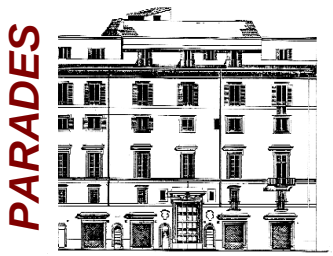
An Overview of (Electronic) System Level Design: beyond hardware-software co-design

Alberto Ferrari
Deputy Director
PARADES GEIE
Alberto.Ferrari@parades.rm.cnr.it



Outline

- Embedded System Applications
- Platform Based Design Methodology
- Electronic System Level Design
  - Functions: MoC, Languages
  - Architectures: Network, Node, SoC
- Metropolis
- Conclusions

ESL Design

- Designing embedded systems requires addressing concurrently different engineering domains, e.g., mechanics, sensors, actuators, analog/digital electronic hardware, and software.
- In this tutorial, we focus on Electronic System Level Design (ESLD), traditionally considered as the design step that pertains to the electronic part (hardware and software) of an embedded system.
- ESL design starts from *system* specifications and ends with a system implementation that requires the definition and/or selection of hardware, software and communication components

Outline

- Embedded System Applications
- Copying with heterogeneity
- Methodology: platform based design
- Electronic System Level Design
  - Functions: MoC, Languages
  - Architectures: Network, Node, SoC
- Metropolis
- Conclusions

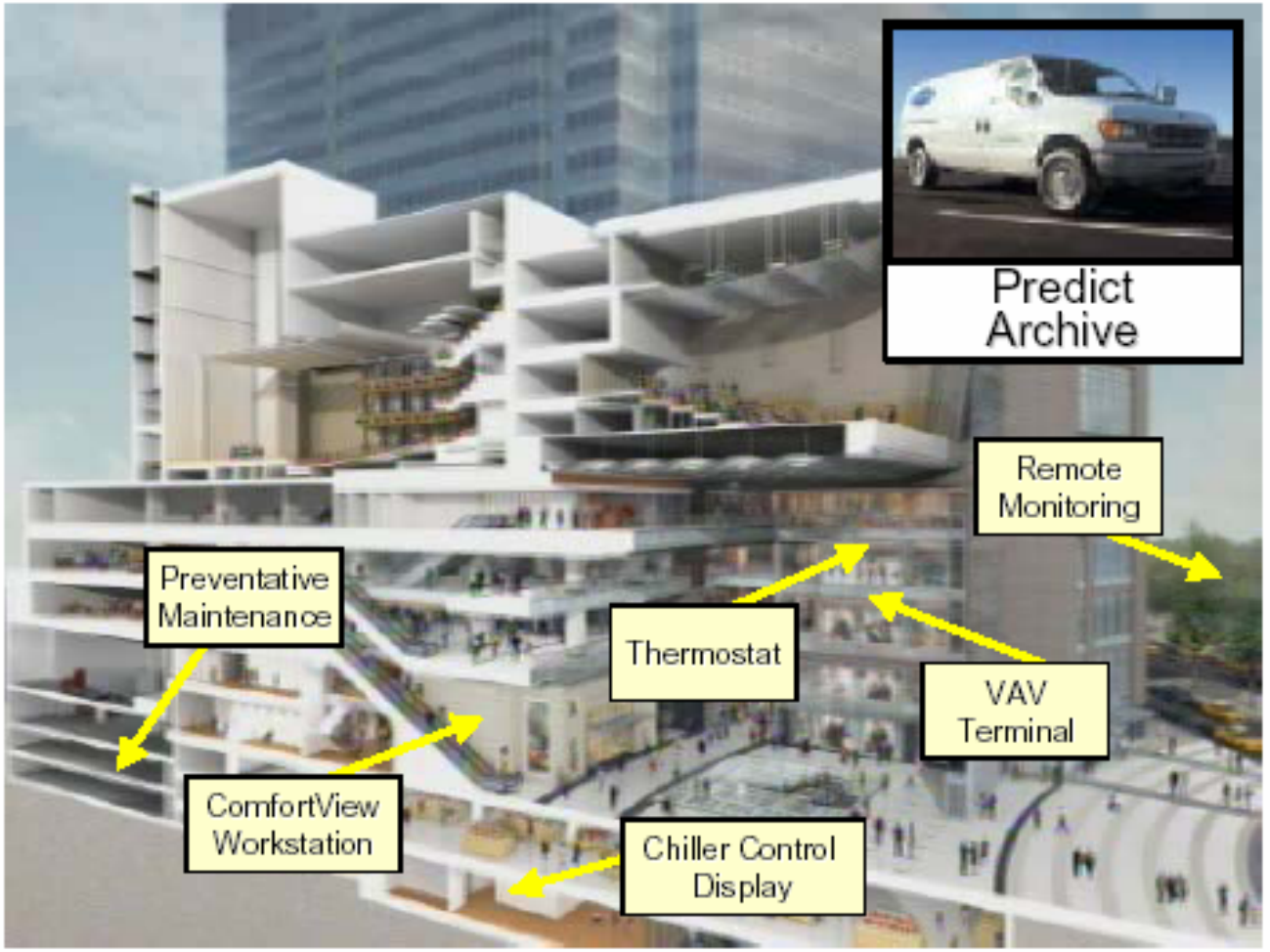
Embedded Systems

- Computational
  - but not first-and-foremost a computer
- Integral with physical processes
  - sensors, actuators
- Reactive
  - at the speed of the environment
- Heterogeneous
  - hardware/software, mixed architectures
- Networked
  - shared, adaptive

Source: Edward A. Lee

FUNCTION OF CONTROLS

Typical commercial HVAC application



Configure

Sense

Actuate

Regulate

Display

Trend

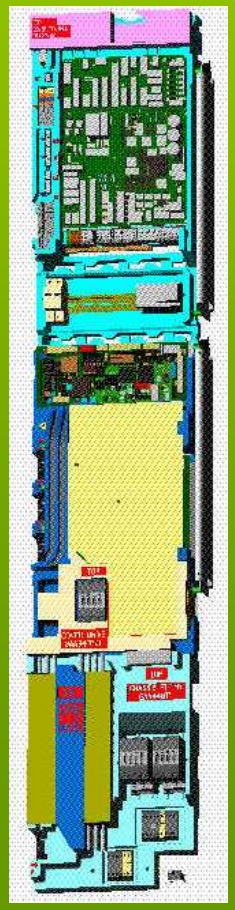
Diagnose

Predict

Archive

OTIS Elevators

1. EN: GeN2-Cx



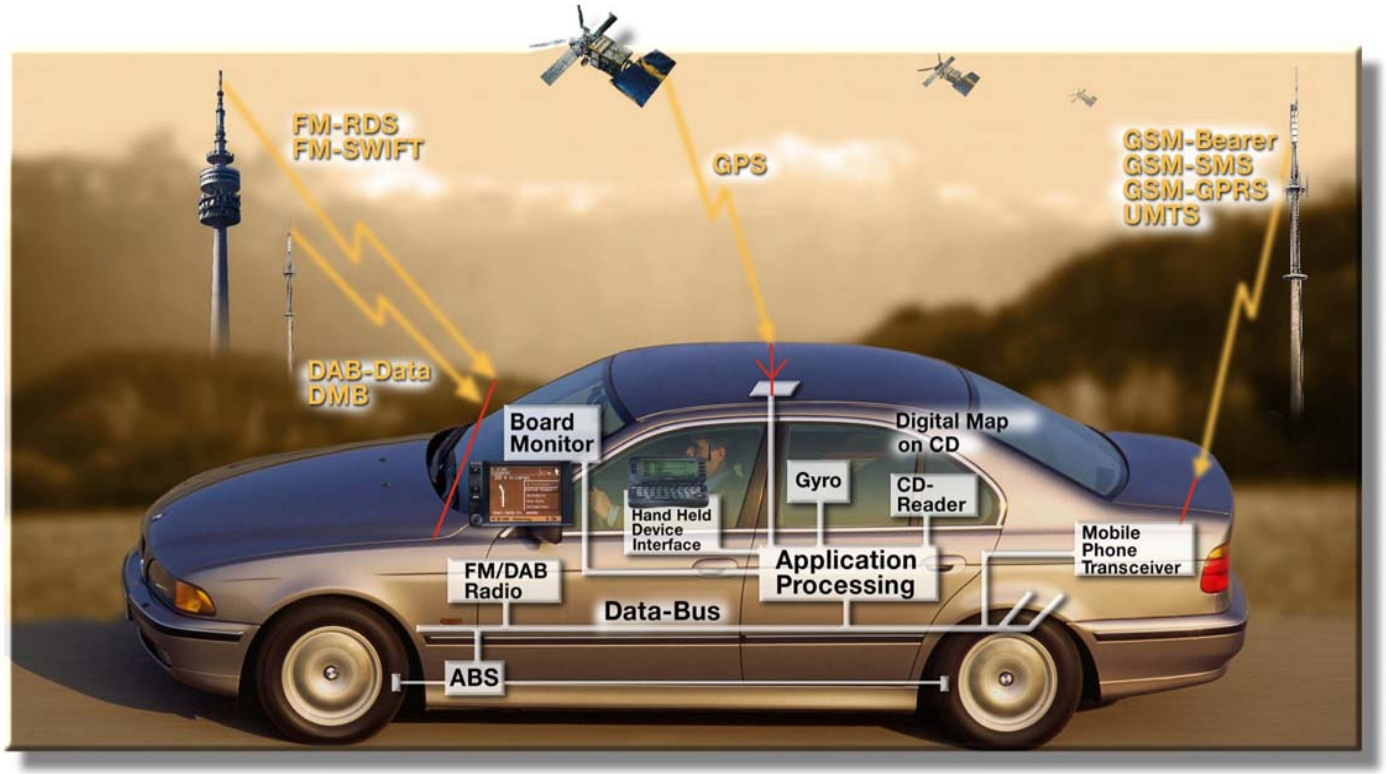
2. ANSI:
Gen2/GEM

3. JIS:
GeN2-JIS

$4 billion development effort
40-50% system integration & validation cost

Electronics and the Car

- More than 30% of the cost of a car is now in Electronics
- 90% of all innovations will be based on electronic systems

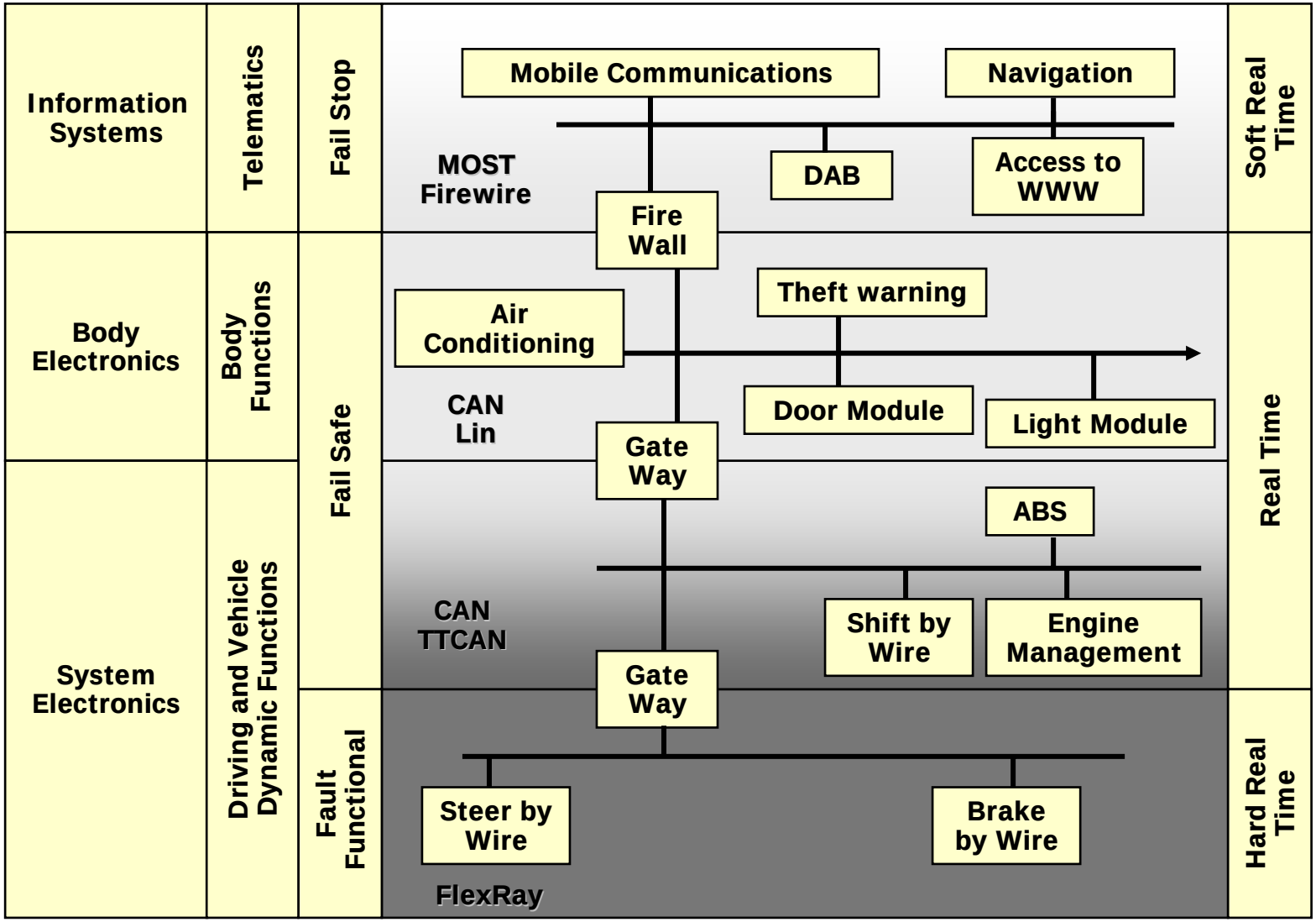


Complexity, Quality, & Time To Market today

		PWT UNIT 	BODY GATEWAY 	INSTRUMENT CLUSTER 	TELEMATIC UNIT 
Memory	256 Kb	128 Kb	184 Kb	8 Mb	
Lines Of Code	50.000	30.000	45.000	300.000	
Productivity	6 Lines/Day	10 Lines/Day	6 Lines/Day	10 Lines/Day*	
Residual Defect Rate @ End Of Dev	3000 Ppm	2500 ppm	2000ppm	1000 ppm	
Changing Rate	3 Years	2 Years	1 Year	< 1 Year	
Dev. Effort	40 Man-yr	12 Man-yr	30 Man-yr	200 Man-yr	
Validation Time	5 Months	1 Month	2 Months	2 Months	
Time To Market	24 Months	18 Months	12 Months	< 12 Months	

*** C++ CODE**

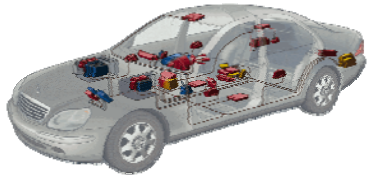
Distributed Car Systems Architectures



Design

- From an idea…
- … build something that performs a certain function
- Never done directly:
  - some aspects are not considered at the beginning of the development:
    - Node and Network
    - Processes and Processors
    - SoC Software and Hardware
  - the designer wants to explore different possible implementations in order to maximize (or minimize) a cost function
- The solution is a trade-off among:
  - Mechanical partition
  - Hardware partition: analog and digital
  - Software partition: low, middle and application level

(Automotive) V-Models: Car level

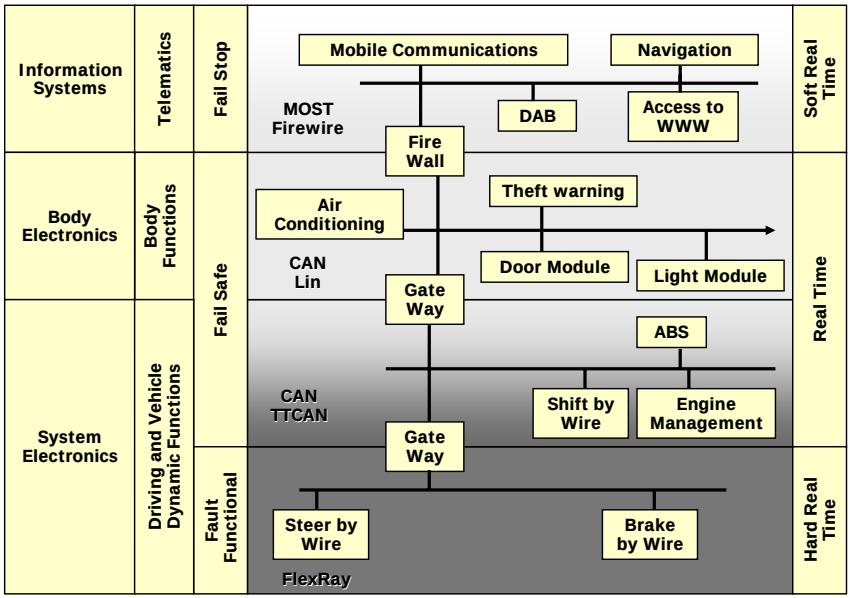


Development of Distributed System

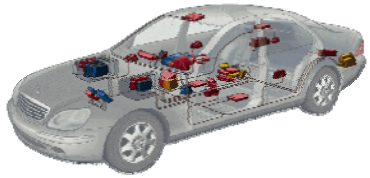
Distributed System Sign-Off!

Sub-System(s) Integration, Test, and Validation

- What:
  - Functionality
- How:
  - Architecture
- Trading (ES):
  - Computation (hw/sw)
  - Communication (hw/sw)
    - Time trigger/Event trigger
- Abstractions ?
- Cost evaluation ?



(Automotive) V-Models: Subsystem Level



Development of Distributed System

Development of Sub-System

Development of Mechanical Part (s)

ECU Development

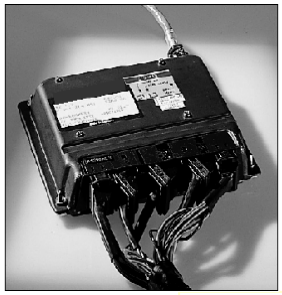
Distributed System Sign-Off!

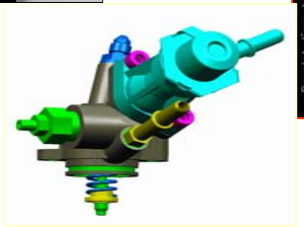
Sub-System(s) Integration, Test, and Validation

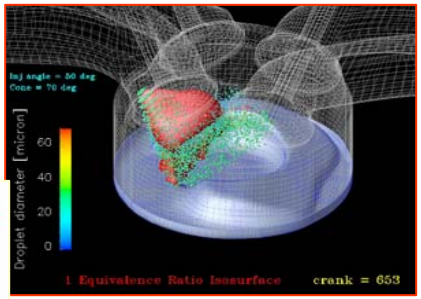
Sub-System Sign-Off!

ECU/ Sens./Actrs./Mech. Part(s) Integration, Calibration, and Test

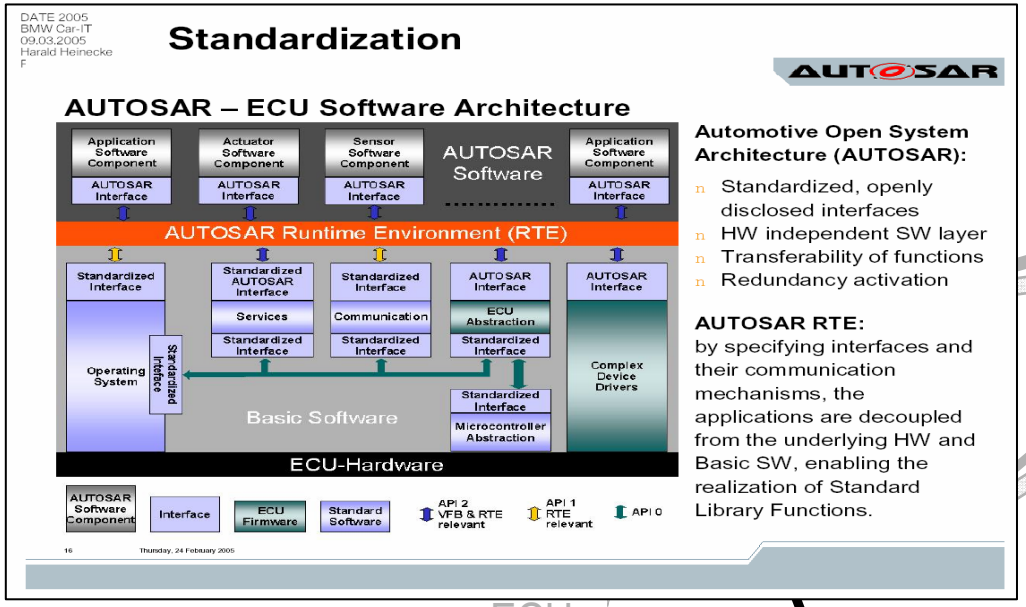
- What: Functionality
- How: Architecture
- Trading (ES):
  - Algorithm complexity (hw/sw)
  - Sensors/Actuators
- Abstractions ?
- Cost evaluation ?

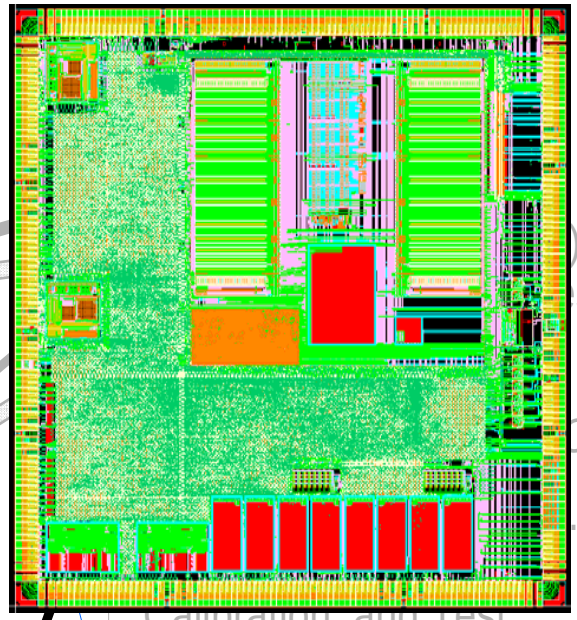






(Automotive) V-Models: ECU level (Hw/Sw)





- What: Functionality
- How: Architecture
- Trade (ES):
  - Hardware
  - Software
- Abstractions ?
- Cost evaluation ?

ECU Development

ECU SW Development

ECU HW Development

ECU SW Implementation

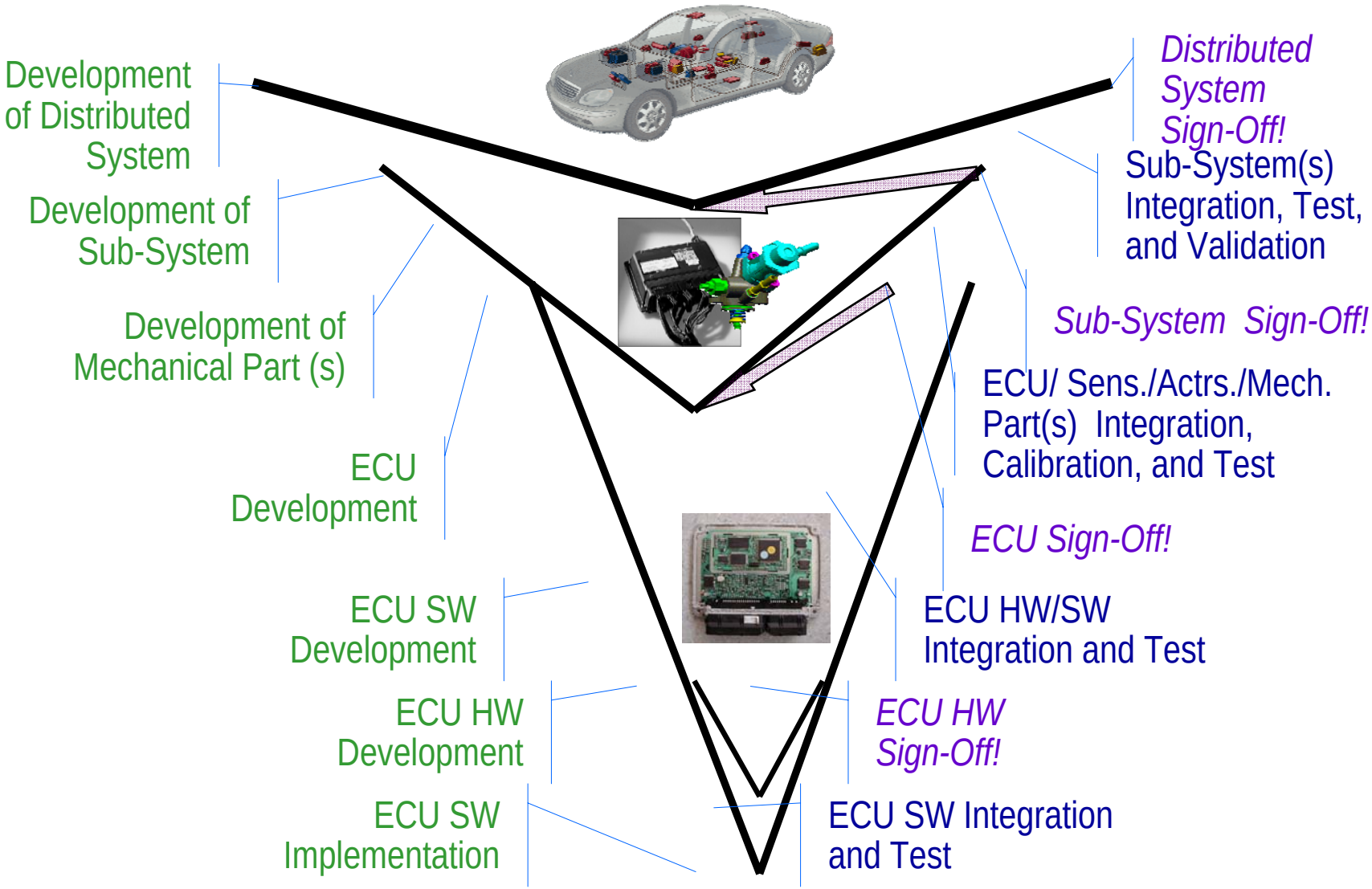
ECU Sign-Off!

ECU HW/SW Integration and Test

ECU HW Sign-Off!

