludes
      - Detailed and constrained timing
      - Asynchronous types of events

- HDL simulation of hardware
  - Can include functional model of system bus
  - Such a model can
    - Execute atomic bus cycles
    - Reads and writes to supporting arrayed logics
    - With correct bus timings
    - Modeling and handling asynchronous events
      - Such as interrupts
  - Does not reflect internals of system CPU
    - Missing registers
    - Internal peripherals
      - Timers
      - Signal converters
  - Aids in debugging peripheral module functionality
    - From bus side
    - Developing device drivers

- Instruction Set Simulation (ISS) – Modeling the CPU
  - Model of system CPU or digital signal processor added
  - Usually done as C/C++ behavioural level model
  - May run as
    - Separate process under Unix / Linux
      - Such a process is separate from the HDL simulation
    - Sockets or similar inter-process communication scheme
      - Used to exchange information
    - Approach is not cycle accurate
    - HDL wrapper surrounding the ISS model
      - Approach can be cycle accurate

- CPU Emulator
  - CPU/DSP model extricated from system
    - Replaced by ICE module(s) for actual processor(s)
  - Accurately replicates CPU bus cycle timing
    - Thereby provides robust platform for sw development
- Complete HDL Model
  Gives full HDL implementation of complete system
  Can be executing at RTL or discrete device level
  Tends to be more useful for developing hw side
  Rather than sw side
  Useful for verifying
  Detailed and critical timing
  Hw / sw interactions
  Boot process
  Memory interfaces and subsystems

- Hardware Emulation
  Hardware modeled through some form of arrayed logic
  Specifically this is an emulation
  Execution speed approaches that of full ASIC
  Contemporary arrayed logics
  Can support inclusion of CPU cores
  Permits both hw and sw debug

Summary
  Have looked at co-design process in some detail
  Noted that such process
  Playing larger and larger role in design and development
  Contemporary embedded applications
  Goal of methodology
  Permit hw and sw engineers to solve design problems
  Early in process so as to enable success
  On first silicon for SOC
  On system integration for discrete systems
  Tools are being developed at increasingly fast pace
  Much work remains to be done