ECU SW Integration and Test

(Automotive) V-Models



Common Situation in Industry

- Different hardware devices and architectures
- Increased complexity
- Non-standard tools and design processes
- Redundant development efforts
- Increased R&D and sustaining costs
- Lack of standardization results in greater quality risks
- Customer confusion

How to…

- How to propagate functionality from top to bottom
- How to evaluate the trade offs
- How to cope with:
  - Design Time
  - Design Reuse
  - Design Heterogeneity
- How to abstract with models that can be used to reason about the properties

Heterogeneity in Electronic Design

- Heterogeneity in:
  - Specification:
    - formal/semi-formal/natural language
    - MoC
    - Language
  - Analysis
  - Synthesis:
    - Manual/automatic/semi-automatic
  - Verification
  - Methodology
  - Design Process

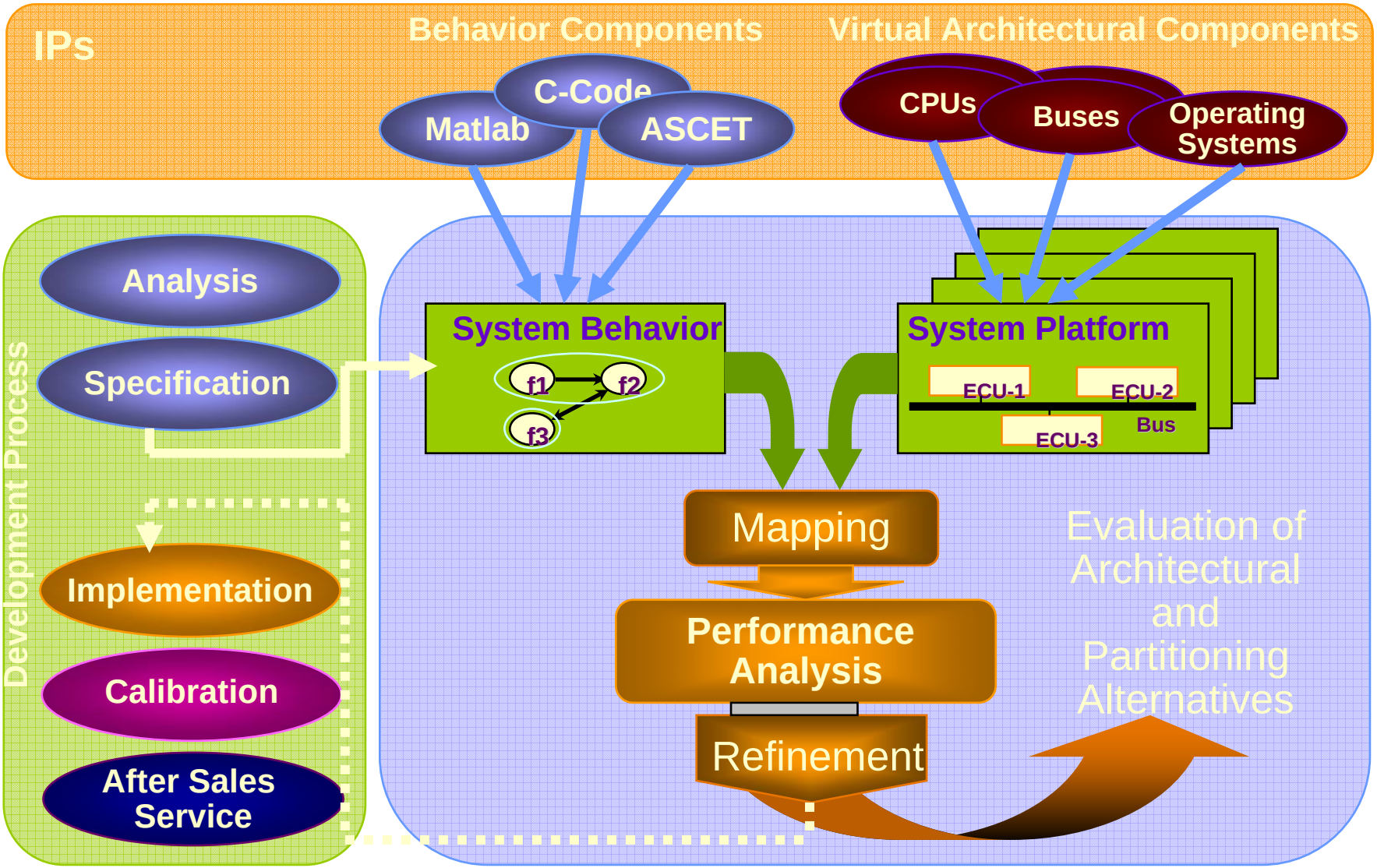
Outline

- Embedded System Applications
- Platform based design methodology
- Electronic System Level Design
  - Functions: MoC, Languages
  - Architectures: Network, Node, SoC
- Metropolis
- Conclusions

Separation of concerns

- Computation versus Communication
- Function versus Architecture
- Function versus Time

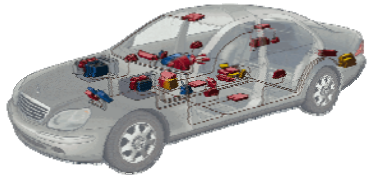
Separation of Concerns (1990 Vintage!)



Principles of Platform methodology: Meet-in-the-Middle

- Top-Down:
  - Define a set of abstraction layers
  - From specifications at a given level, select a solution (controls, components) in terms of components (Platforms) of the following layer and propagate constraints
- Bottom-Up:
  - Platform components (e.g., micro-controller, RTOS, communication primitives) at a given level are abstracted to a higher level by their functionality and a set of parameters that help guiding the solution selection process. The selection process is equivalent to a covering problem if a common semantic domain is used.

Platform Models for Model Based Development



Development of Distributed System

Distributed System Requirements

Distributed System Partitioning

Sub-Systems *Model Based* Development

Sub-Systems (s) Requirements

Network Protocol Requirements

Sub-System(s) Sign-Off!

Platform Abstraction

Distributed System Sign-Off!

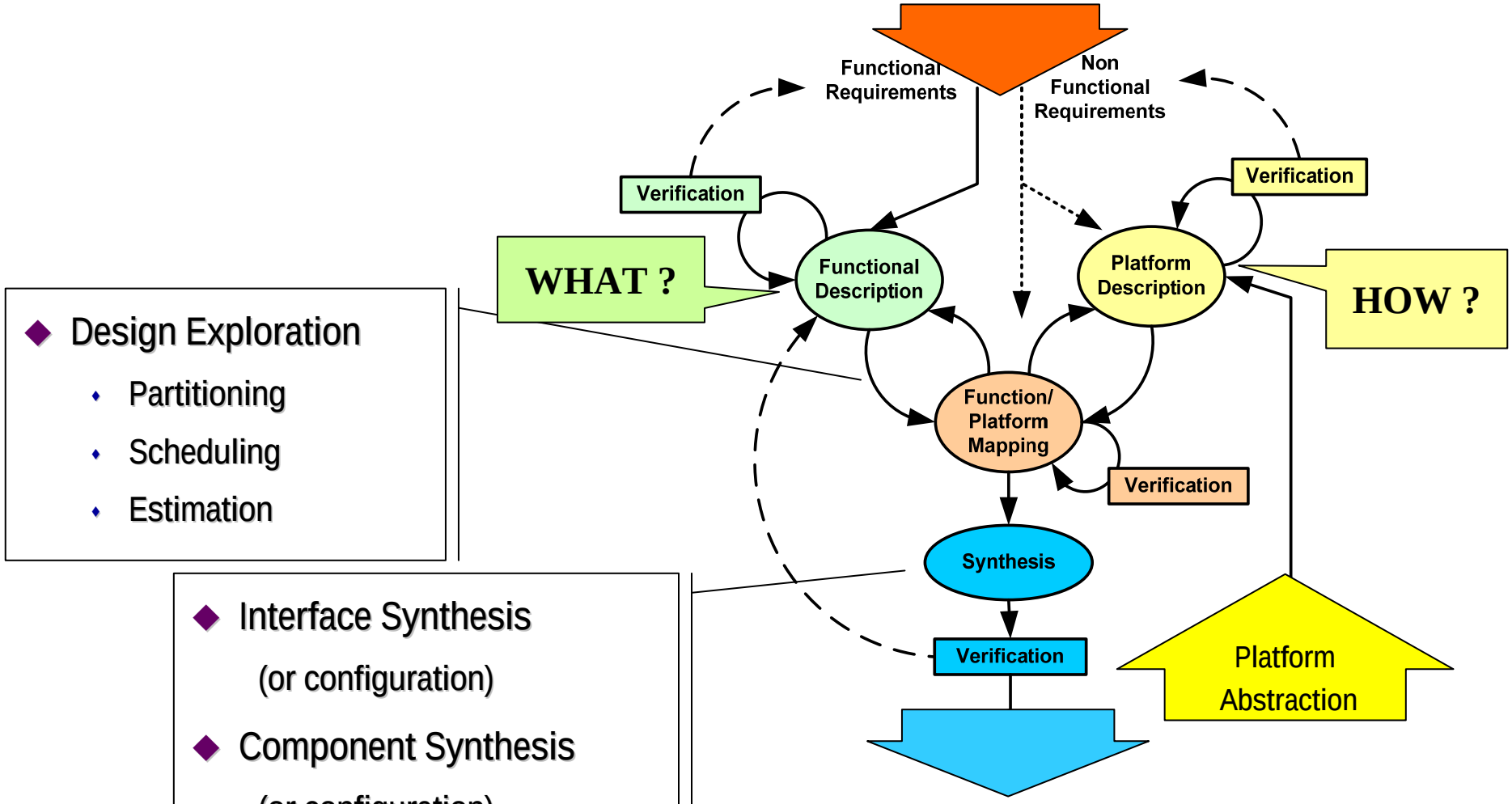
Sub-System(s) Integration, Test, and Validation

Virtual Integration of Sub-System(s) w/ Network Protocol, Test, and Validation

Sub-System(s) Implementation Models Sign-Off!

Network Communication Protocol Sign-Off!

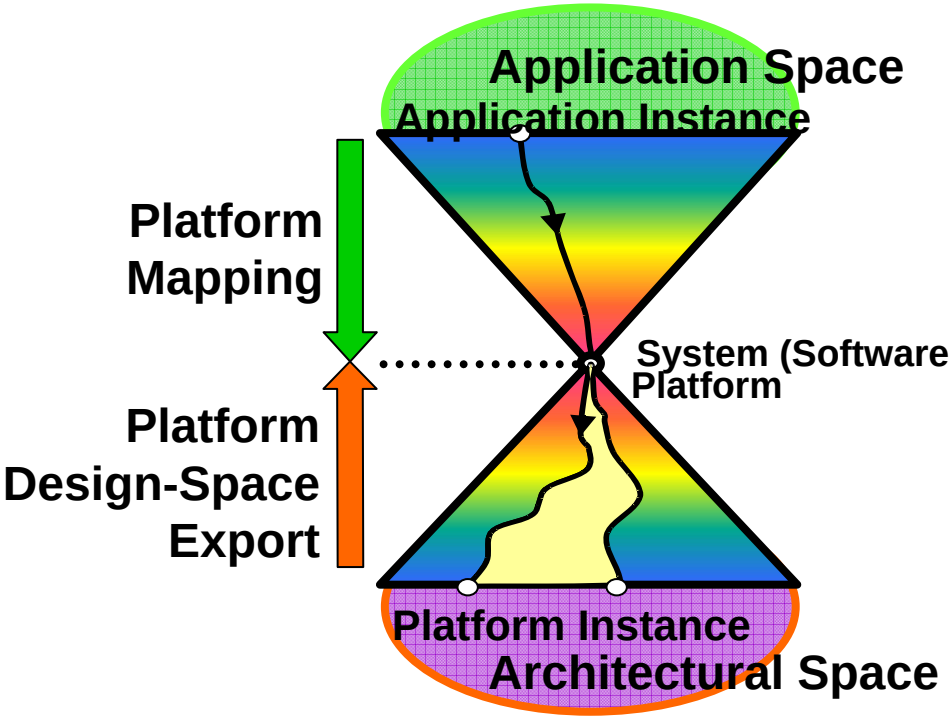
Meet-in-the-middle

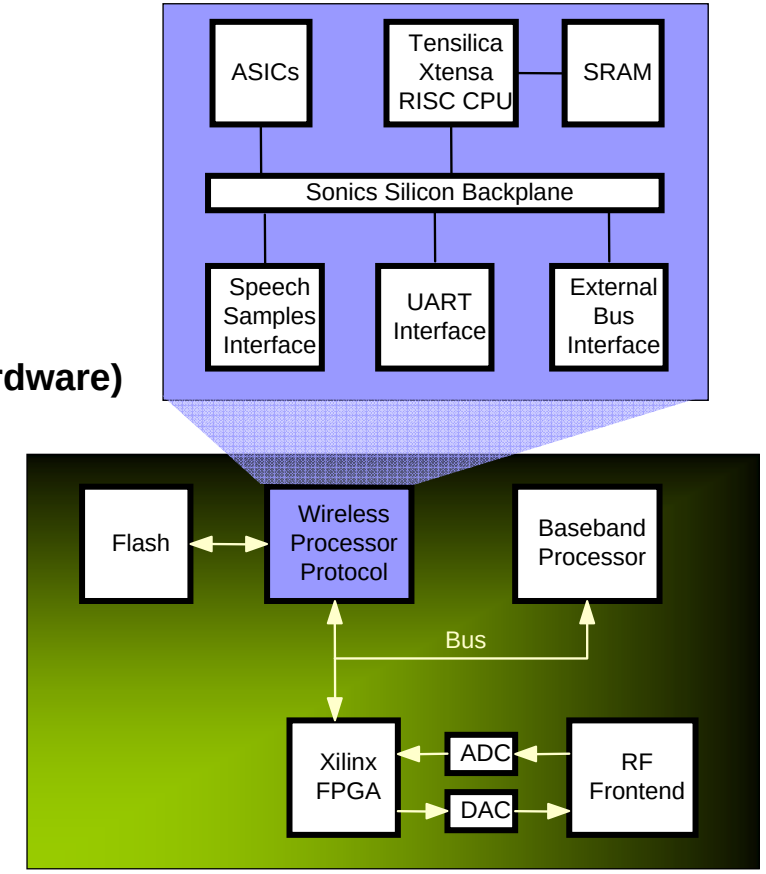


Aspects of the Hw/Sw Design Problem

- Specification of the system (top-down)
- Architecture export (bottom-up)
  - Abstraction of processor, of communication infrastructure, interface between hardware and software, etc.
- Partitioning
  - Partitioning objectives
    - Minimize network load, latency, jitter,
    - Maximize speedup, extensibility, flexibility
    - Minimize size, cost, etc.
  - Partitioning strategies
    - partitioning by hand
    - automated partitioning using various techniques, etc.
- Scheduling
  - Computation
  - Communication
- Different levels:
  - Transaction/Packet scheduling in communication
  - Process scheduling in operating systems
  - Instruction scheduling in compilers
  - Operation scheduling in hardware
- Modeling the partitioned system during the design process

Platform-based Design

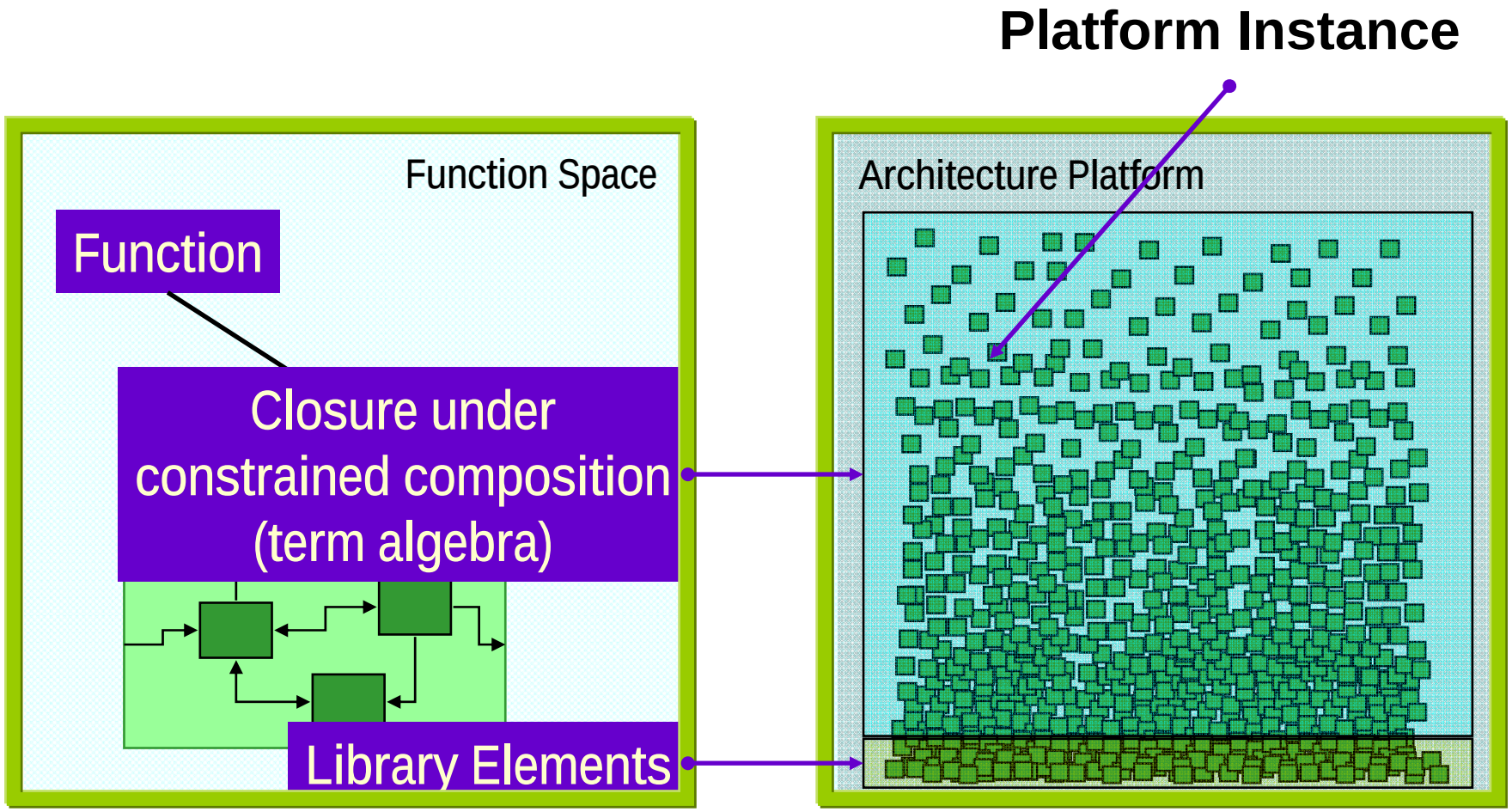




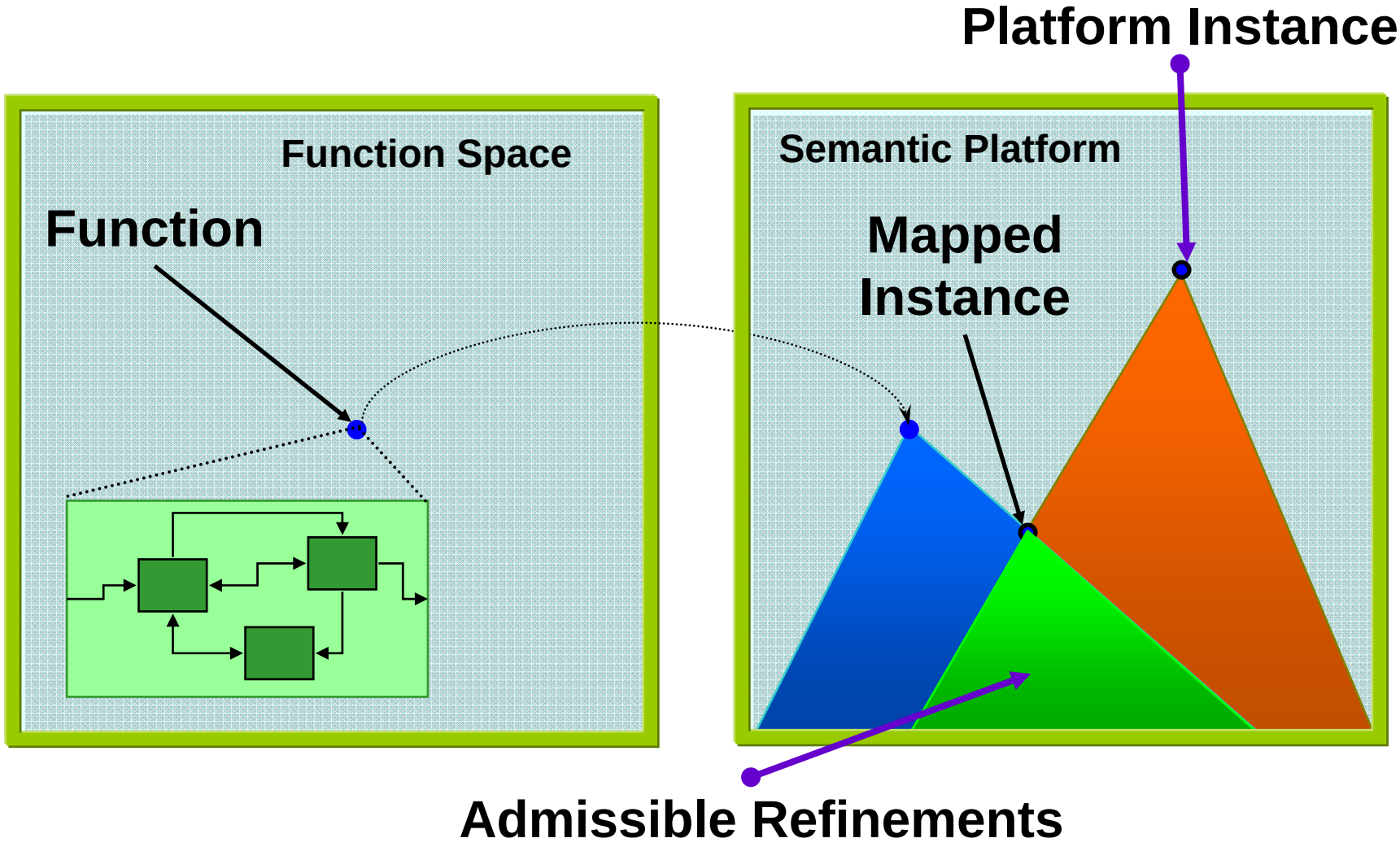
Intercom Platform (BWRC, 2001)

- Platform: library of resources defining an abstraction layer
  - hide unnecessary details
  - expose only relevant parameters for the next step

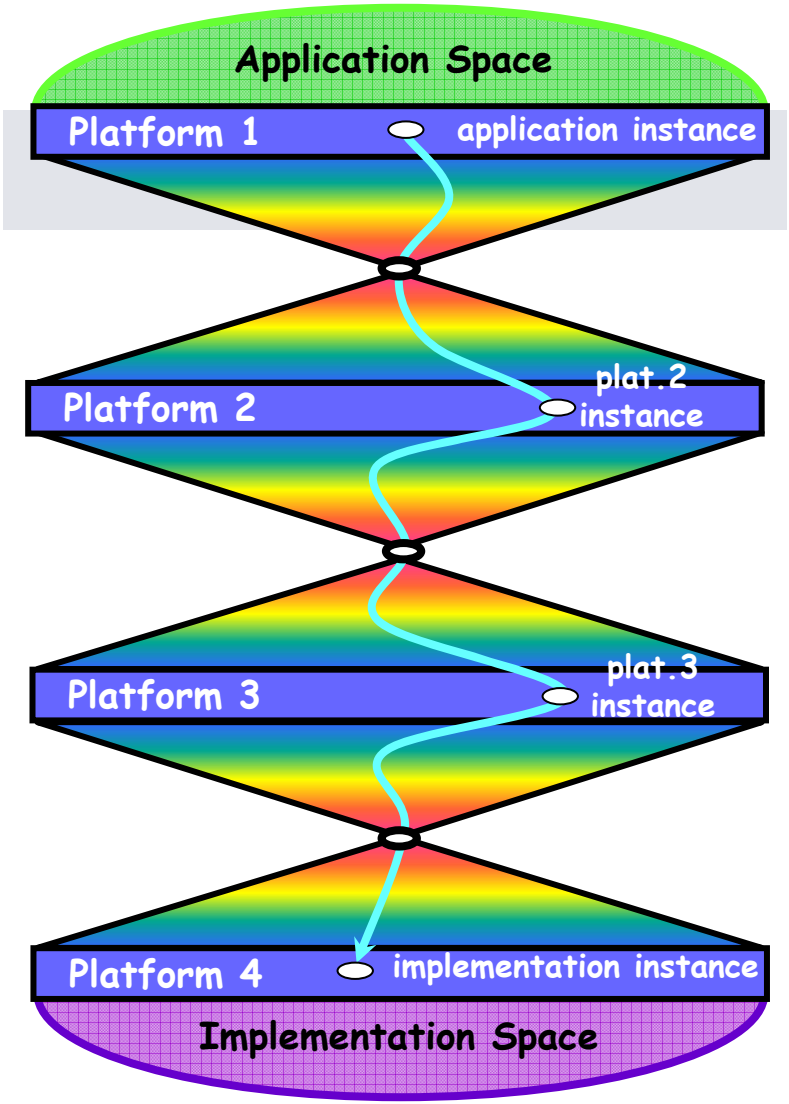
Formal Mechanism

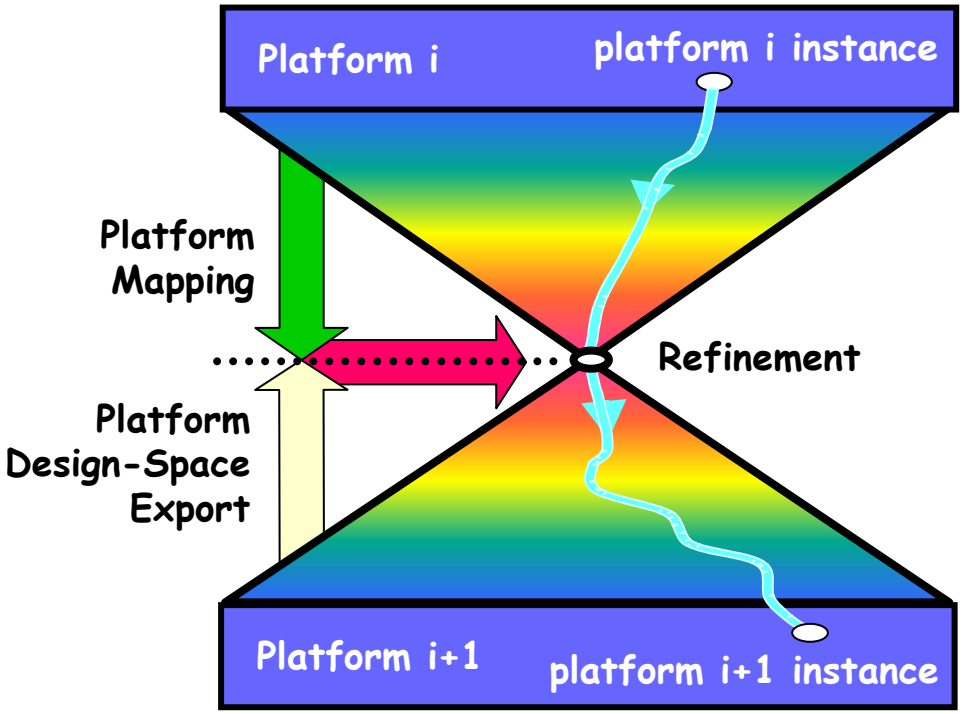


Mapping

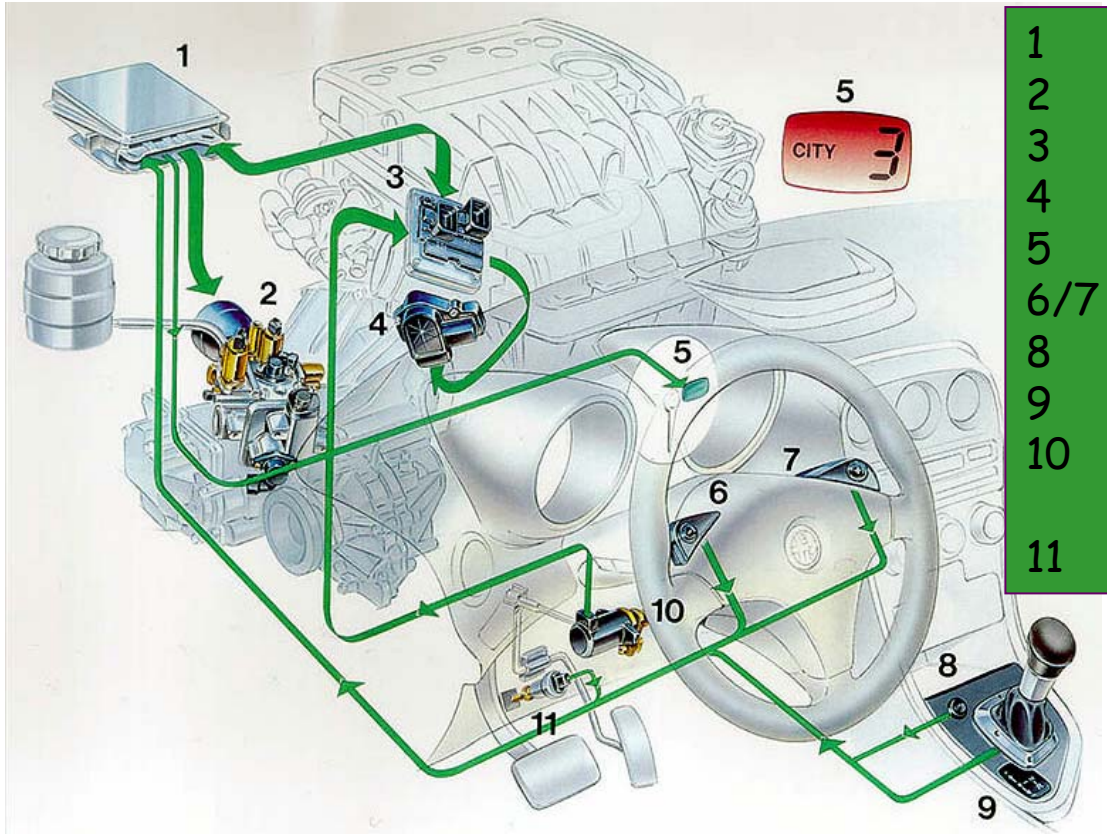

Platform Instance
Function Space
Function
Semantic Platform
Mapped
Instance
Admissible Refinements

Platform stack & design refinements





Automotive Supply Chain: Tier 1 Subsystem Providers



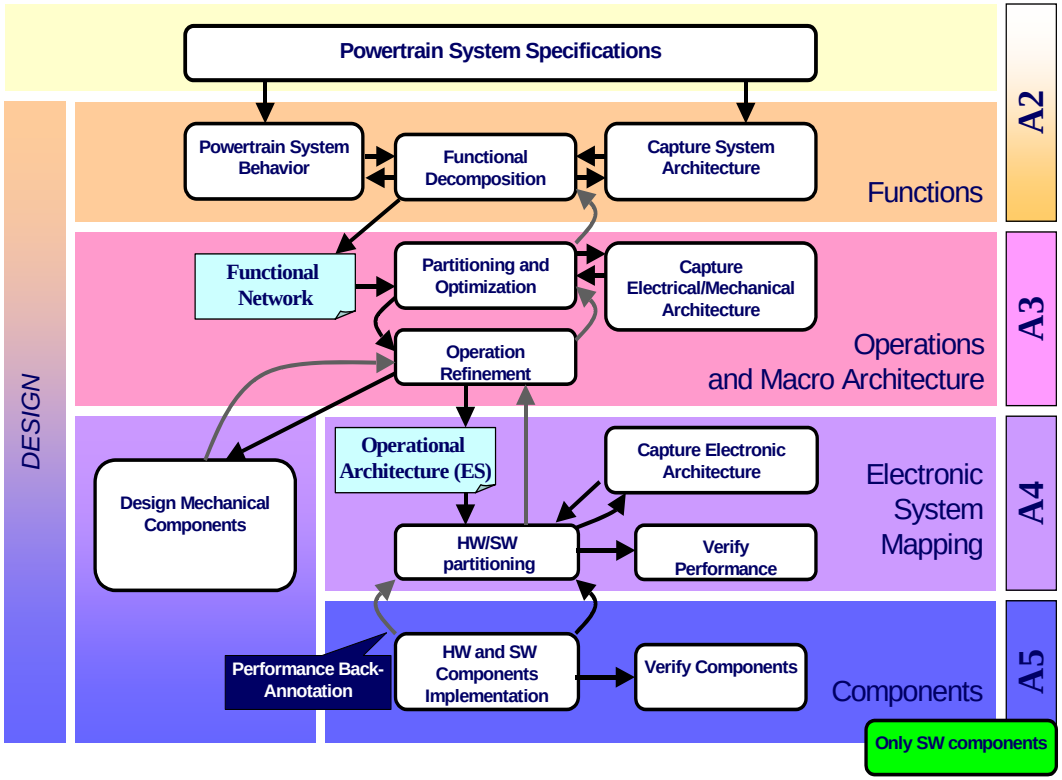
1	Transmission ECU
2	Actuation group
3	Engine ECU
4	DBW
5	Active shift display
6/7	Up/Down buttons
8	City mode button
9	Up/Down lever
10	Accelerator pedal position sensor
11	Brake switch

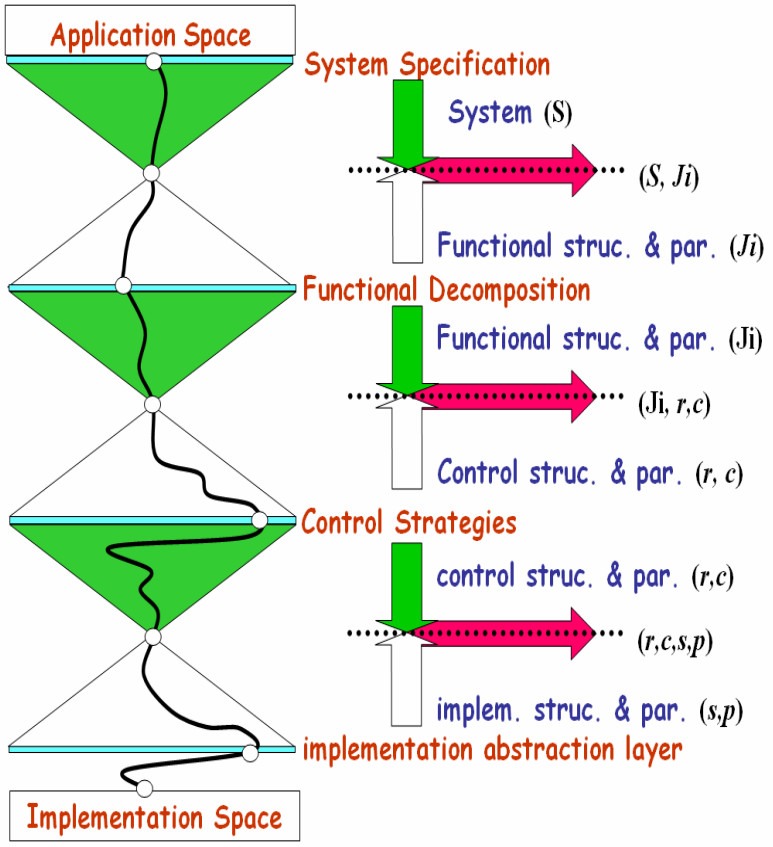
BOSCH

MAGNETI MARELLI

- Subsystem Partitioning
- Subsystem Integration
- Software Design: Control Algorithms, Data Processing
- Physical Implementation and Production

Magneti Marelli Power-train Platform Stack





Outline

- Embedded System Applications
- Platform based design methodology
- Electronic System Level Design
  - Functions: MoC, Languages
  - Architectures: Network, Node, SoC
- Metropolis
- Conclusions

Design Formalization

- Model of a design with precise unambiguous semantics:
- Implicit or explicit relations: inputs, outputs and (possibly) state variables
- Properties
- "Cost" functions
- Constraints

Formalization of Design + Environment =
closed system of equations and inequalities over some algebra.

What: Functional Design

- A rigorous design of functions requires a mathematical framework
  - *The functional description must be an invariant of the design*
  - The mathematical model should be expressive enough to capture easily the functions
    - The different nature of functions might be better captured by heterogeneous model of computations (e.g. finite state machine, data flows)
- The functional design requires the abstraction of
  - Time (i.e. un-timed model)
    - Time appears only in constraints that involve interactions with the environment
  - Data type (i.e. infinite precision)
- Any implementation MUST be a refinement of this abstraction (i.e. functionality is "guaranteed"):
  - E.g. Un-timed -> logic time -> time
  - E.g. Infinite precision -> float -> fixed point

Models of Computation

- FSMs
- Discrete Event Systems
- CFSMs
- Data Flow Models
- Petri Nets
- The Tagged Signal Model
- Synchronous Languages and De-synchronization
- Heterogeneous Composition: Hybrid Systems and Languages
- Interface Synthesis and Verification
- Trace Algebra, Trace Structure Algebra and Agent Algebra

Definition: A mathematical description that has a syntax and rules for computation of the behavior described by the syntax (semantics). Used to specify the semantics of computation and concurrency.

Usefulness of a Model of Computation

- Expressiveness
- Generality
- Simplicity
- Compilability/ Synthesizability
- Verifiability

The Conclusion

One way to get all of these is to mix diverse, simple models of computation, while keeping compilation, synthesis, and verification separate for each MoC. To do that, we need to understand these MoCs relative to one another, and understand their interaction when combined in a single system design.

Reactive Real-time Systems

- Reactive Real-Time Systems
  - "React" to external environment
  - Maintain permanent interaction
  - Ideally never terminate
  - timing constraints (real-time)
- As opposed to
  - transformational systems
  - interactive systems

Models Of Computation for reactive systems

- We need to consider essential aspects of reactive systems:
  - time/synchronization
  - concurrency
  - heterogeneity
- Classify models based on:
  - how specify behavior
  - how specify communication
  - implementability
  - composability
  - availability of tools for validation and synthesis

Models Of Computation for reactive systems

- Main MOCs:
  - Communicating Finite State Machines
  - Dataflow Process Networks
  - Petri Nets
  - Discrete Event
  - (Abstract) Codesign Finite State Machines
  - Synchronous Reactive
  - Task Programming Model

Details

Details

- Main languages:
  - StateCharts
  - Esterel
  - Dataflow networks
  - Simulink
  - UML

Models Of Computation for reactive systems

- Main MOCs:
  - Communicating Finite State Machines
  - Dataflow Process Networks
  - Petri Nets
  - Discrete Event
  - Codesign Finite State Machines
  - Synchronous Reactive
  - Task Programming Model
- Main languages:
  - StateCharts
  - Esterel
  - Dataflow networks
  - Simulink
  - UML

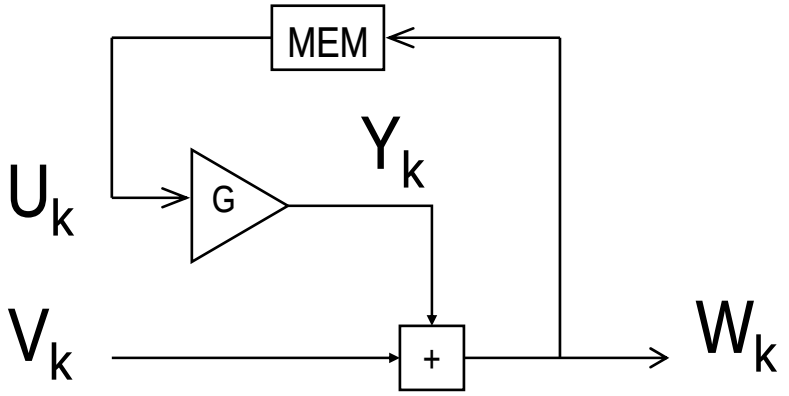
The Synchronous Programming Model

- Synchronous programming model* is dealing with concurrency as follows:
  - non overlapping computation and communication phases taking zero-time and triggered by a global tick
- Widely used and supported by several tools: Simulink, SCADE, ESTEREL …
- Strong constraints on the final implementation to preserve the separation between computation and communication phases

A. Benveniste and G. Berry: The synchronous approach to reactive and real-time systems, Proc IEEE, 1991

The Synchronous Reactive (SR) MoC (*)

- Discrete model of time (global set of totally ordered “time ticks”)
- Blocks execute atomically at every time tick
- Blocks are computed in causal order (writer before reader)
- State variables (MEMs) are used to break combinatorial paths
- Combinatorial loops have fixed-point semantics

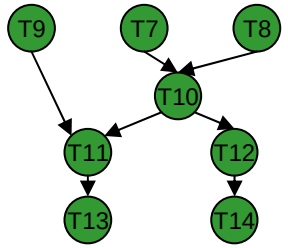


$$U_k = W_{k-1}$$
$$Y_k = G*U_k = G*W_{k-1}$$
$$W_k = V_k+Y_k = V_k+G*W_{k-1}$$

(*) S. A. Edwards and E. A. Lee, *“The semantics and execution of a synchronous block-diagram language”*, *Science of Computer Programming*, 48(1):21–42, jul 2003.

The Task Programming Model

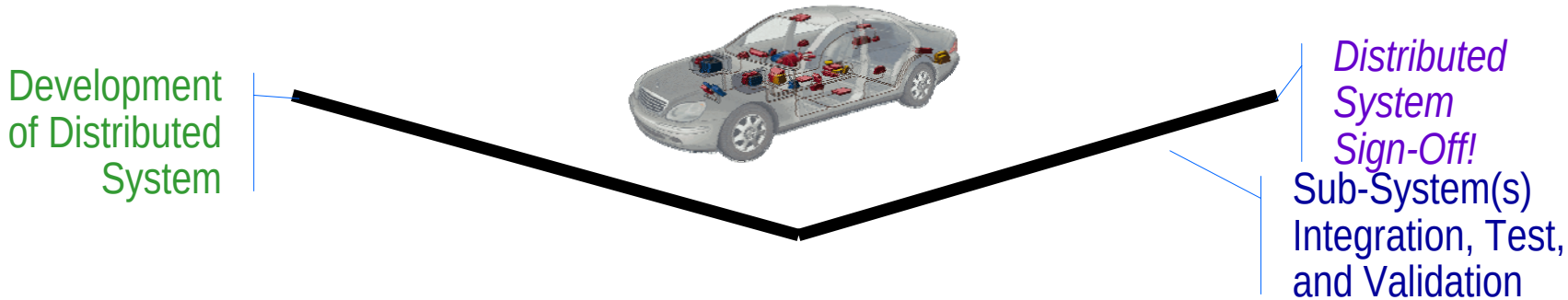
- The Task Programming Model (TPM)
  - A task is a logically grouped sequence of operations
  - Each task is released for execution on an event/time reference
  - Task execution can be deferred as long as it meets its deadline
  - Task scheduling is priority-based possibly with preemption
    - Priorities can be static or dynamic
  - Communication between tasks occurs:
    - Locally: via shared variables
    - Globally: via communication network
  - Output values depend on scheduling
- Represented by Task Graphs

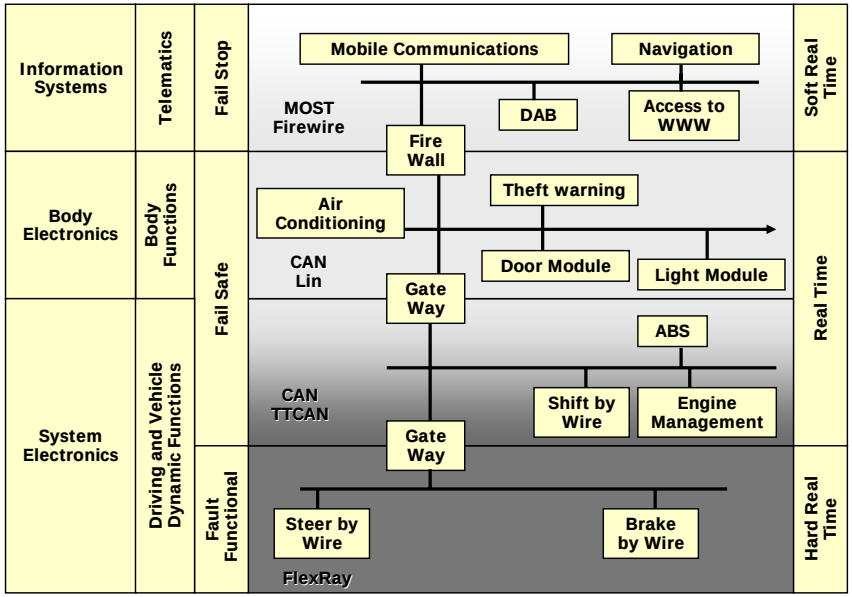


Outline

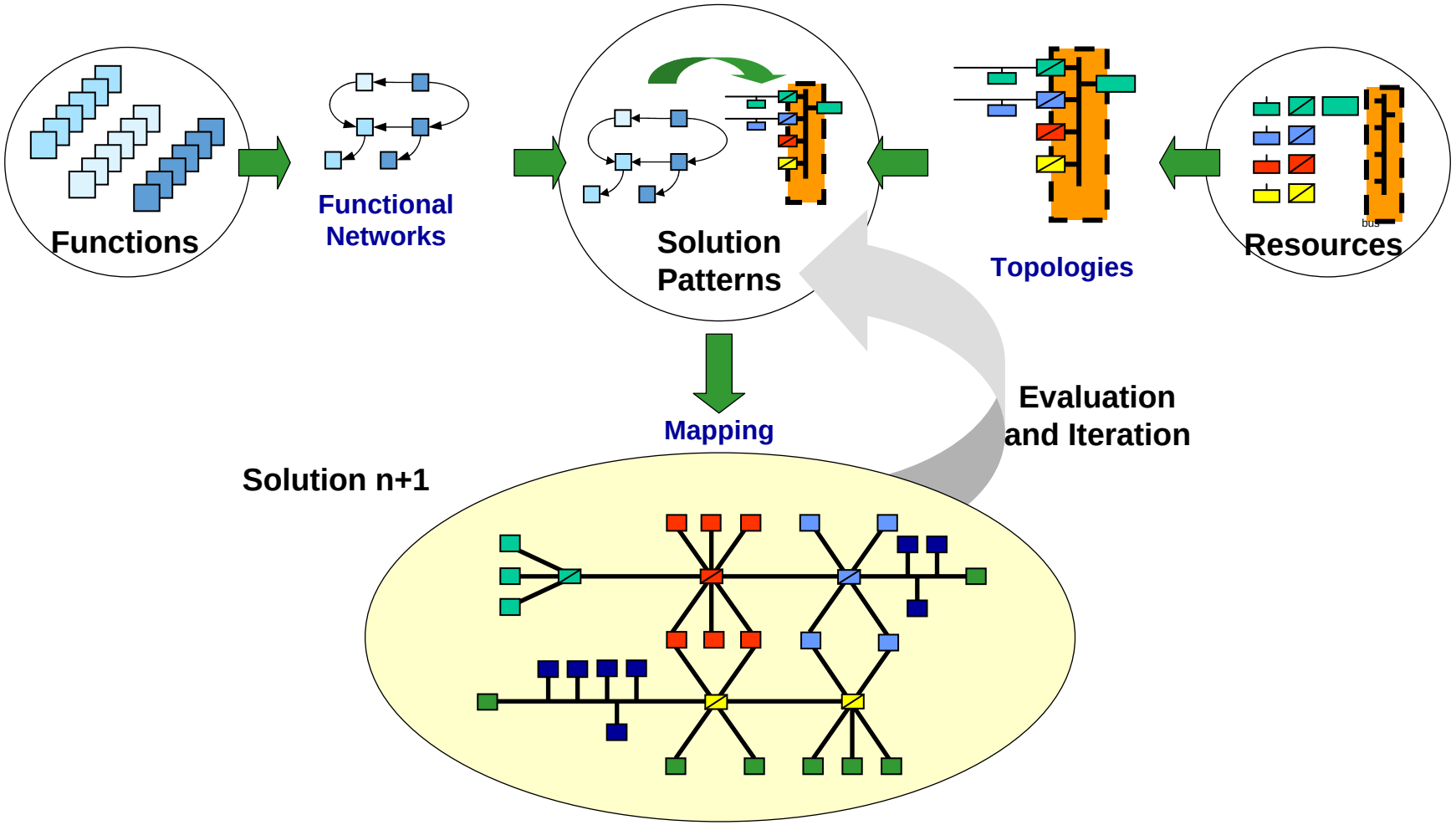
- Embedded System Applications
- Platform based design methodology
- Electronic System Level Design
  - Functions: MoC, Languages
  - Architectures: Network, Node, SoC
- Metropolis
- Conclusions

(Automotive) V-Models: Car level





The Design Components at work



Co-Design Problem

- From:
  - a model of the functionality (e.g. TPM or SPM)
  - a model of the platform (abstraction of topology, network protocol, CPU, Hw/Sw etc)
- Allocate:
  - The tasks to the nodes
  - The communication signals to the network segments
- Schedule:
  - The task sets in each node
  - The packets (mapping signals) in each network segment
- Such that:
  - The system is schedulable and the cost is minimized
- Design solutions:
  - Architectural constrains
  - Analytical approaches
  - Simulation models

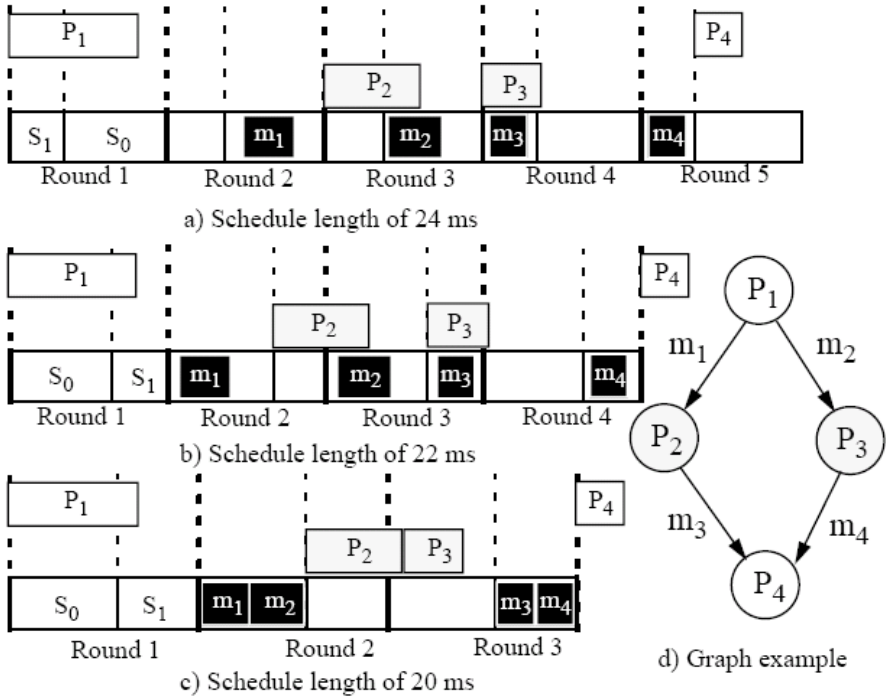
The Time Triggered Approach

- Time Triggered Architecture: Global notion of time
  - Communication and computation are synchronized and MUST HAPPEN AND COMPLETE in a given cyclic *time-division schema*
- *Time-Triggered Architecture (TTA) C. Scheidler, G. Heiner, R. Sasse, E. Fuchs, H. Kopetz*

- Find optimal allocation and scheduling of a Time Triggered TPM

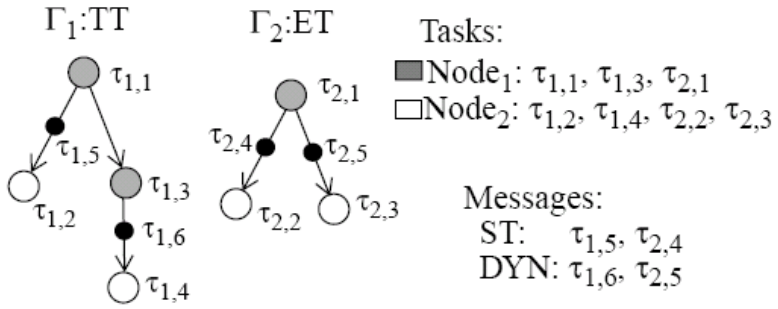
- *An Improved Scheduling Technique for Time-Triggered Embedded Systems, Paul Pop, Petru Eles, and Zebo Peng*
- *Extensible and Scalable Time Triggered Scheduling , EEWei Zheng, Jike Chong, Claudio Pinello, Sri Kanajan, Alberto L. Sangiovanni-Vincentelli*

- *Models of bus/network speed and topology (Hw) and WCET (Hw/Sw) are needed*

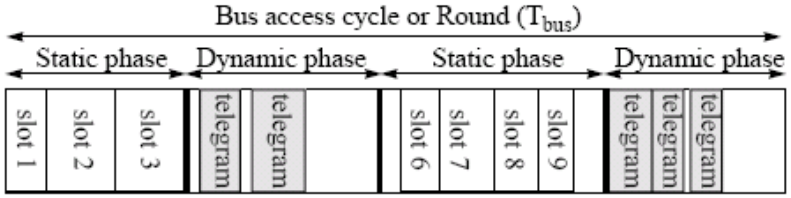


The Holistic Scheduling and Analysis

- Based on a Time and Event Triggered Task Graph Model allocated to a set of nodes
  - Worst Case Execution Time of Tasks and Communication time of each message are known



- Construct a correct static schedule for the TT tasks and ST messages (a schedule which meets all time constraints related to these activities) and conduct a schedulability analysis in order to check that all ET tasks meet their deadlines.

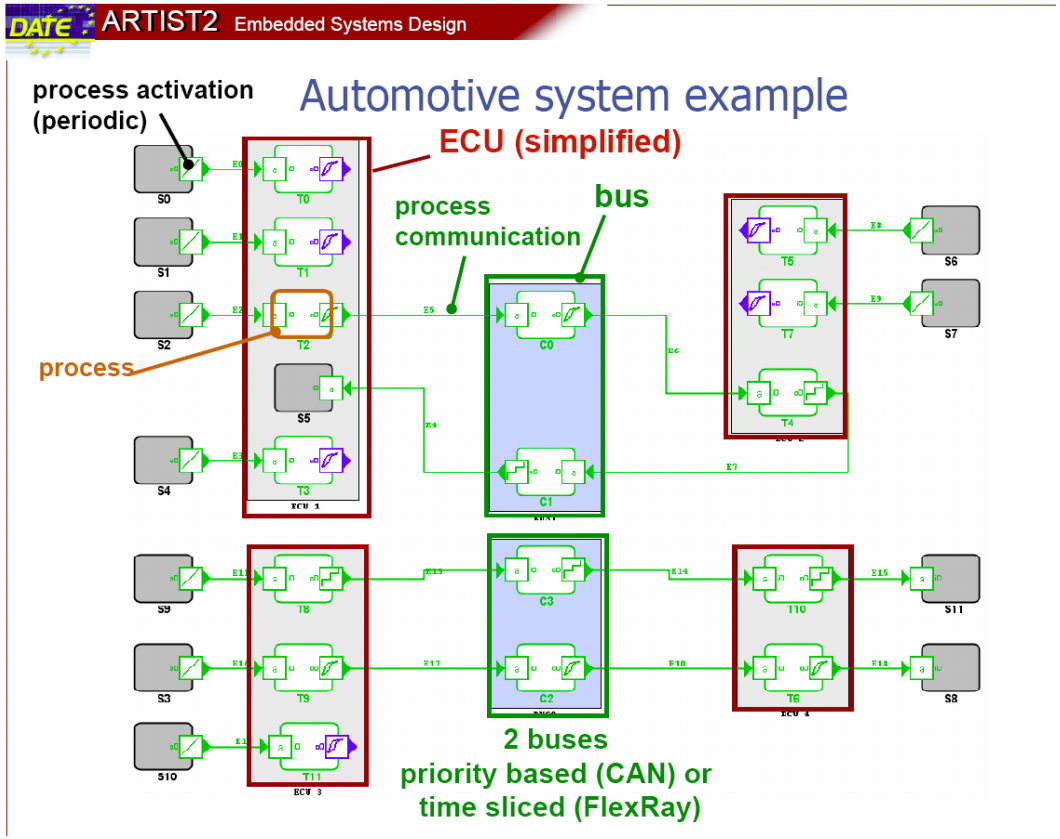


Holistic Scheduling and Analysis of Mixed Time/Event-Triggered Distributed Embedded Systems (2002) Traian Pop, Petru Eles, Zebo Peng

Network Calculus Modelings

Network calculus:

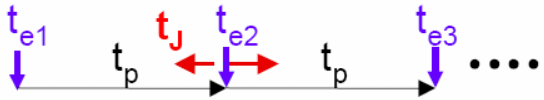
- "Network calculus", J-Y Le Boudec and P. Thiran, Lecture Notes in Computer Sciences vol. 2050, Springer Verlag



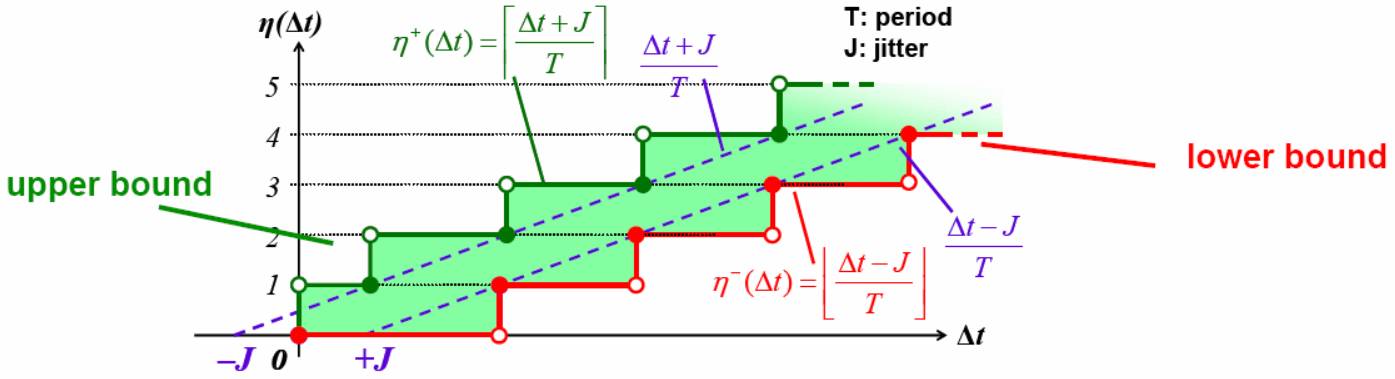
Event Models

Event models

- event stream model w. parameters
  - individual events replaced by stream variables with parameters period, jitter, min. distance, …

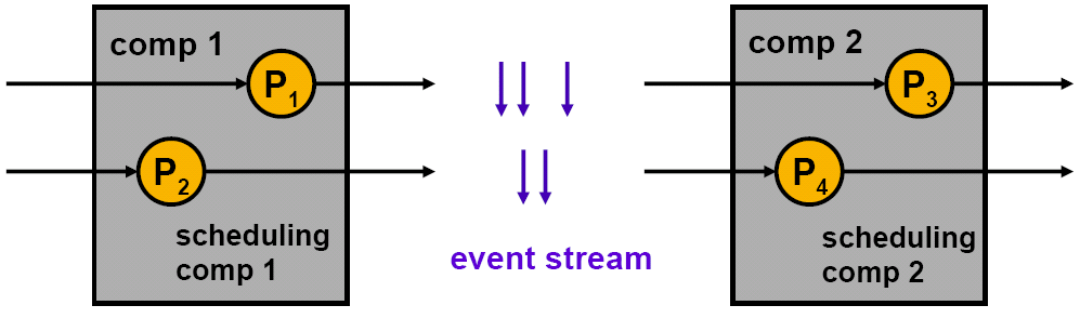


- Network Calculus
  - individual events replaced by sum of events in sliding time window Δt



Composition and Analysis

❖ independently scheduled subsystems are coupled by data flow



Px transformation based on:

- **Output event dependency**
- **WCET**
- **BCET**

⇒ subsystems coupled by stream of data

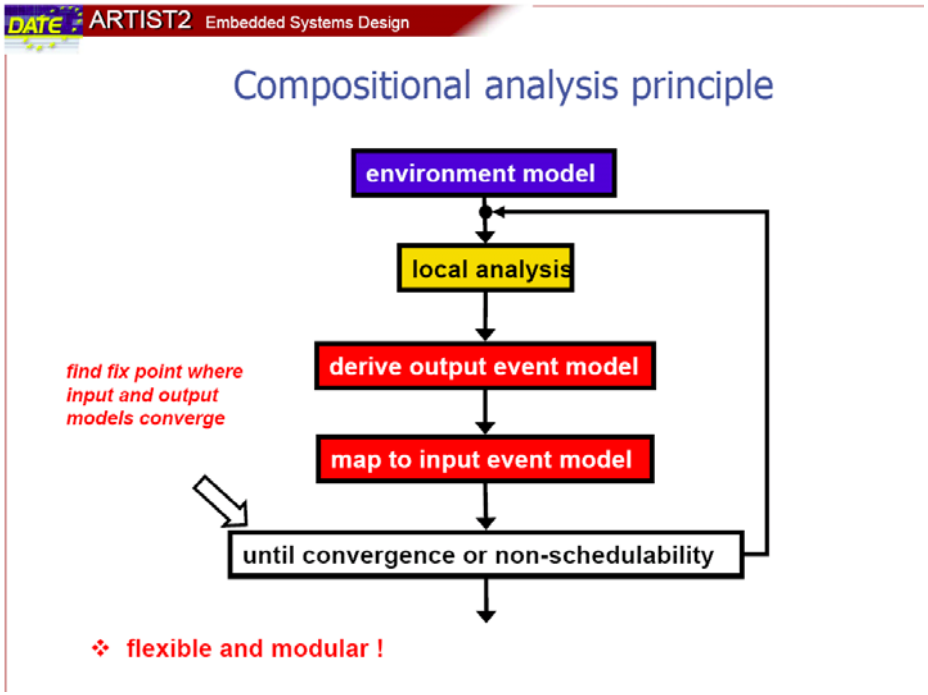
⇒ interpreted as activating events

⇒ coupling corresponds to event propagation

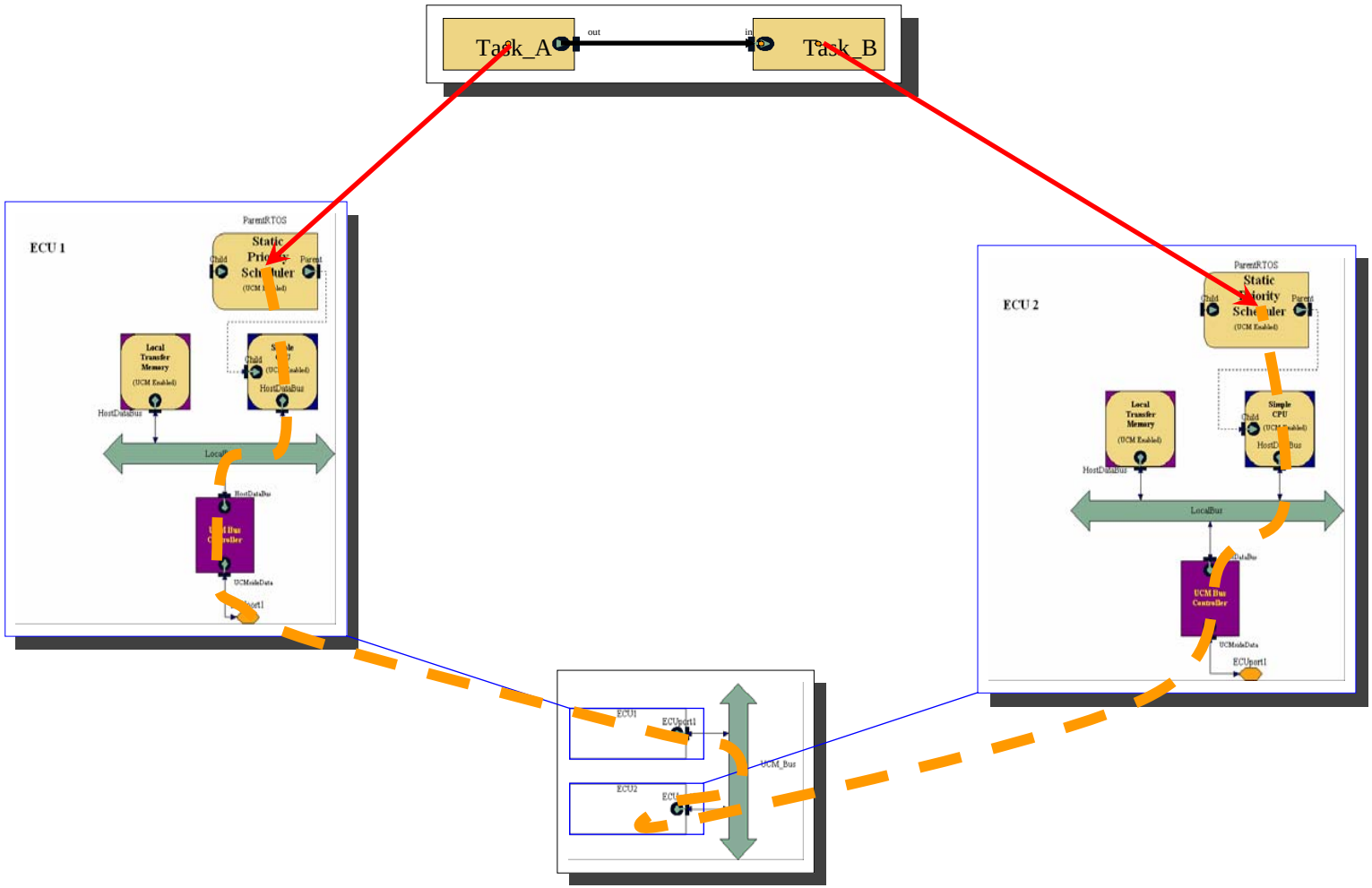
Provide:

- **Schedulability check**
- **Output stream models**

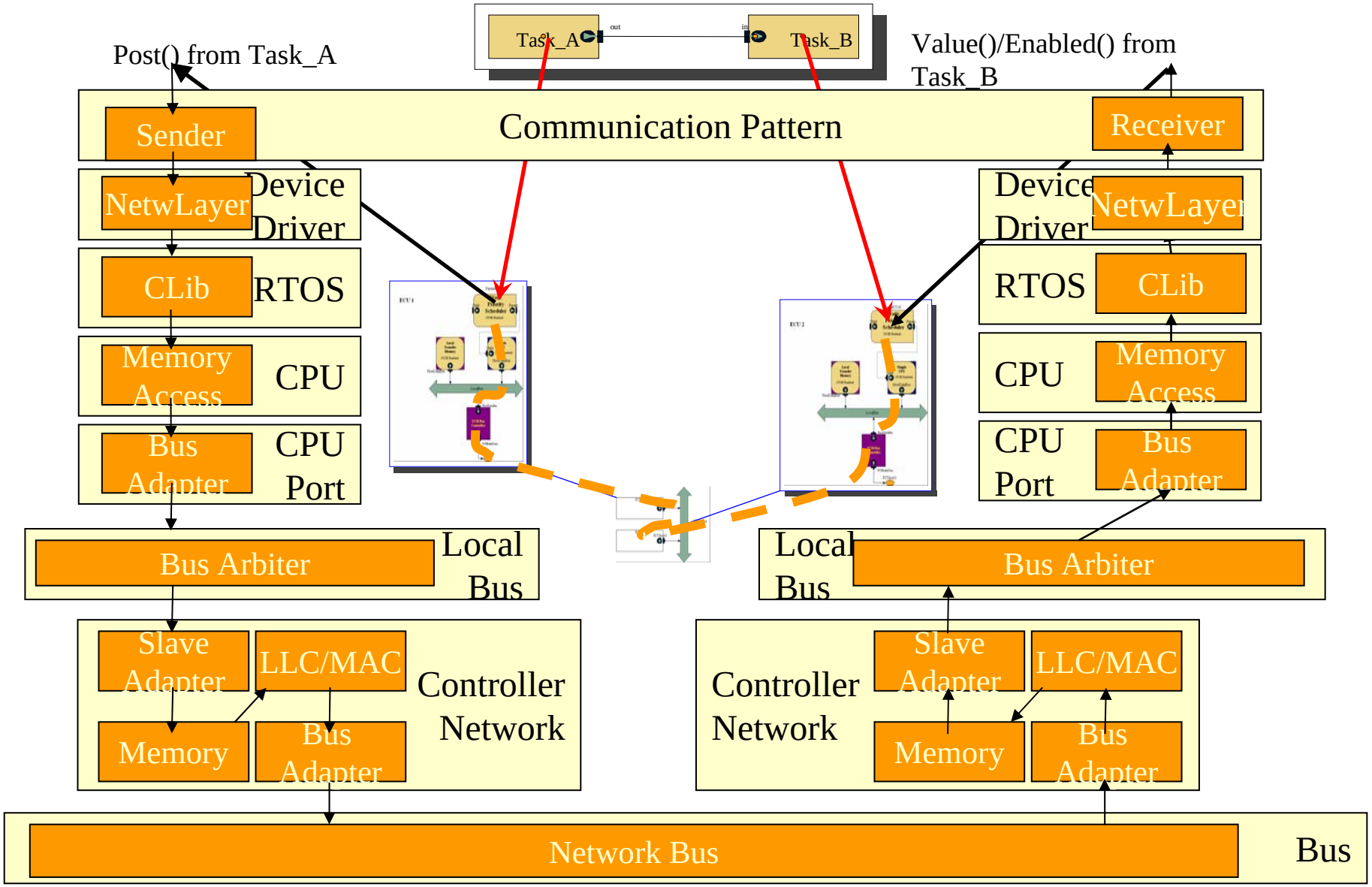
Other strategy to search solutions (allocation and scheduling)



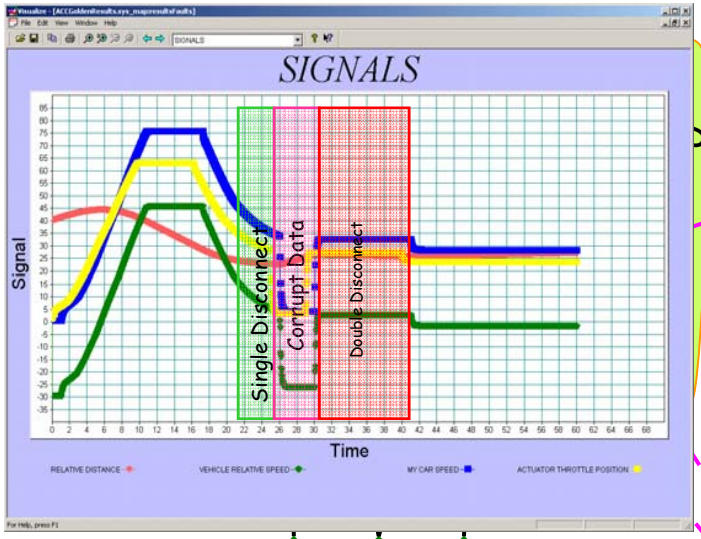
Executable Model: Computation and Communication

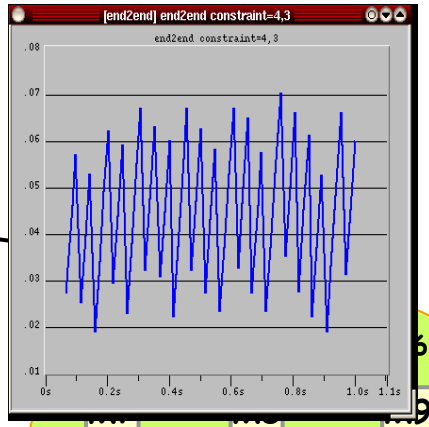


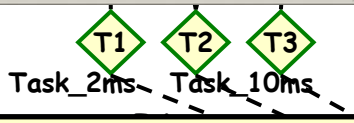
Communication Refinement: Platform Model

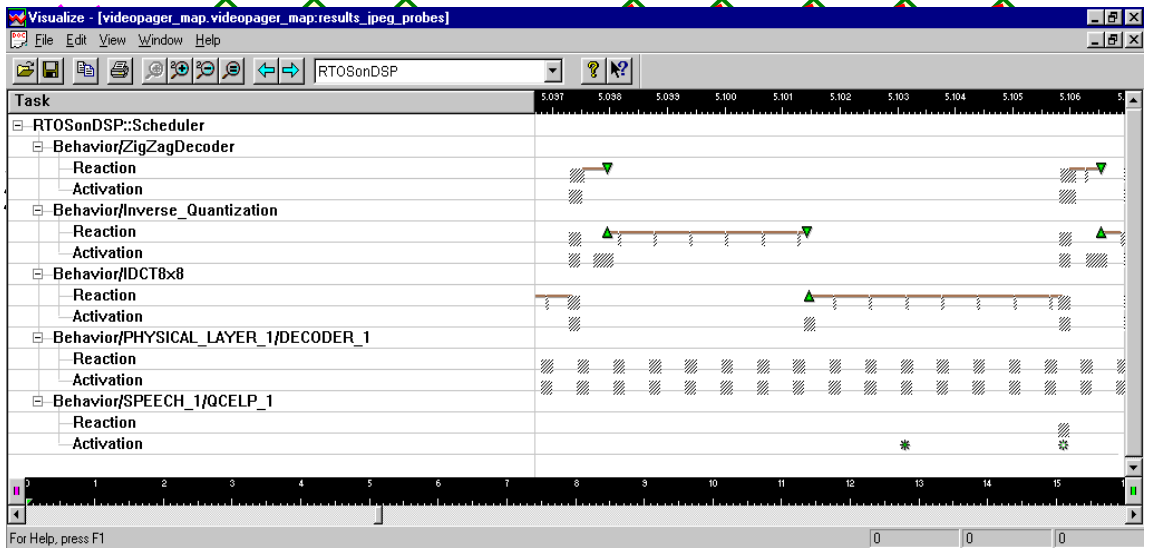


Exploring Solutions by Simulation





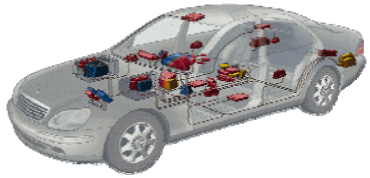




Requires a *model of the functionality* and *performance models of CPUs and network protocols*

It is trace based!

(Automotive) V-Models: Subsystem Level



Development of Distributed System

Development of Sub-System

Development of Mechanical Part (s)

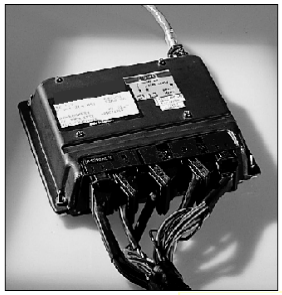
ECU Development

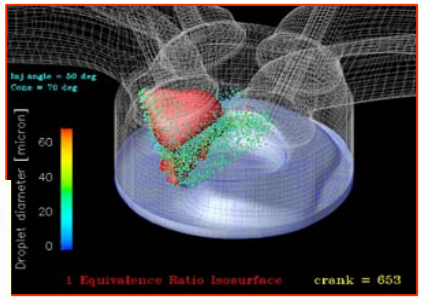
Distributed System Sign-Off!

Sub-System(s) Integration, Test, and Validation

Sub-System Sign-Off!

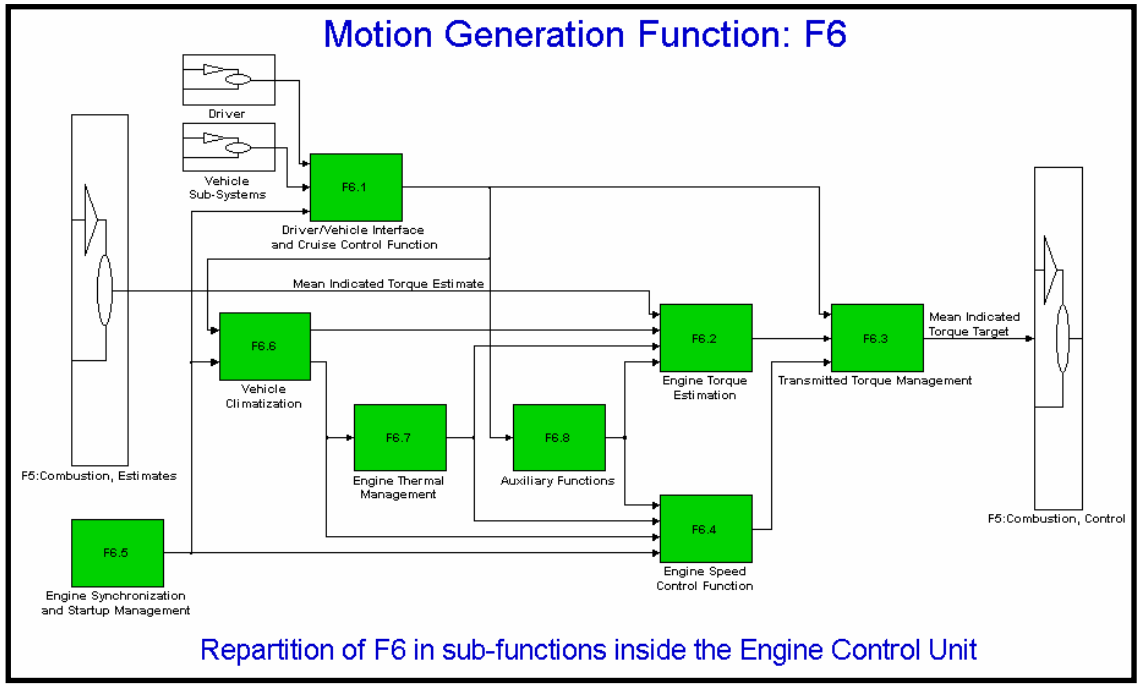
ECU/ Sens./Actrs./Mech. Part(s) Integration, Calibration, and Test





Control system design

- Specifications given at a high level of abstraction:
  - known input/output relation (or properties) and constraints on performance indexes

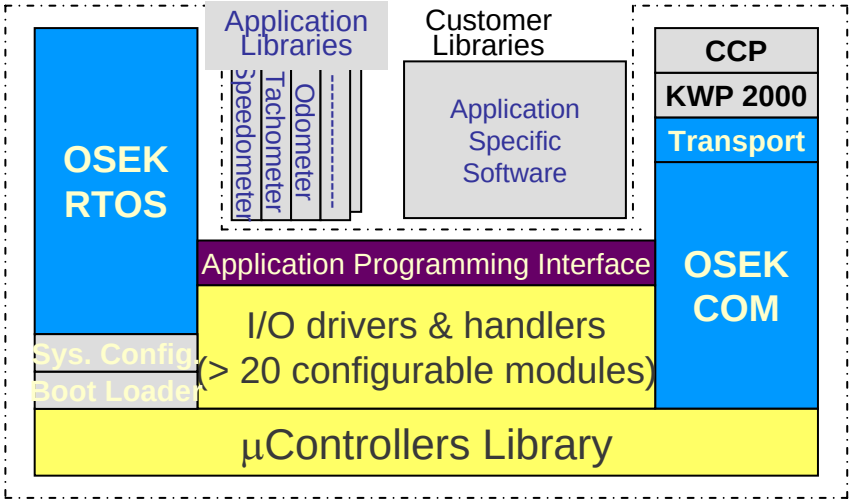


Repartition of F6 in sub-functions inside the Engine Control Unit

- Control algorithms design
- Mapping to different architectures using performance estimation techniques and automatic code generation from models
- Mechanical/Electronic architecture selected among a set of candidates

HW/SW implementation architecture

- a set of possible hw/sw implementations is given by
  - M different hw/sw implementation architectures
  - for each hw/sw implementation architecture $m \in \{1,...,M\}$,
    - a set of hw/sw implementation parameters z
      - e.g. CPU clock, task priorities, hardware frequency, etc.
    - an admissible set X_Z of values for z

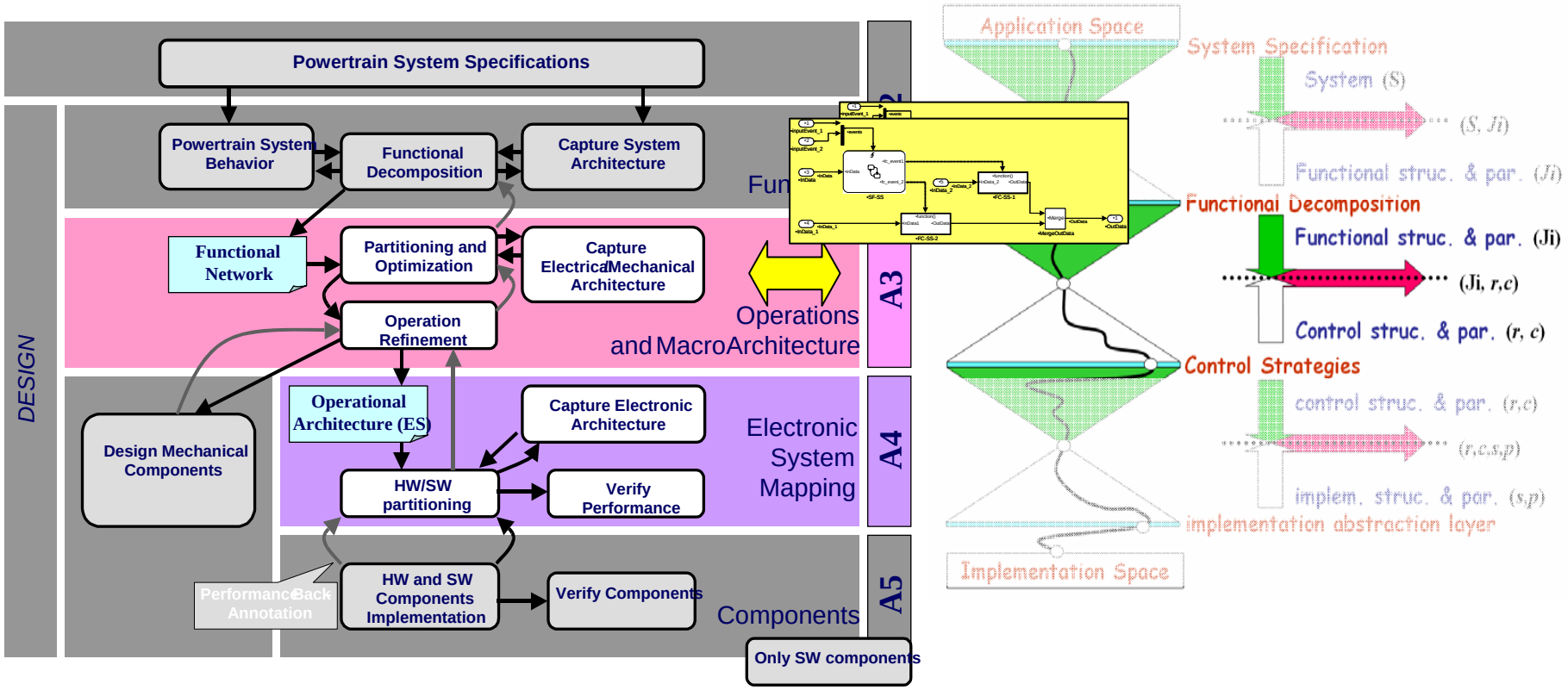


The classical and the ideal design approach

- Classical approach (decoupled design)
  - controller structure and parameters $(r \in R, c \in X_C)$
    - are selected in order to satisfy system specifications
  - implementation architecture and parameters $(m \in M, z \in X_Z)$
    - are selected in order to minimize implementation cost
  - if system specifications are not met, the design cycle is repeated

- Ideal approach
  - both controller and architecture options (r, c, m, z) are selected at the same time to
    - minimize implementation cost
    - satisfy system specifications
  - too complex!!

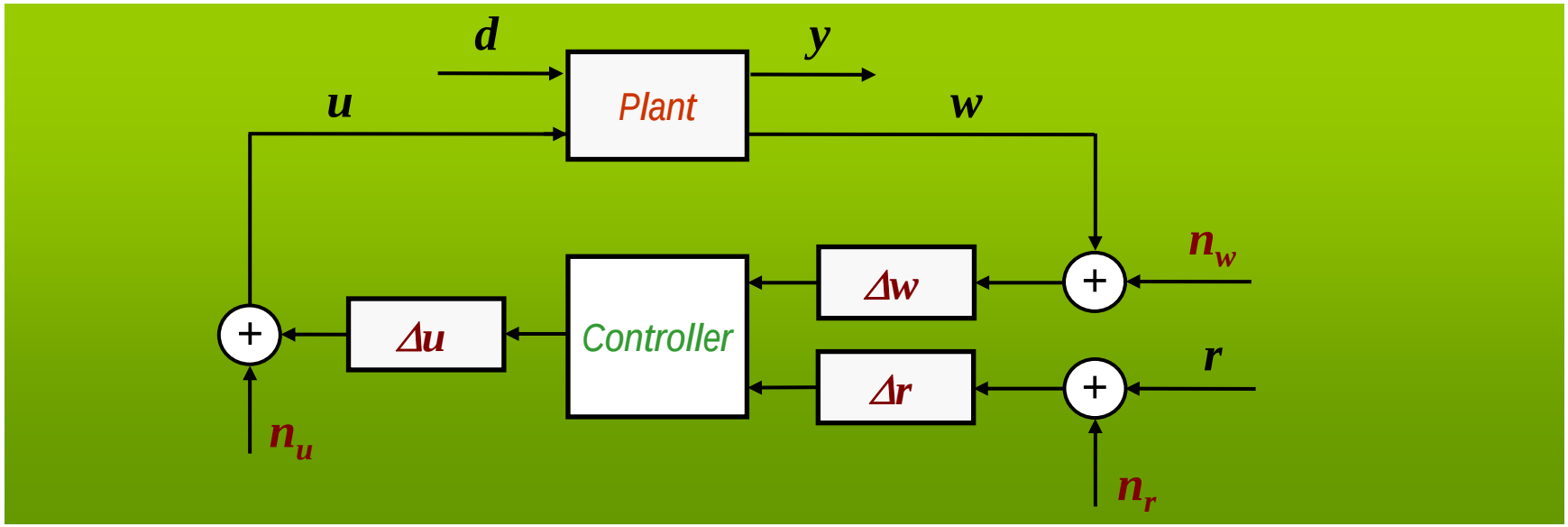
Algorithm Explorations and Control Synthesis



Implementation abstraction layer

- we introduce an implementation abstraction layer
  - which exposes ONLY the implementation non-idealities that affect the performance of the controlled plant, e.g.
    - control loop delay
    - quantization error
    - sample and hold error
    - computation imprecision
- at the implementation abstraction layer, platform instances are described by
  - S different implementation architectures
  - for each implementation architecture $s \in \{1,...,S\}$,
    - a set of implementation parameters p
      - e.g. latency, quantization interval, computation errors, etc.
    - an admissible set X_P of values for p

Effects of controller implementation in the controlled plant performance

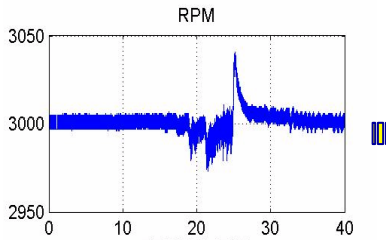


- modeling of implementation non-idealities:
  - $\Delta u, \Delta r, \Delta w$: time-domain perturbations
    - control loop delays, sample & hold , etc.
  - n_u, n_r, n_w :value-domain perturbations
    - quantization error, computation imprecision, etc.

Algorithm Development
Control Algorithm Design

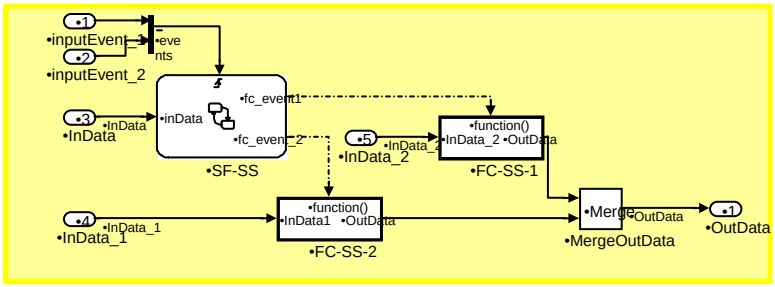
- **Control Algorithm Specification**
  - **Model and Simulation files**
    - **Simulink model**
    - **Calibrations data**
    - **Time history data**

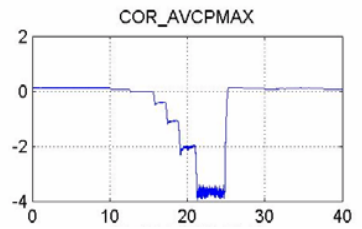
Simulation Results

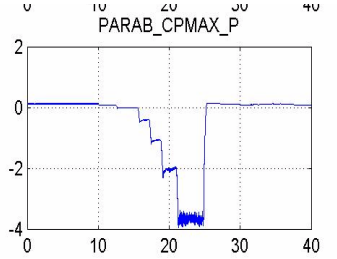


Time History

Simulink Model

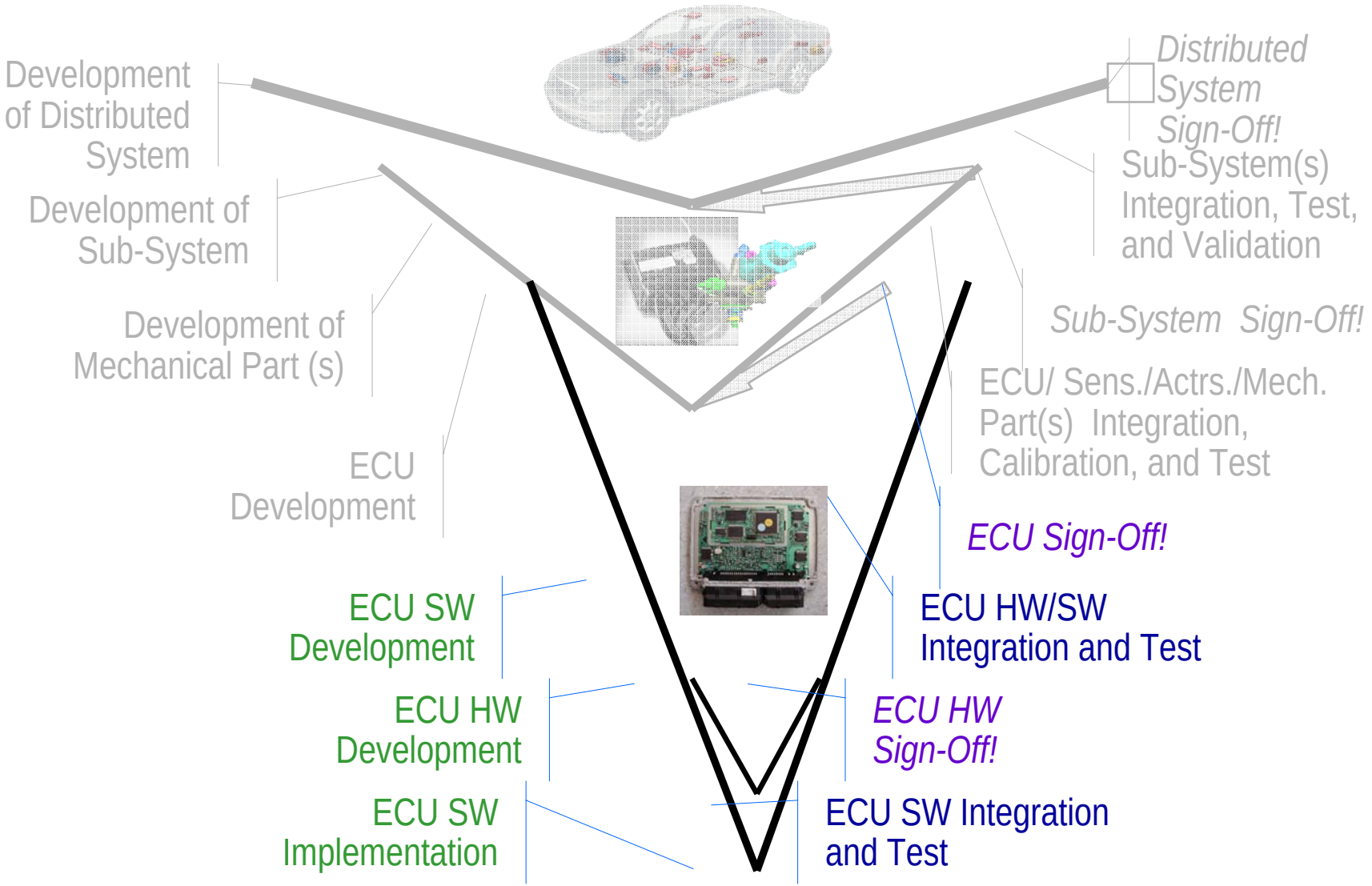






Calibration data

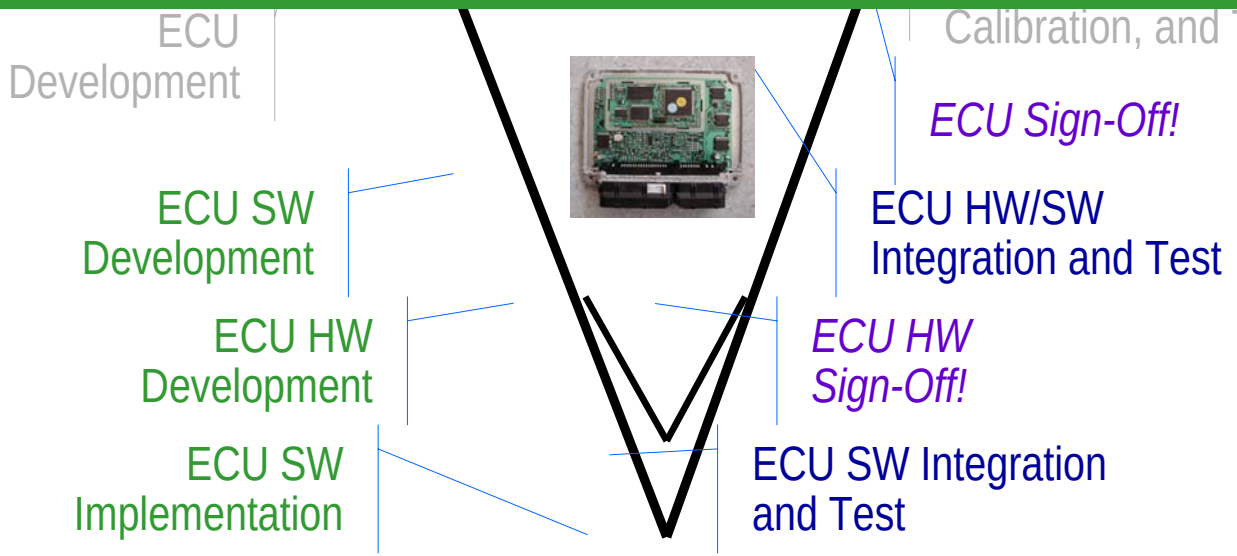
(Automotive) V-Models: ECU level (Hw/Sw)



(Automotive) V-Models: ECU level (Hw/Sw)

Main design tasks:

- Define ECU Hardware/Software Partitioning
  - Platform instance structure selection
- Software Implementation
- Hardware (SoC) Design and Implementation



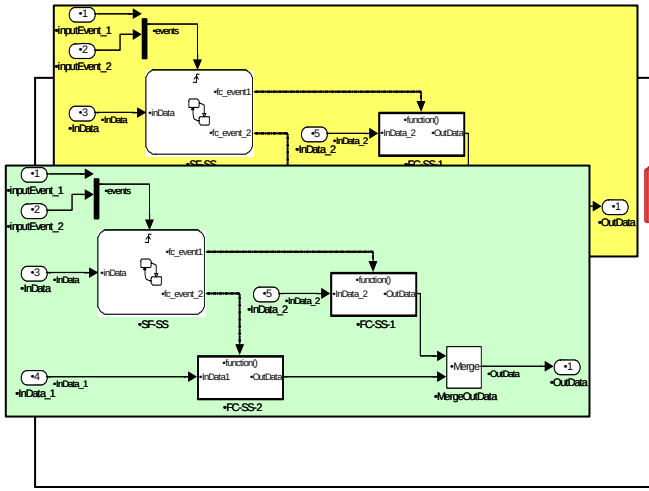
Control Algorithm Implementation Strategy

- Control algorithms are mapped to the target platform to achieve the best performance/cost trade-off.
  - In most cases the platform can accommodate in software the control algorithms, if not:
  - New platform services might be required or
  - New hardware components might be implemented or
  - New control algorithms must be explored.

Platform Design Strategy

- Minimize software development time
  - Maximize model based software
    - Software generation is possible today from several MoC and languages:
      - StateCharts, Dataflow, SR, …
    - Implement the same MoC of specification or guarantee the equivalence
    - Fit into the chosen software architecture to maximize reuse at component level
      - E.g. AUTOSAR for automotive
  - Maximize the reuse of hand-written software component
    - Define application and platform software architecture
- Minimize the change requests for the hardware platform
  - Implement as much as possible in software

System Platform Definition



Application Software

Sensor/Actuator Layer

The software application is composed of model-based and hand-written application-dependent software components (sources)

Software Platform
(API services)

RTOS | Net

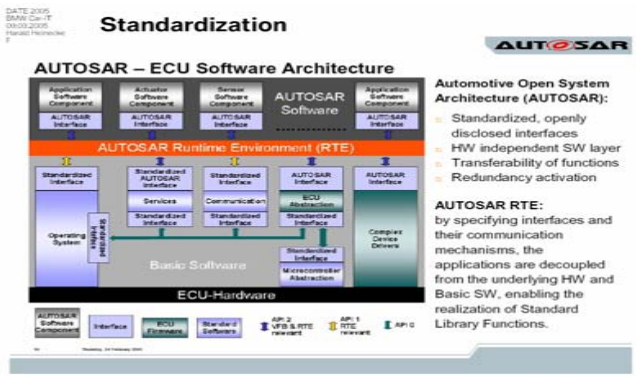
Device Drivers

BIOS

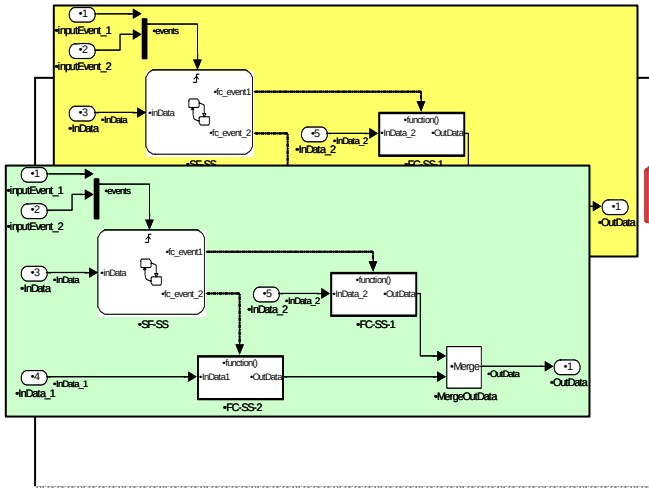
CPUs

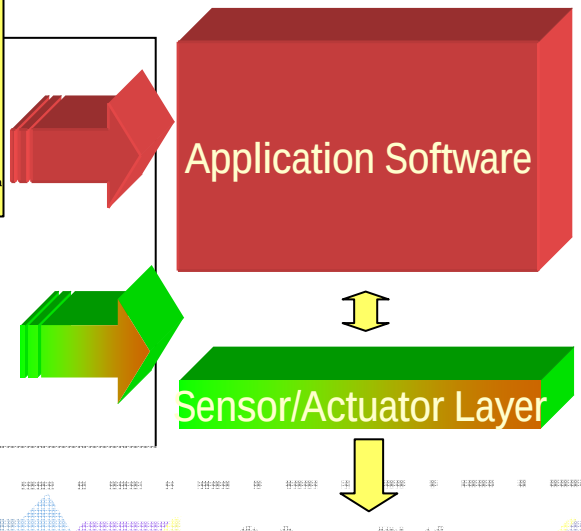
ECU output devices | ECU input devices

The software platform is cross applications and cross HW plats and is composed of parameterized software components (sources)

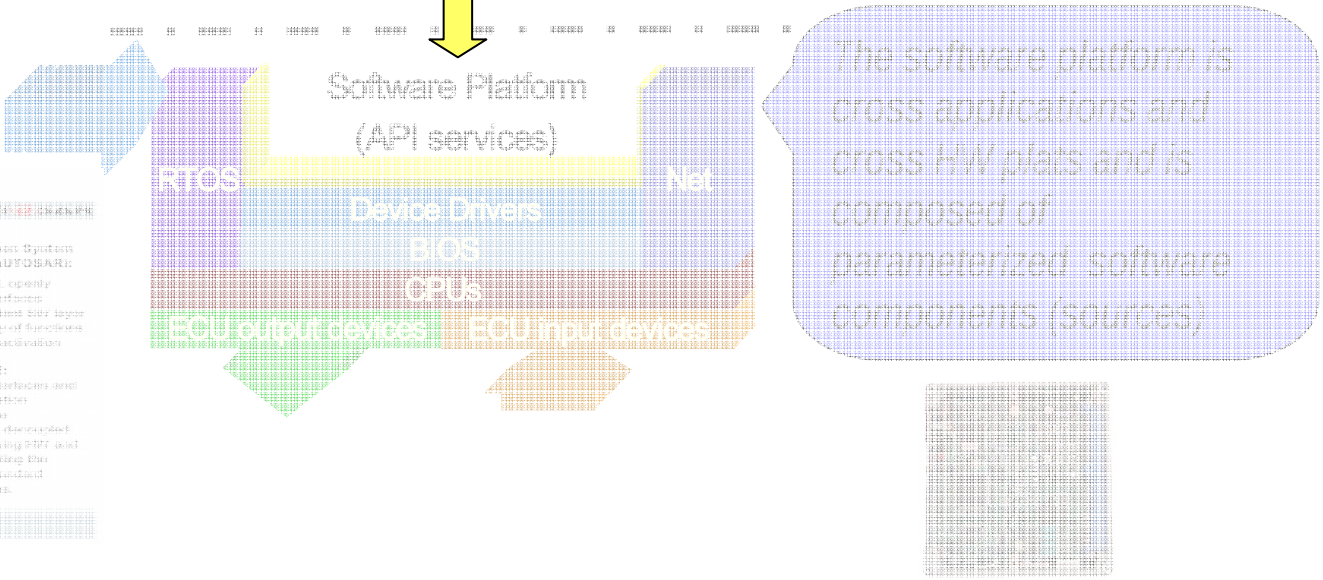


Software Implementation Flow



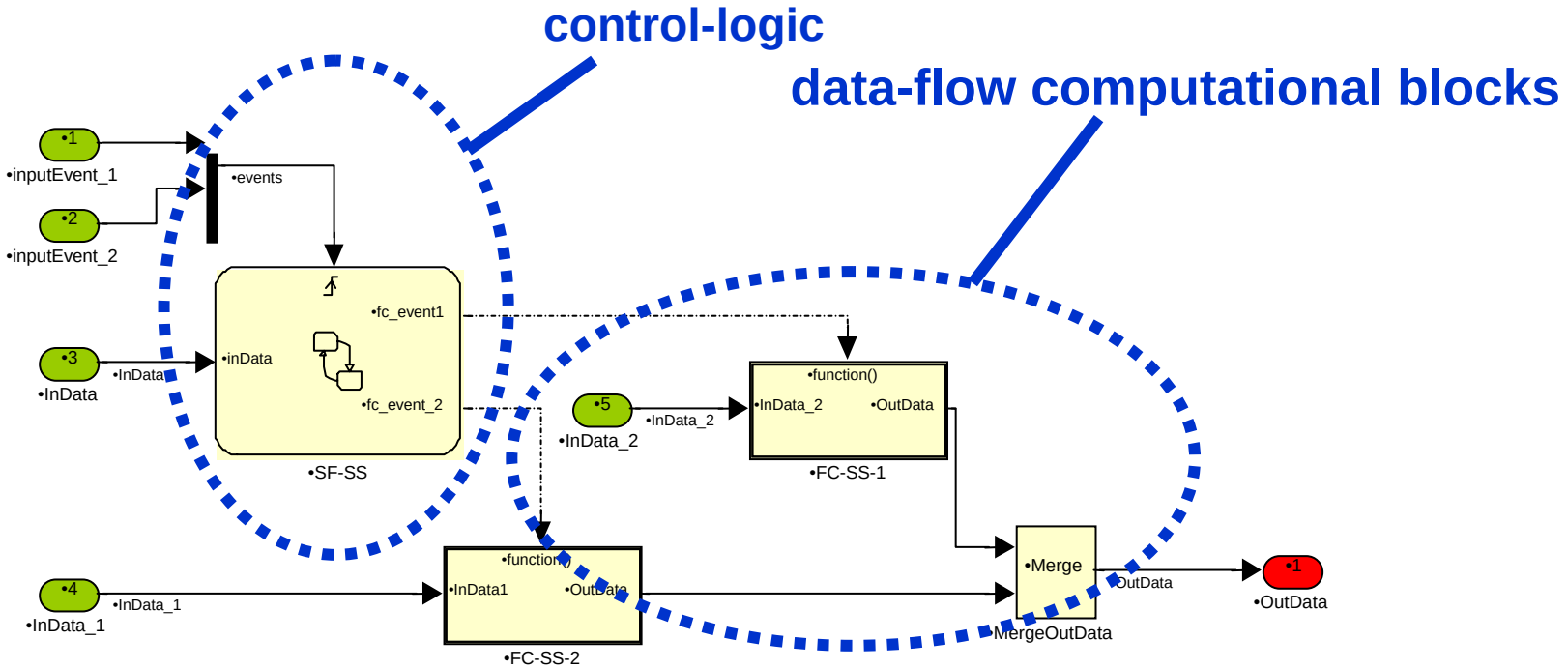


The software application is composed of model-based and hand-written application-dependent software components (sources)



Exampe of Specification of Control Algorithms

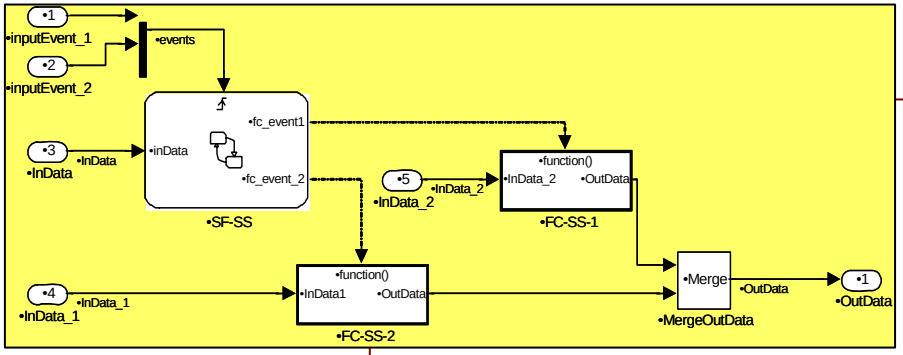
- A control algorithm is a (synch or a-synch) composition of extended finite state machines (EFSM).



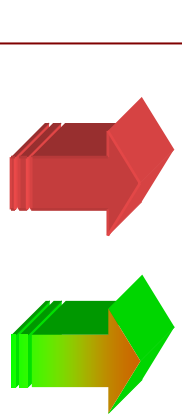
Code Generation

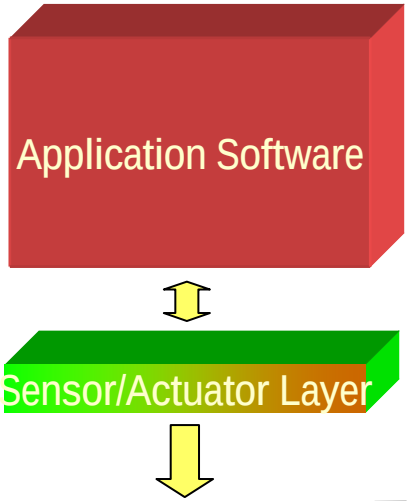
- Mapping a functional model to software platform:
  - Data refinement
  - Software platform services mapping (communication and computation)
  - Time refinement (scheduling)
- Data refinement
  - Float to Fixed Point Translation.
    - Range, scaling and size setting (by the designer).
    - Worst case analysis for internal variable ranges and scaling.
  - Signals and parameters to C-variables mapping.
- Software platform model:
  - variables and services (naming).
    - Access variable method are mapped with variable classes.
  - execution model:
    - Multi-rate subsystems are implemented as multi-task software components scheduled by an OSEK/VDX standard RTOS
- Time refinement
  - Task scheduling

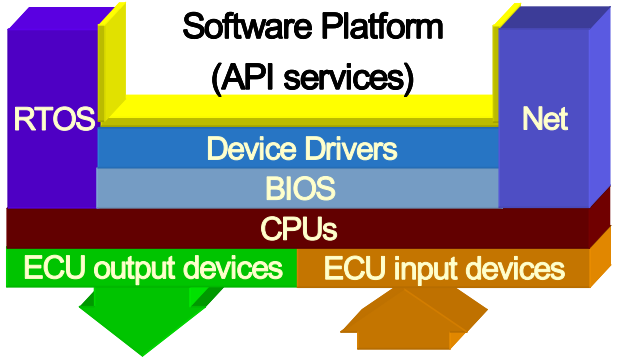
Mapping Control Algorithms to the Platform



Automatic synthesis







Handwritten code

From high level models:

- Automatic translation to C/C++ code
- (Semi)-Automatic data refinement for computation
- Automatic refinement of communication services

Flow examples:

ASCET, Simulink/eRTW/TargetLink, UML

Example: Gasoline Direct Injection Engine Control

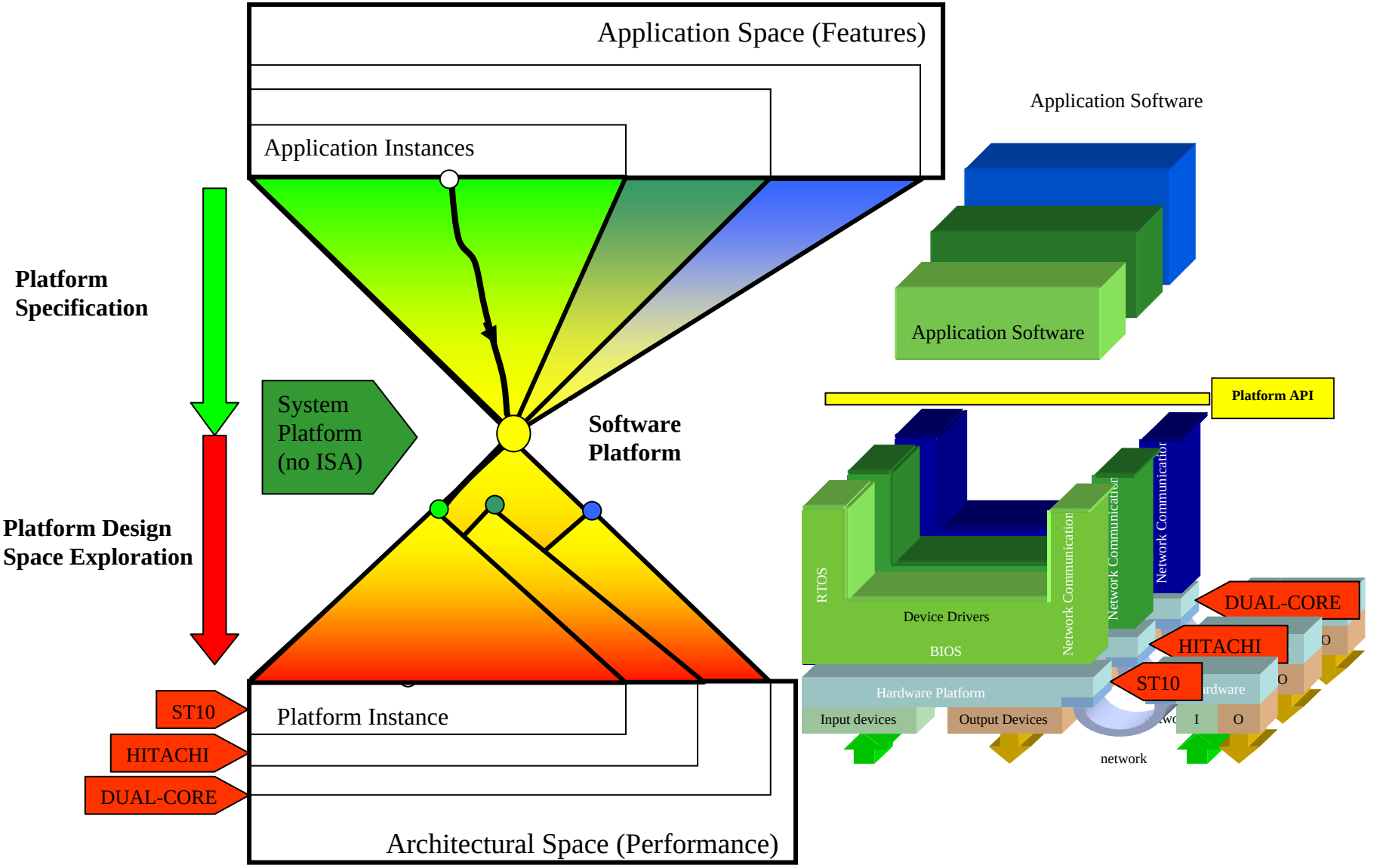
	Modelled Components	SLOC	% of Model Compiled SLOC
Platform Components	26-HandCoded	26500	0%
Application Components	86-AutomCoded 13-HandCoded	93600	90%

	% of the total memory occupation	
	ROM %	RAM %
Platform	17.9	2.9
Application	82.1	97.1

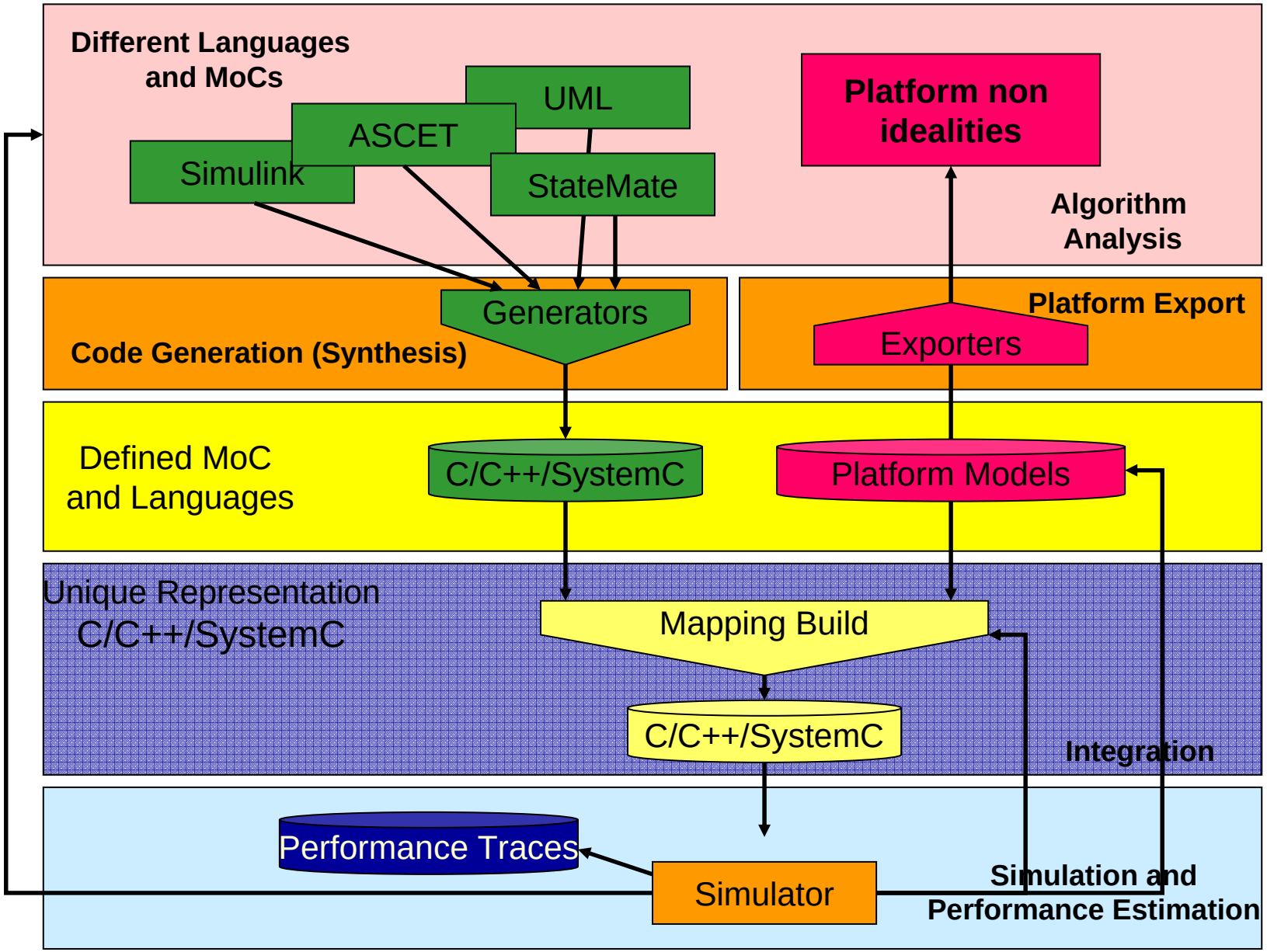
Example: Gasoline Direct Injection Engine Control

- Tremendous increase in application-software productivity:
  - Up to 4 time faster than in the traditional hand-coding cycle.
- Tremendous decrease in verification effort:
  - Close to 0 ppm
- Tremendous reuse of modes and source code

Defining the Platform



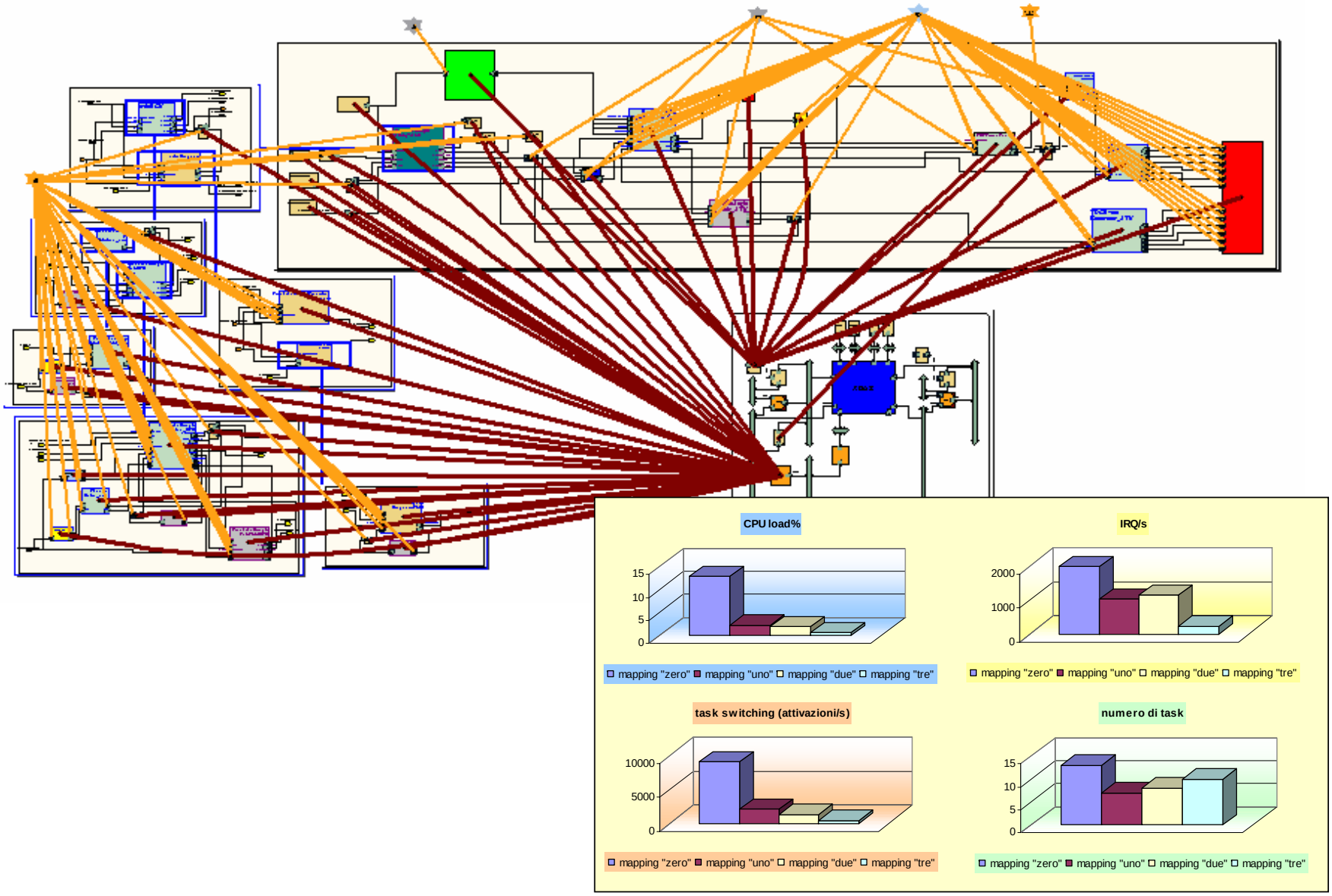
Simulation Based (C/C++/SystemC) Exploration Flow



SystemC and OCP Abstraction Levels

Communication (I/F)			
SystemC	Abstraction Accuracy	OCP Layers	Abstraction Removes
Untimed Functional	Token	Message (L-3)	Time Resource Sharing
Programmers View (PV)	+Address		
Programmers View + Time (PVT)	+Transaction time	Transaction (L-2)	Clocks, protocols
Bus cycle Accurate (BCA)	+Clock cycle	Transfer (L-1)	Wire registers
Pin Cycle Accurate (PCA)	+Pin/clock	RTL (L-0)	Gates
Computation			
Untimed Functional (UTF)	Function		
Time Functional (TF)	+Computation Time		
Register Transfer (RT)	+Clock cycle		

Mapping application to platform



SW estimation

- SW estimation is needed to
  - Evaluate HW/SW trade-offs
  - Check performance/constraints
    - Higher reliability
  - Reduce system cost
    - Allow slower hardware, smaller size, lower power consumption

SW estimation: Static vs. Dynamic

- Static estimation
  - Determination of runtime properties at compile time
  - Most of the (interesting) properties are undecidable => use approximations
  - An approximation program analysis is safe, if its results can always be depended on.
    - E.G. WCET, BCET
  - Quality of the results (precision) should be as good as possible
- Dynamic estimation
  - Determination of properties at runtime
  - DSP Processors
    - relatively data independent
    - most time spent in hand-coded kernels
    - static data-flow consumes most cycles
    - small number of threads, simple interrupts
  - Regular processors
    - arbitrary C, highly data dependent
    - commercial RTOS, many threads
    - complex interrupts, priorities

SW estimation overview

- Two aspects to be considered
  - The structure of the code (program path analysis)
    - E.g. loops and false paths
  - The system on which the software will run (micro-architecture modeling)
    - CPU (ISA, interrupts, etc.), HW (cache, etc.), OS, Compiler
- Level at which it is done
  - Low-level
    - e.g. gate-level, assembly-language level
    - Easy and accurate, but long design iteration time
  - High/system-level
    - Fast: reduces the exploration time of the design space
    - Accurate “enough”: approximations are required
    - Processor model must be cheap
      - “what if” my processor did X
      - future processors not yet developed
      - evaluation of processor not currently used
    - Must be convenient to use
      - no need to compile with cross-compilers and debug on my desktop

SW estimation in VCC

Virtual Processor Model (VPM)
compiled code virtual instruction set simulator

- An virtual processor functional model with its own ISA estimating computation time based on a table with instruction time information
  - Pros:
    - does not require target software development chain (uses host compiler)
    - fast simulation model generation and execution
    - simple and cheap generation of a new processor model
    - Needed when target processor and compiler not available
  - Cons:
    - hard to model target compiler optimizations (requires “best in class” Virtual Compiler that can also as C-to-C optimization for the target compiler)
    - low precision, especially for data memory accesses

SW estimation by ISS

Interpreted instruction set simulator (I-ISS)

- A model of the processor interpreting the instruction stream and accounting for clock cycle accurate or approximate time evaluation
  - Pros:
    - generally available from processor IP provider
    - often integrates fast cache model
    - considers target compiler optimizations and real data and code addresses
  - Cons:
    - requires target software development chain and full application (boot, RTOS, Interrupt handling, etc)
    - often low speed
    - different integration problem for every vendor (and often for every CPU)
    - may be difficult to support communication models that require waiting to complete an I/O or synchronization operation

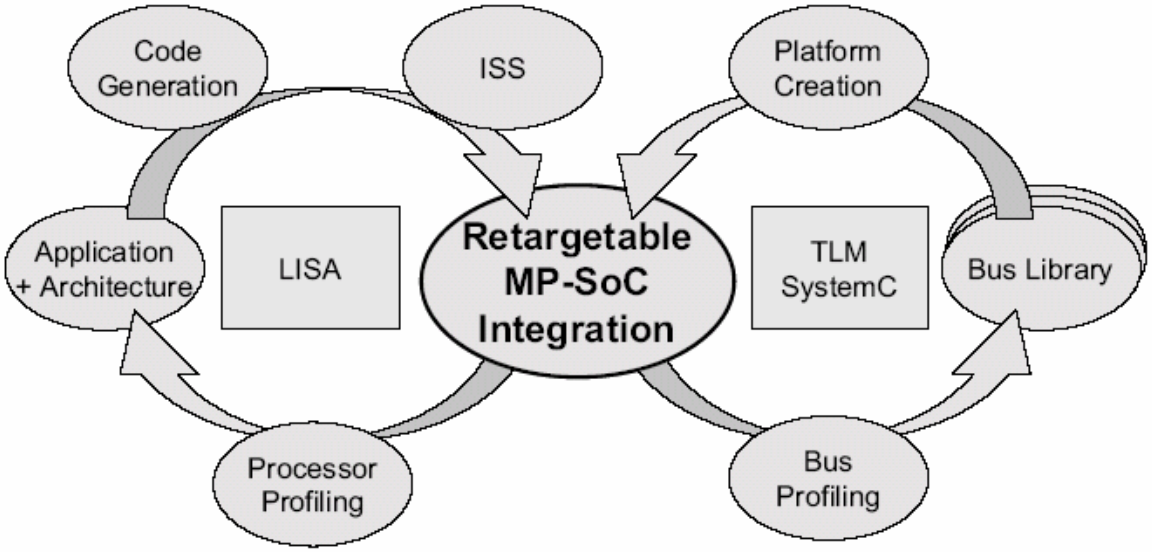
Accuracy vs Performance vs Cost

	Accuracy	Speed	$$$*
Hardware Emulation	+++	+-	---
Cycle accurate model	++	--	--
Cycle counting ISS	++	+	-
Dynamic estimation	+	++	++
Static spreadsheet	-	+++	+++

*$$$ = NRE + per model + per design

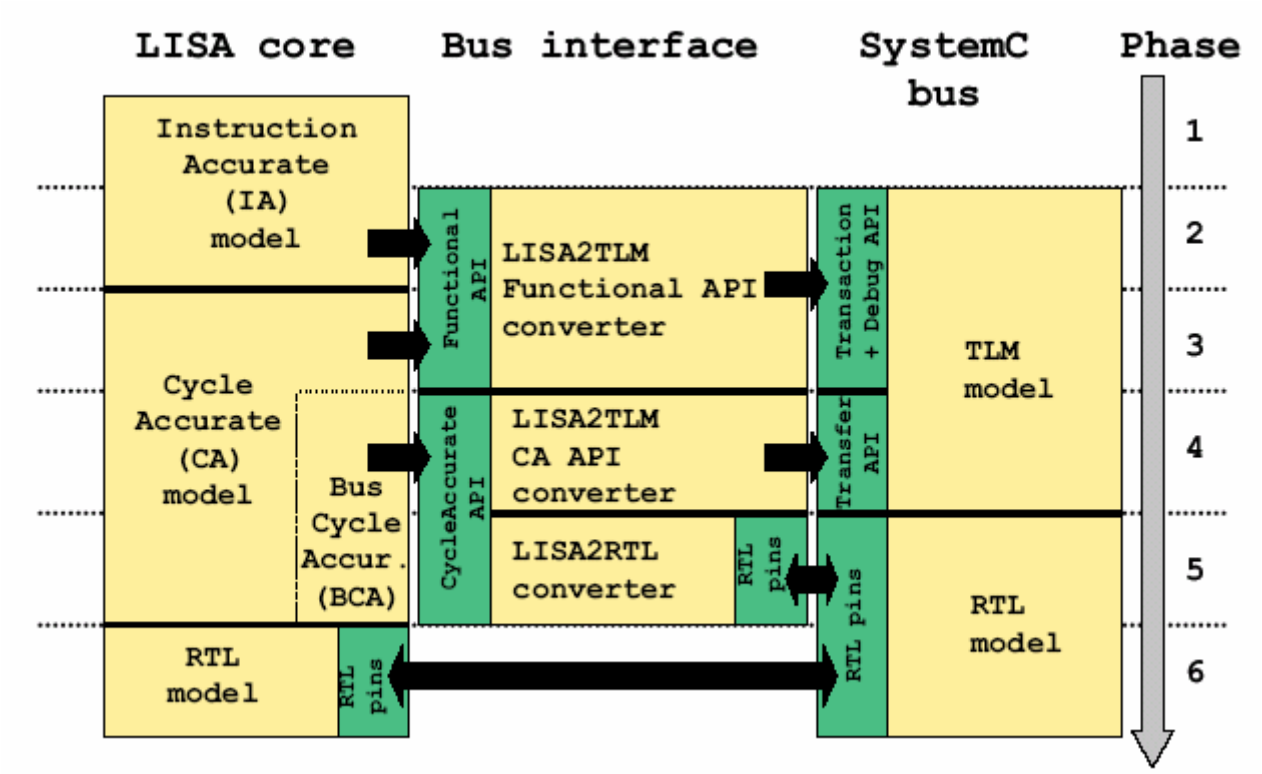
CoWare Platform Modeling Environment

- Focus on computation/communication separation
- Leverage their LISA platform and SystemC Transaction Level Models

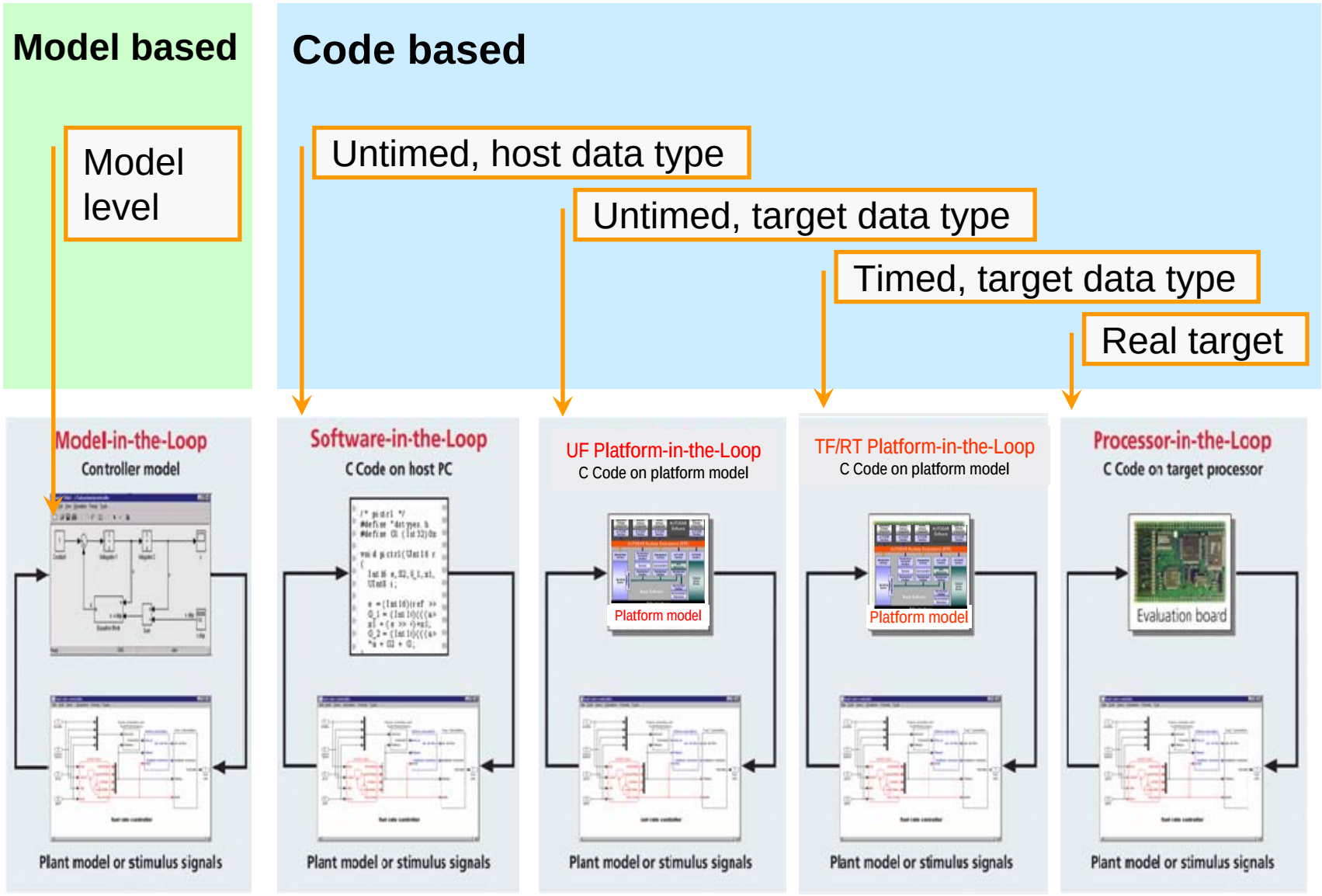


CoWare Support for Multiple Abstraction Levels

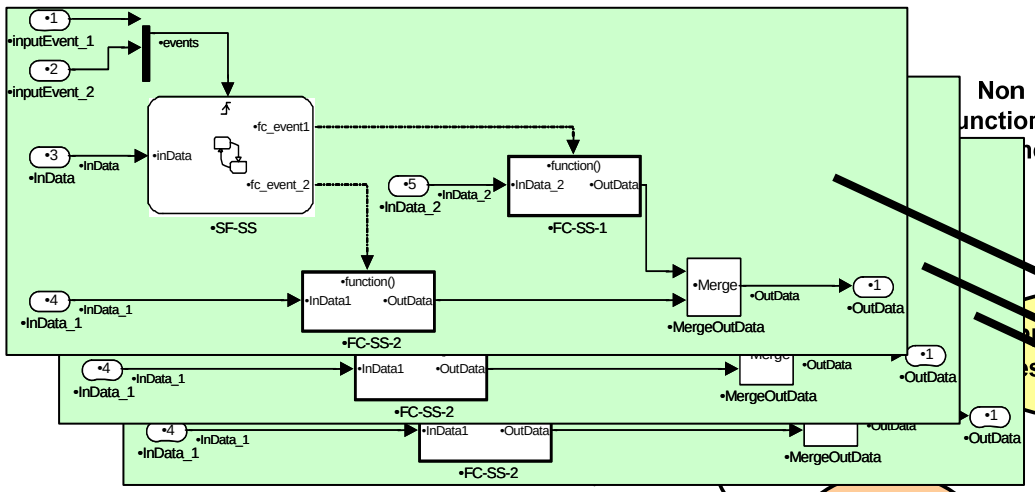
- Support successive refinement for both processors and bus models
- Depending on abstraction level, simulation performance of 100 to 200 Kcycles/sec



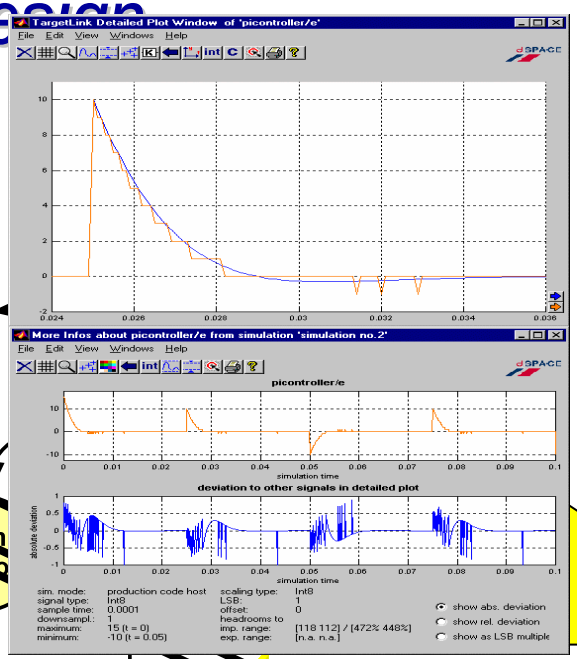
Refining the Control Algoritm



Model Based Control-Platform Co-Design



Non functional elements



Control Specification

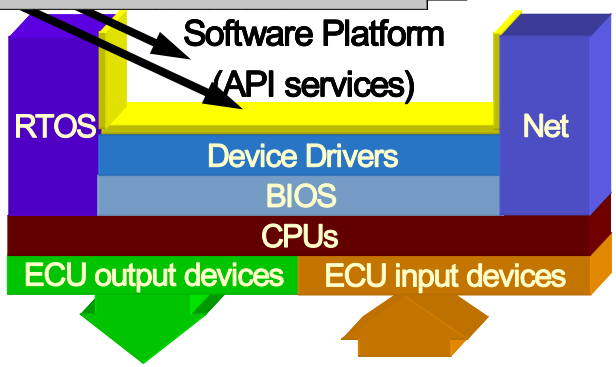
Function/ Platform Mapping

Verification

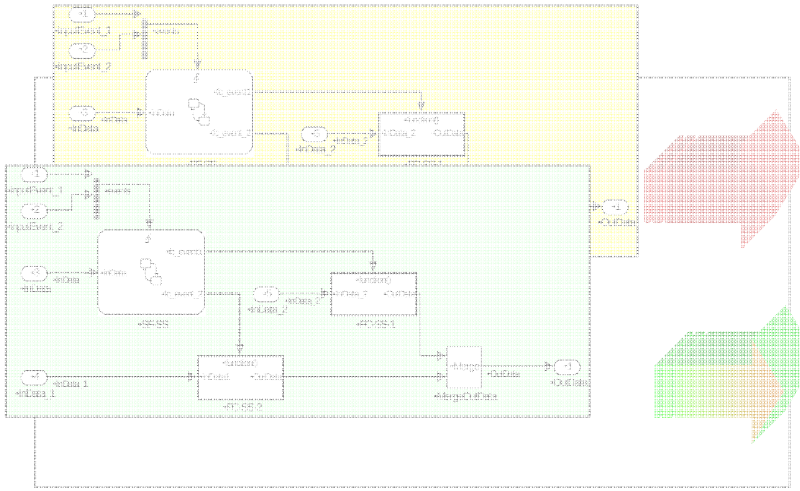
Synthesis

```
void integratutto4_initializer( void )
{
  /* Initialize machine's broadcast event variable */
  _sfEvent_ = CALL_EVENT;

  _integratutto4MachineNumber_ =
    sf_debug_initialize_machine("integratutto4","sfun",0,3,0,0,0);
  sf_debug_set_machine_event_thresholds(_integratutto4MachineNumber_,0,0);
  sf_debug_set_machine_data_thresholds(_integratutto4MachineNumber_,0);
}
```



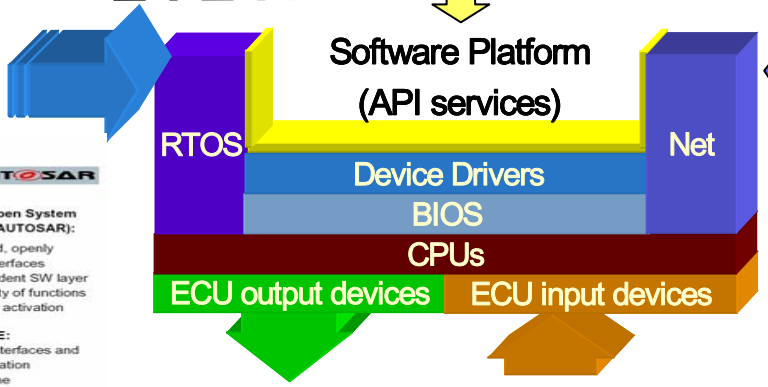
Platform Design



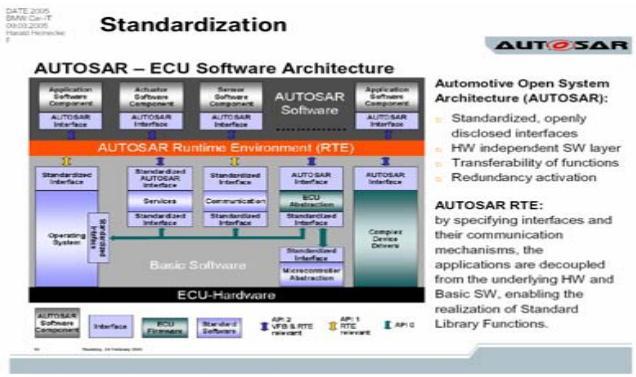
Application Software

Sensor/Actuator Layer

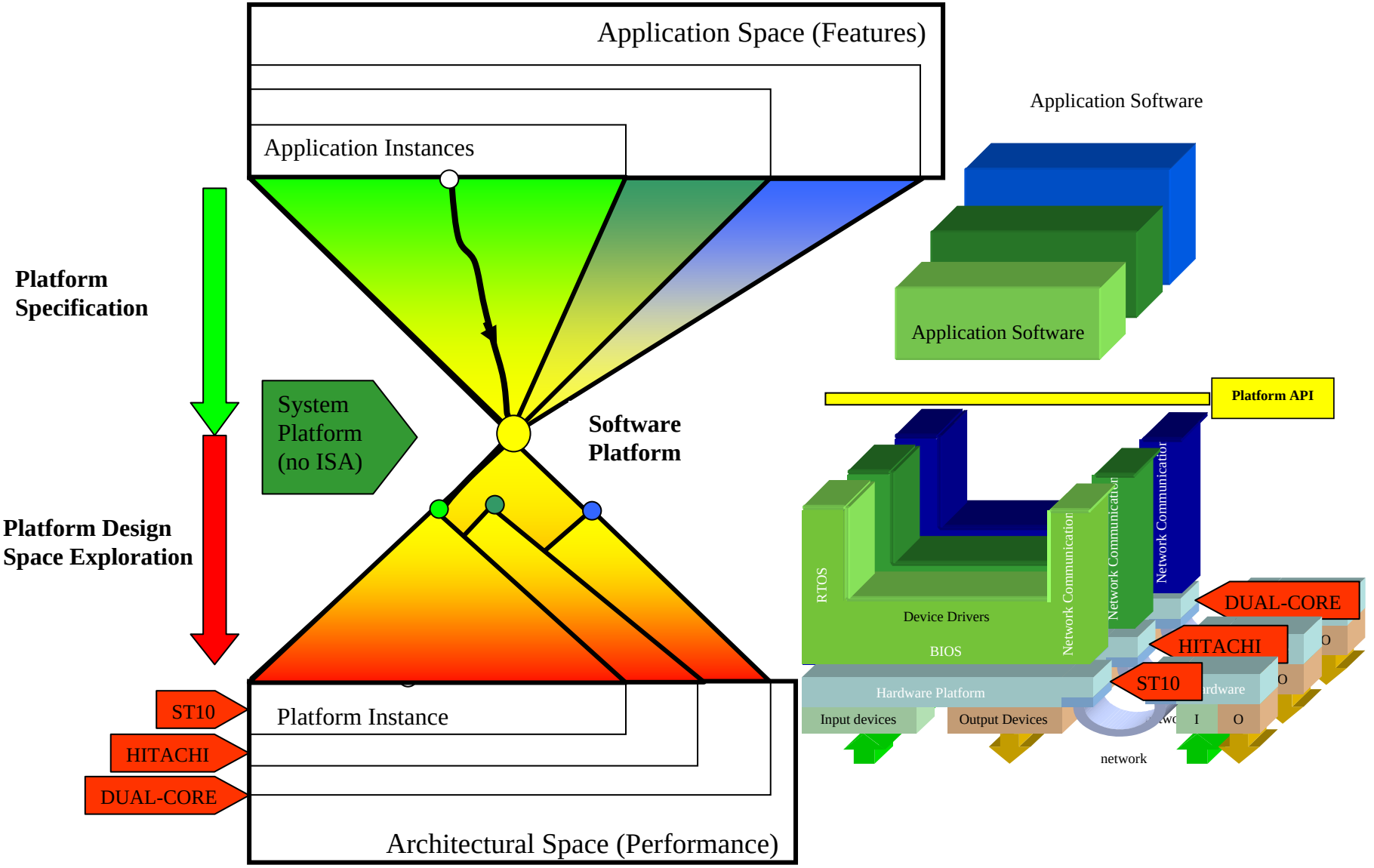
The software application is composed of model based and hand written application-dependent software components (sources)



The software platform is cross applications and cross HW plats and is composed of parameterized software components (sources)



Choosing an Implementation Architecture



Platform Design and Implementation

- Hardware, computation:
  - Cores:
    - Core selection
    - Core instantiation
  - Coprocessors:
    - Selection (Peripherals)
    - Configuration/Synthesis
  - Instructions:
    - ISA definition (VLIW)
    - ISA Extension Flow
- Hardware, communication:
  - Busses
  - Networks

- Software, granularity:
  - Set of Processes
  - Process/Thread
  - Instruction sequences
  - Instructions
- Software, layers:
  - RTOS
  - HAL
  - Middle layers

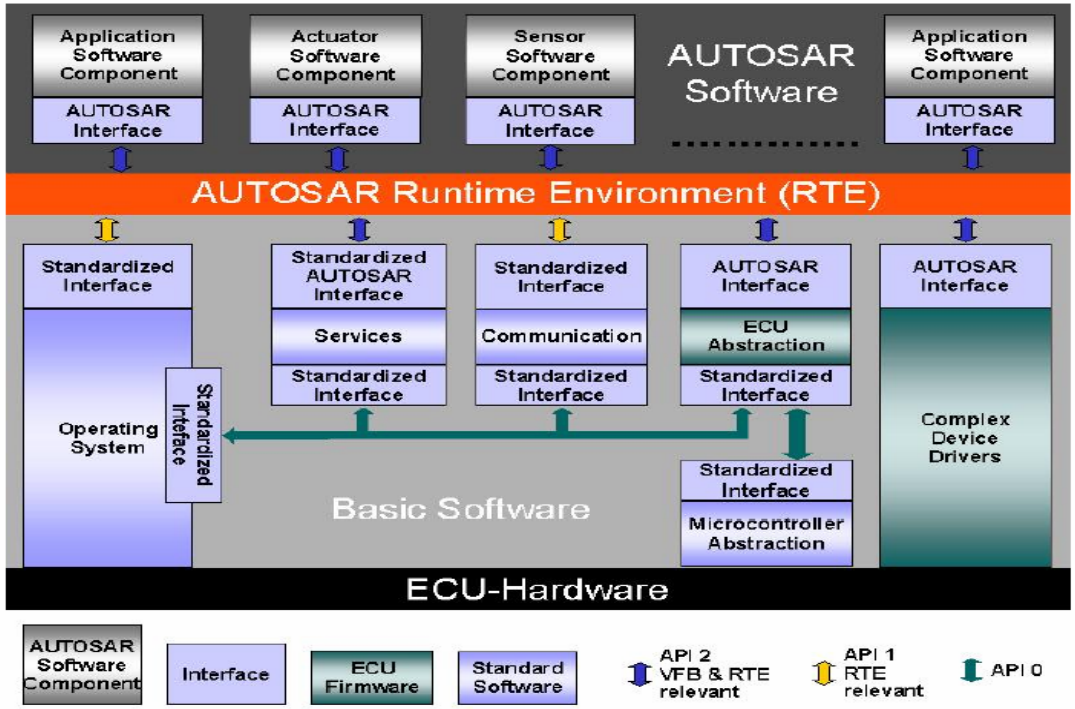
AUTOSAR Software Platform Standardization

DATE 2005
BMW Car-IT
09.03.2005
Harald Heinecke
F

Standardization

AUTOSAR

AUTOSAR – ECU Software Architecture



Automotive Open System Architecture (AUTOSAR):

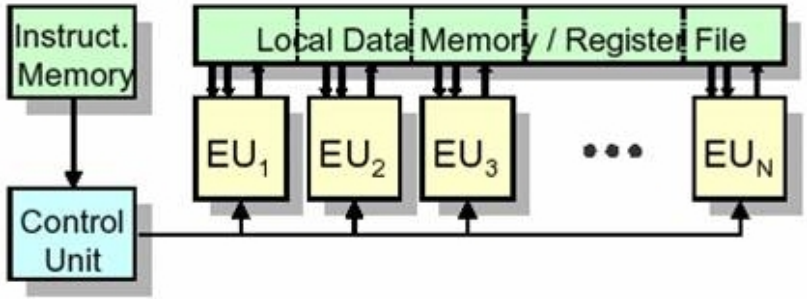
- Standardized, openly disclosed interfaces
- HW independent SW layer
- Transferability of functions
- Redundancy activation

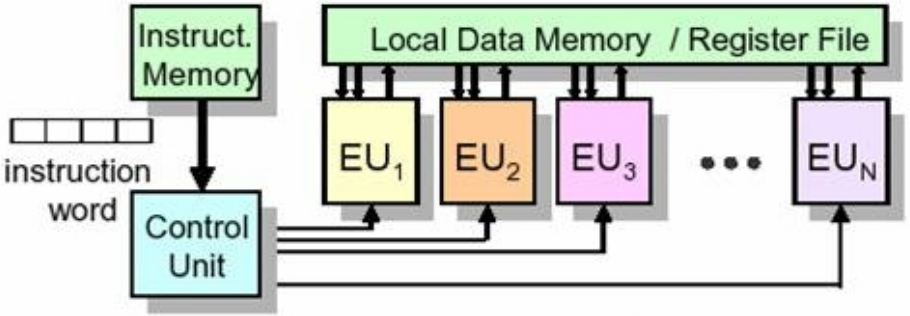
AUTOSAR RTE:
by specifying interfaces and their communication mechanisms, the applications are decoupled from the underlying HW and Basic SW, enabling the realization of Standard Library Functions.

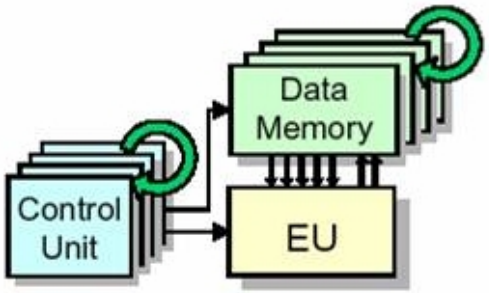
16 Thursday, 24 February 2005

The Different Levels of Parallelism to Exploit

- Data Level Parallelism
  - Inherent in most BB alg.'s
  - SIMD Architecture
    - High Efficiency
    - Compiler issues
    - Amdahl's Law !
- Instruction Level Parallelism
  - VLIW Architecture
    - Compiler friendly
    - Register file issues
  - Complex Instructions
- Task Level Parallelism
  - Multiple Interleaved Threads
    - Relaxed Memory Requirements
    - Increased Latency
  - Multiple Processors Core
    (ASIPs, Coprocessors, Accelerators)





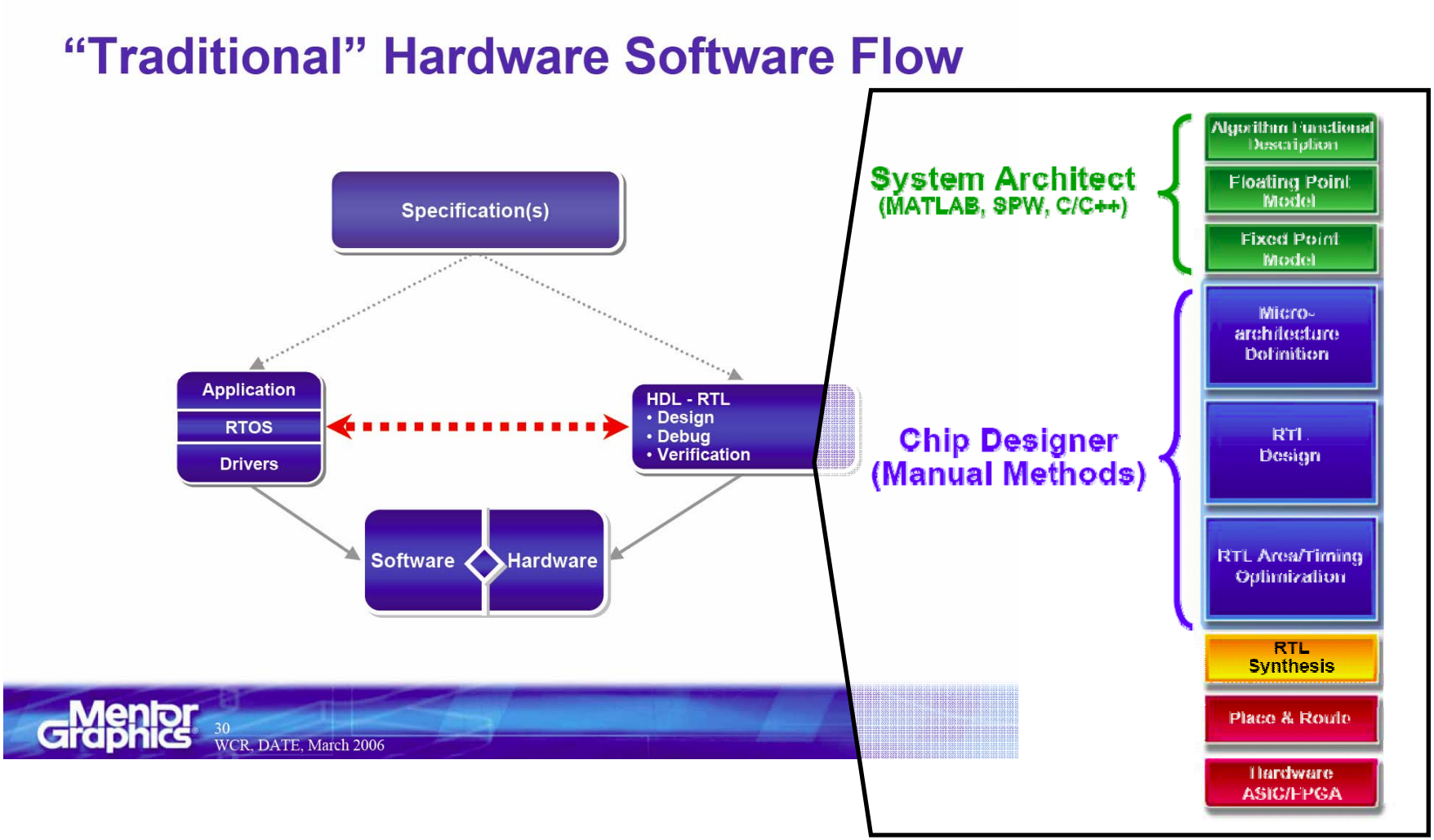


Best Results By Exploiting All Levels Of Parallelism !

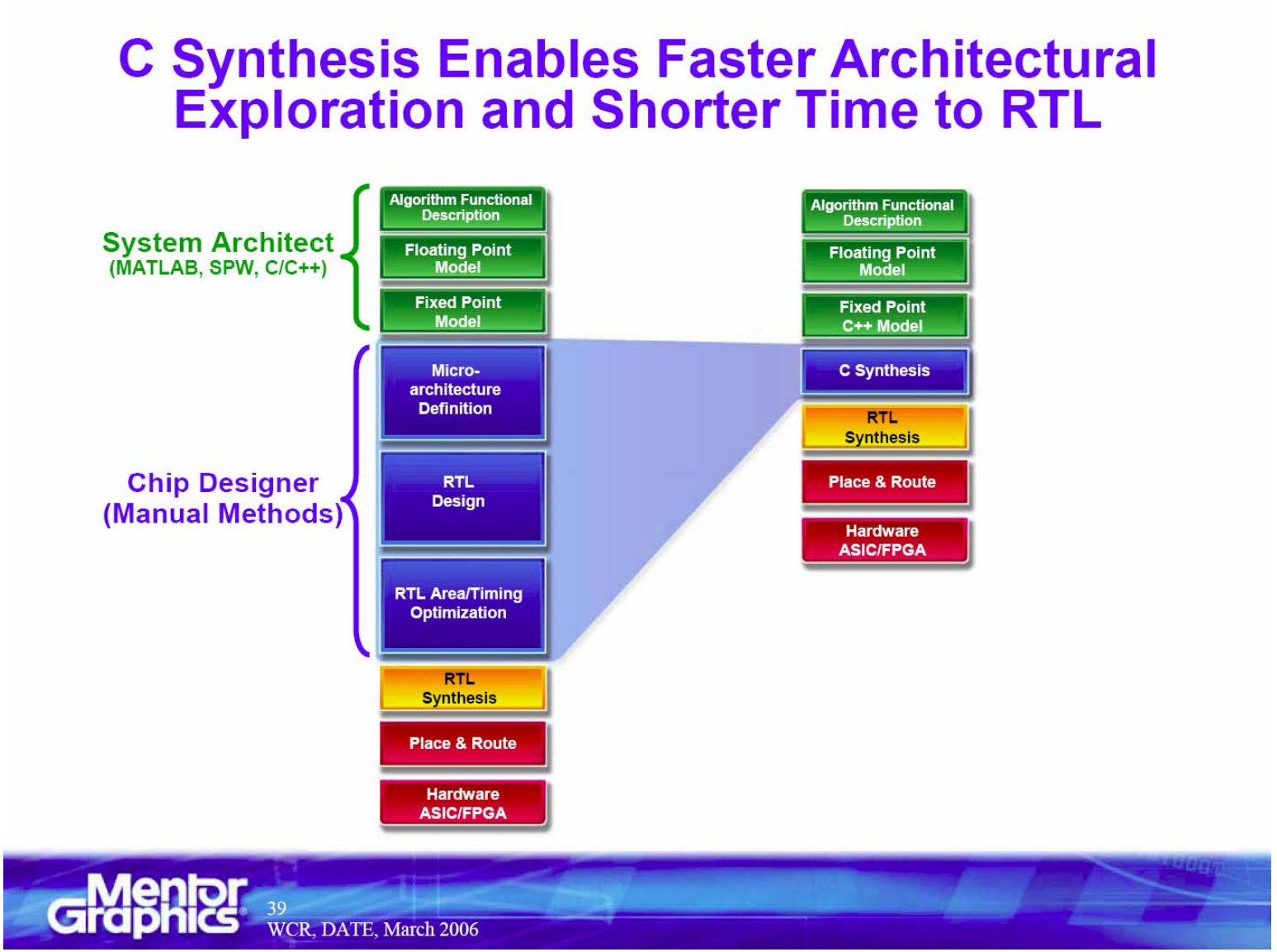
Hardware Design Flow

- Not a unified approach to explore the different levels of parallelism
- The macro level architecture must be selected
  - Implementing function in RTL (SystemC/C++ Flow)
    - Hardware implementation of RTOS
  - Partition the function and implements some parts using a dedicated Co-Processor
  - Change Core Instruction Set Application (ISA):
    - Parameterization of a configurable processor
    - Custom extension of the ISA
    - Define a new ISA (e.g. VLIW)

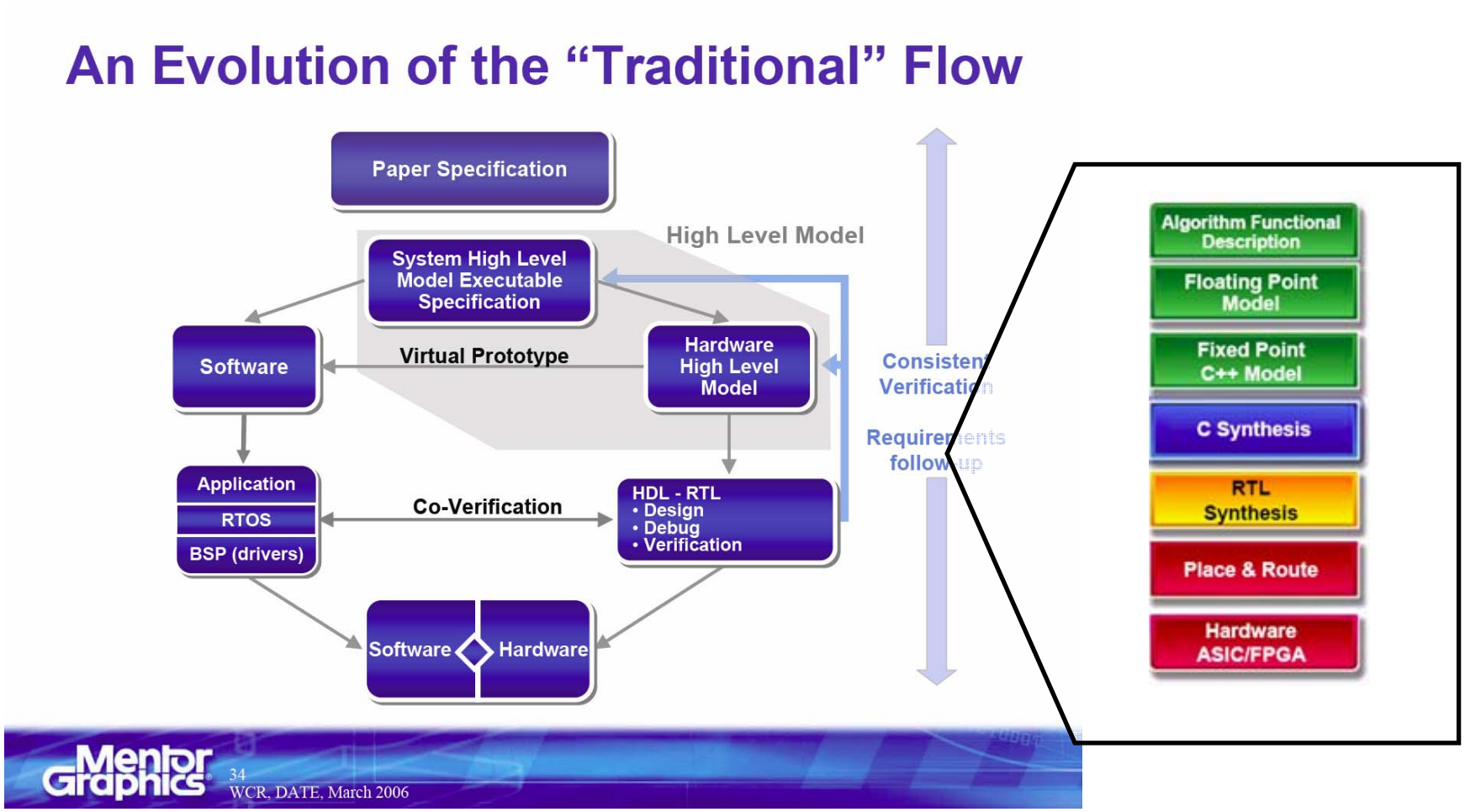
Traditional System-On-Chip Design Flow



C/C++ Synthesis Flow

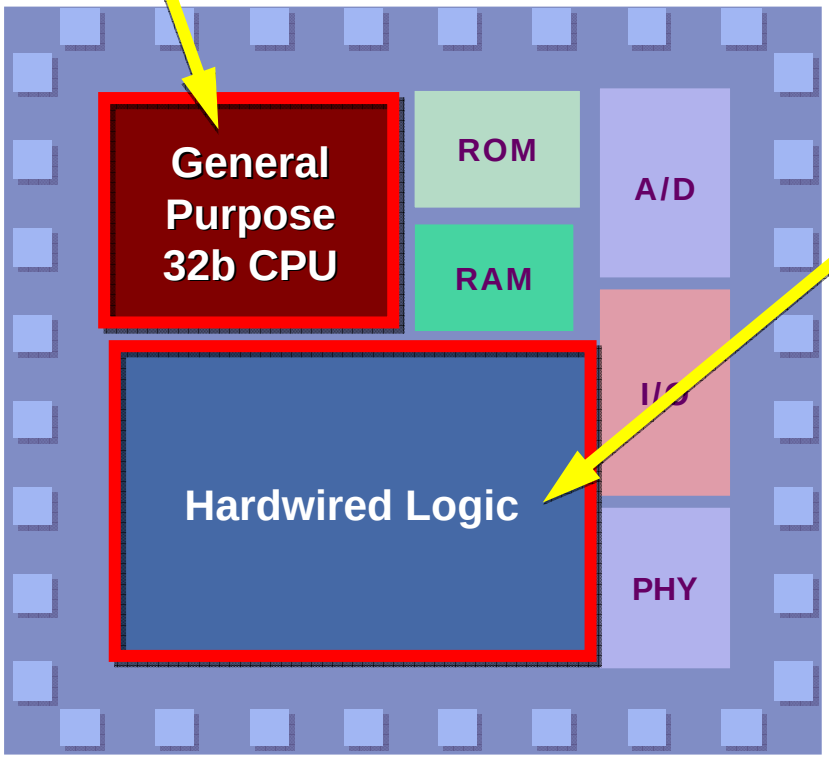


Evolution of System-On-Chip Design Flow



Implementing Function in RTL

General-purpose CPUs used in traditional SOCs are not fast enough for data-intensive applications, don't have enough I/O or compute bandwidth, lacks efficiency



Hardwired Logic

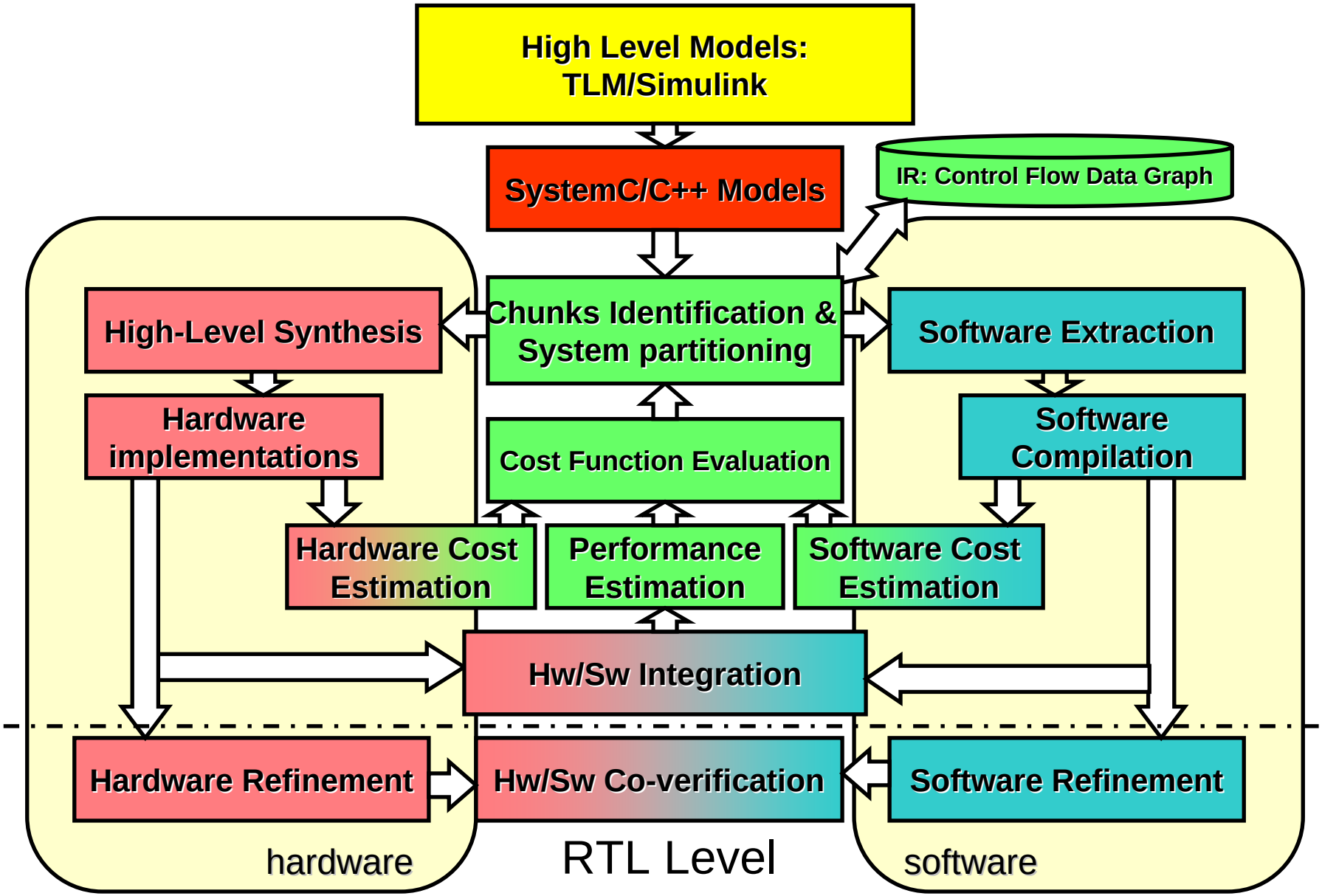
- High performance due to parallelism
- Large number of wires in/out of the block
- Languages/Tools familiar to many

But …

- Slow to design and verify
- Inflexible after tapeout
- High re-spin risk and cost
- Slows time to market

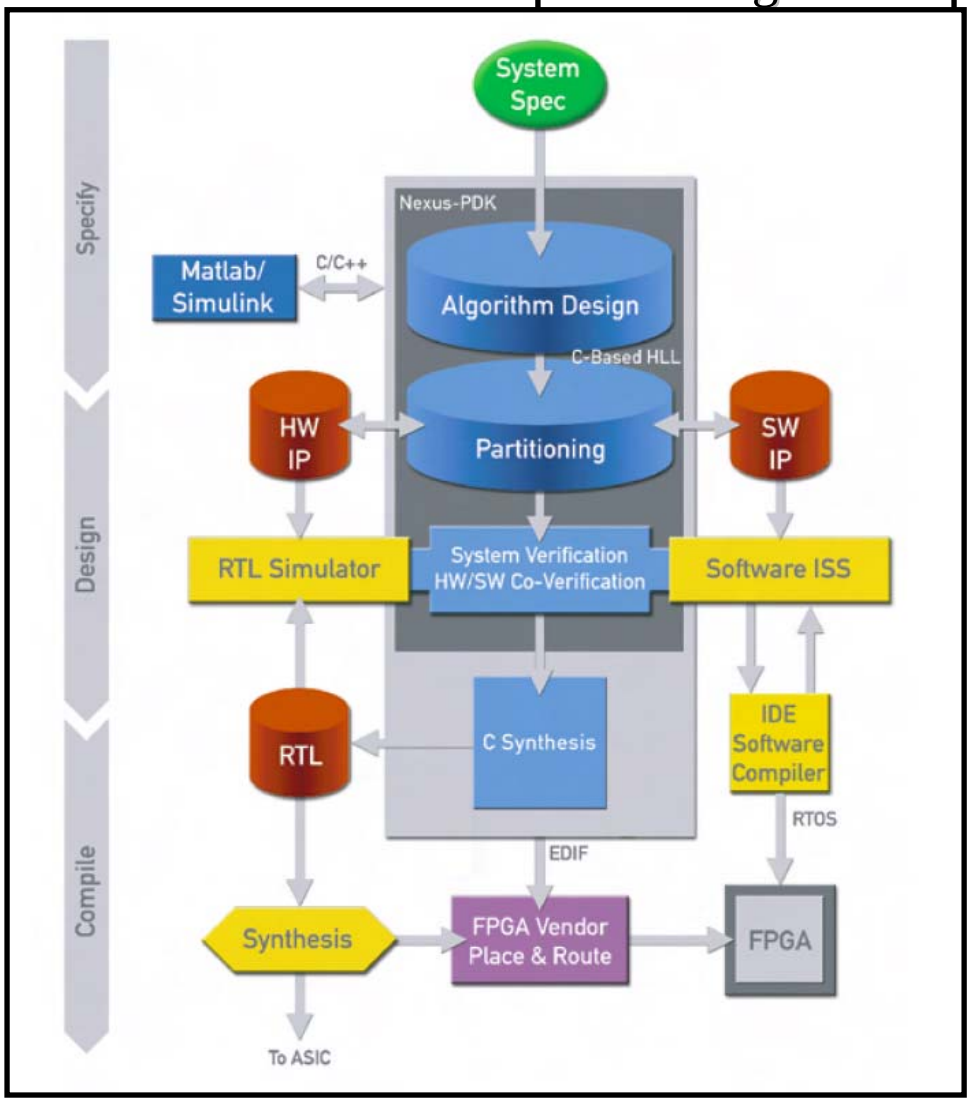
Courtesy of Grant Martin, Chief Scientist, Tensilica

SystemC/C++ Synthesis Flow

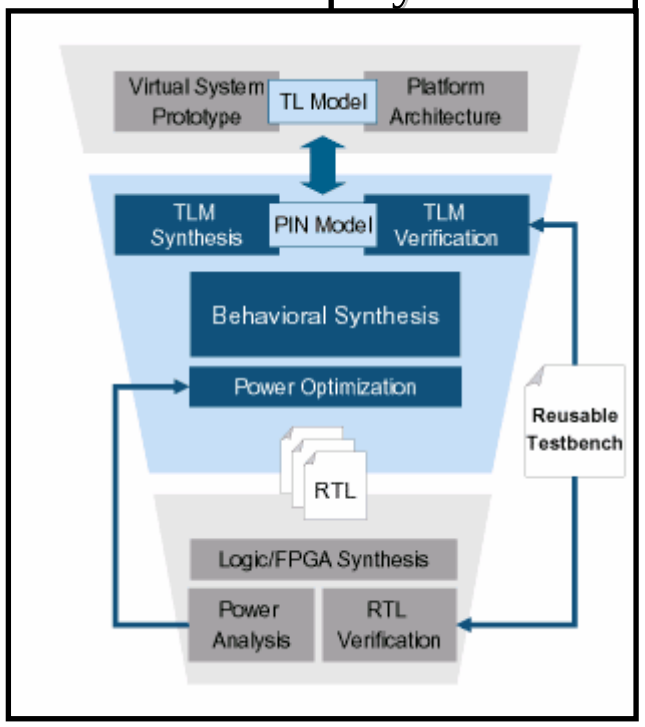


Celoxica and Forte Flows

DK Design Suite



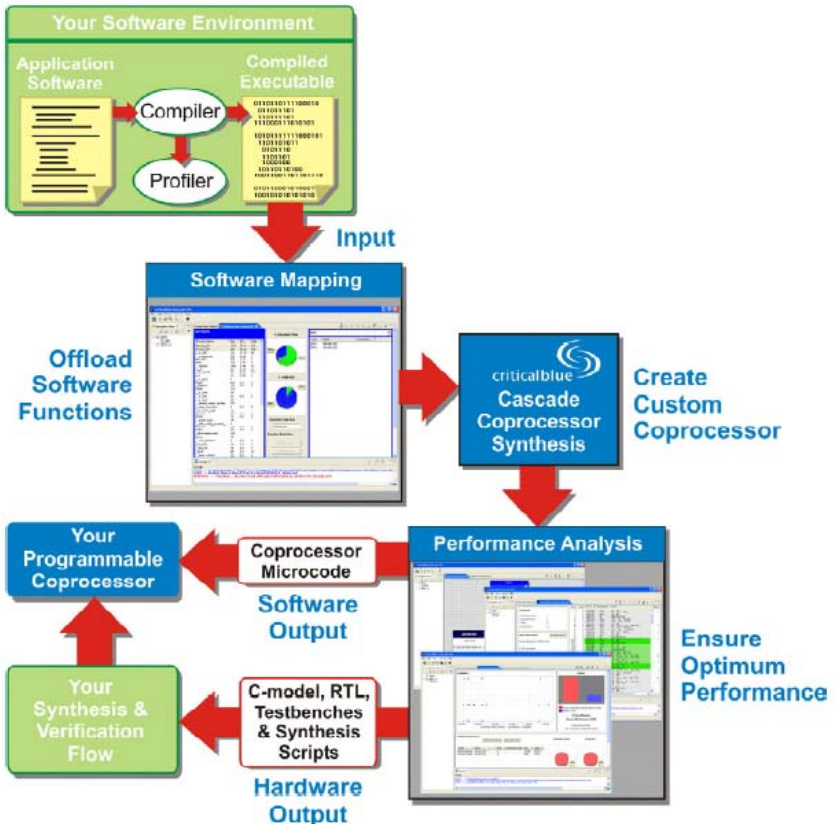
Cynthesizer



Coprocessor Synthesis

◆ Loosely coupled coprocessor that accelerates the execution of compiled binary executable software code offloaded from the CPU

- Delivers the parallel processing resources of a custom processor.
- Automatically synthesizes programmable coprocessor from software executable (hw and sw).
- Maximizes system performance through memory access and bus communication optimizations.



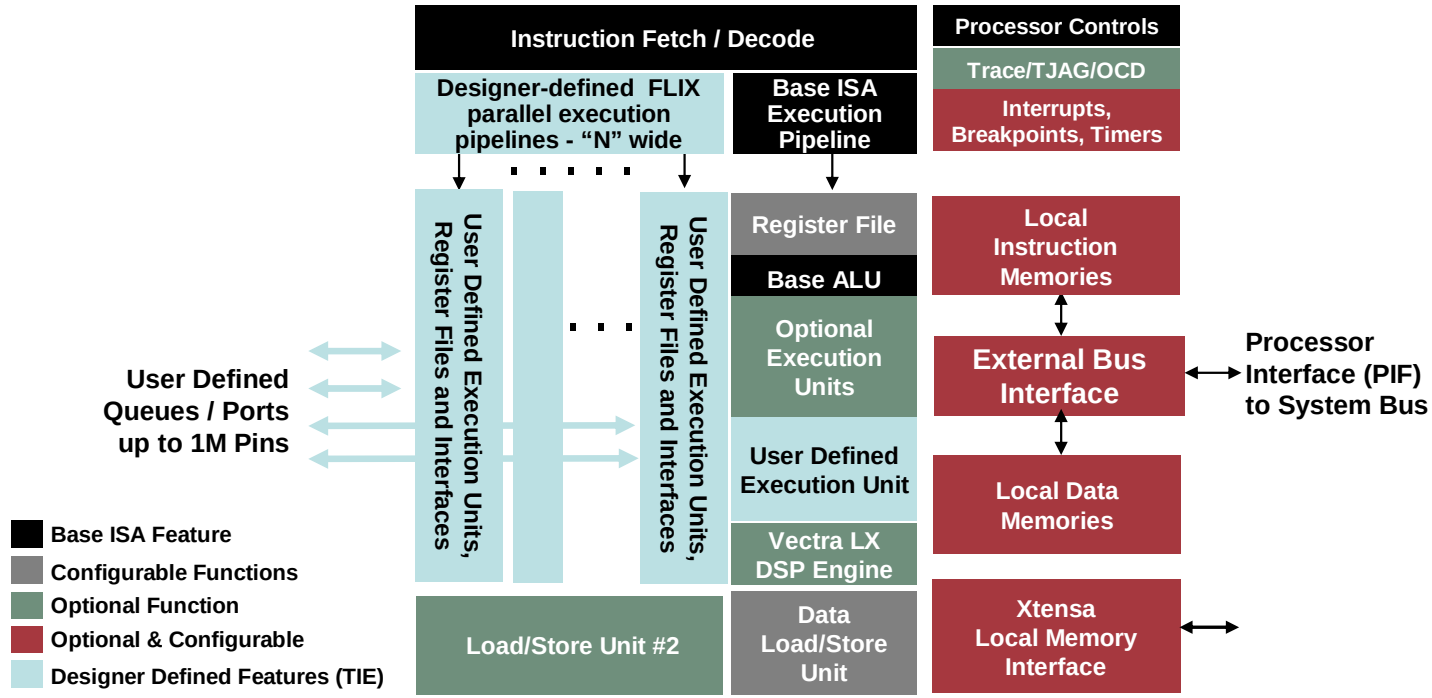
Criticalblue Approach

- Bottleneck Identification:
  - Analyze the profiling results of the application software running on the main microprocessor.
  - Manually identifies the specific tasks to be migrated to the coprocessor.
- Architecture Synthesis and Performance Estimation:
  - User-defined constraints like gate count, clock cycle count, and bus utilization
  - Analysis of the instruction code and architecte the coprocessor deploy the maximum parallelism consistent with the input constraints.
  - Estimation of gate-count and performance including estimates of communication overhead with the main processor.
- Coprocessor-Performance and “What-If” Analysis:
  - Generation of an instruction- and bit-accurate C model of the coprocessor architecture used in conjunction with the main processor’s instruction-set simulator (ISS).
  - Typical analysis: performance profiling, memory-access activity, and activation trace data
  - The model also is used to validate the coprocessor within a standard C or SystemC simulation environment.
- Hardware Synthesis and Microcode generation:
  - Generation of the coprocessor hardware, delivering synthesizable RTL code in either VHDL or Verilog and of the circuitry that’s needed to enable the coprocessor to communicate with the main processor’s bus interface.
  - Generation of the coprocessor microcode.
  - It automatically modifies the original executable code so that function calls are directed to a communications library.
  - This library manages the coprocessor handoff. It also communicates parameters and results between the main processor and the coprocessor.
  - Microcode can be generated independently of the coprocessor hardware, allowing new microcode to be targeted at an existing coprocessor design.

Configurable and Extensible Processor

Courtesy of Grant Martin, Chief Scientist, Tensilica

tensilica Fully Configurable Processor Features



Instruction Extension : Simple Example

Courtesy of Grant Martin, Chief Scientist, Tensilica

1 2

```
operation TRUNCATE_16 {out AR z, in AR m}{}
{
assign z = {16'b0, m[23:8] };
}
```

3

The ***operation*** statement describes an entire new instruction, including:

1. Instruction name
2. Instruction format and arguments
3. Functional Behavior

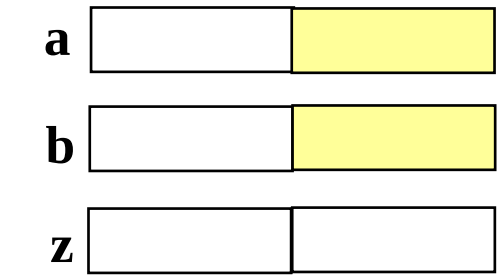
From this single statement, Tensilica's technology generates processor hardware, simulation and software development tool support for the new instruction.

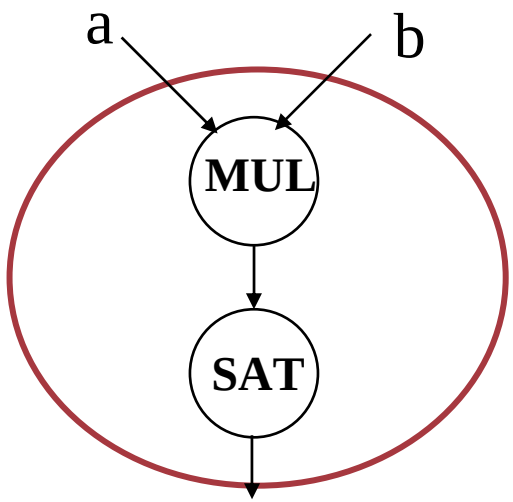
More Complex Extensions

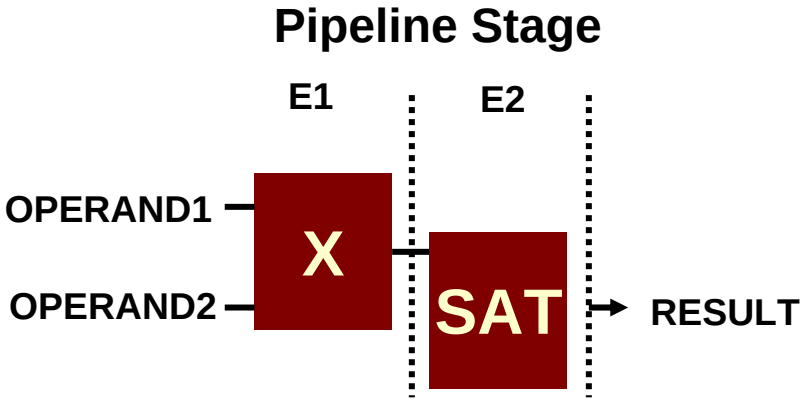
Courtesy of Grant Martin, Chief Scientist, Tensilica

```
operation MUL_SAT_16 {out AR z, in AR a, in AR b} {}
{
wire [31:0] m = TIEmul(a[15:0],b[15:0],1);

assign z = {16'b0,
            m[31] ? ((m[31:23]==9'b1) ? m[23:8] : 16'h8000)
                  : ((m[31:23]==9'b0) ? m[23:8] : 16'h7fff) };
}
schedule ms {MUL_SAT_16} {def z 2;}
```





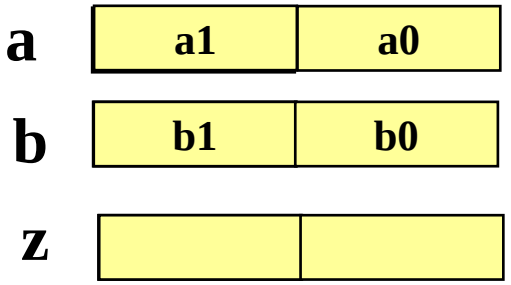


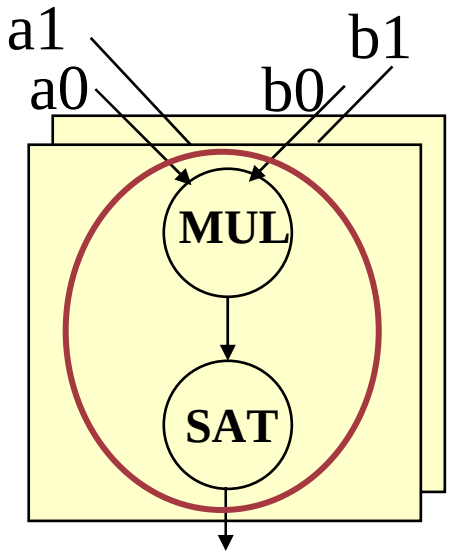
SIMD : Exploiting Data Parallelism

Courtesy of Grant Martin, Chief Scientist, Tensilica

```
operation MUL_SAT_2x16 {out AR z, in AR a, in AR b} {}
{
  wire [31:0] m1 = TIEmul(a[31:16],b[31:16],1);
  wire [31:0] m0 = TIEmul(a[15:0], b[15:0], 1);
  assign z = {m1[31] ? ((m1[31:23]==9'b1) ? m1[23:8] : 16'h8000)
                     : ((m1[31:23]==9'b0) ? m1[23:8] : 16'h7fff),
              m0[31] ? ((m0[31:23]==9'b1) ? m0[23:8] : 16'h8000)
                     : ((m0[31:23]==9'b0) ? m0[23:8] : 16'h7fff) };
}
schedule ms {MUL_SAT_2x16} {def z 2;}
```

Core 32bit Register File (AR)



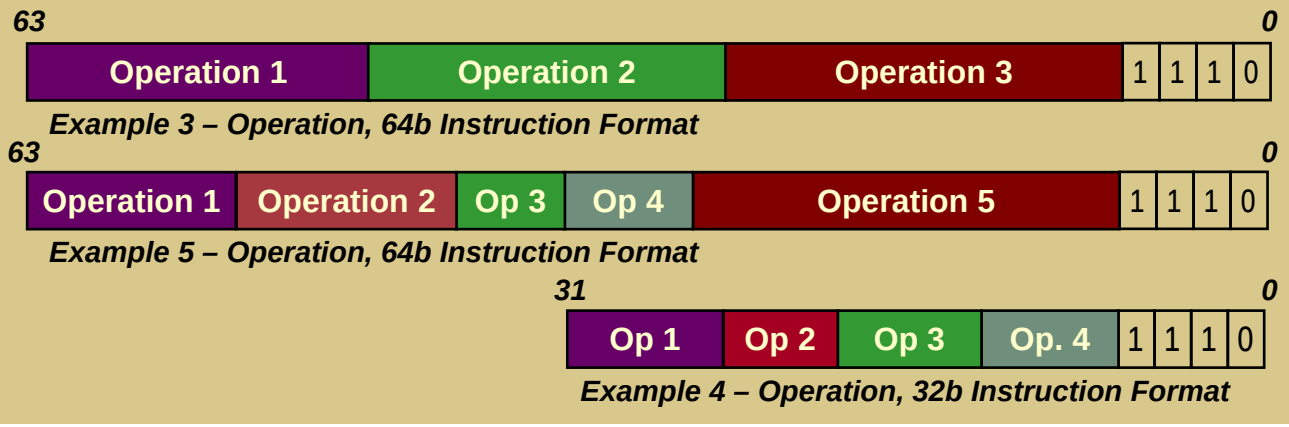


Multiple Instruction Issues - FLIX™ Architecture

Courtesy of Grant Martin, Chief Scientist, Tensilica

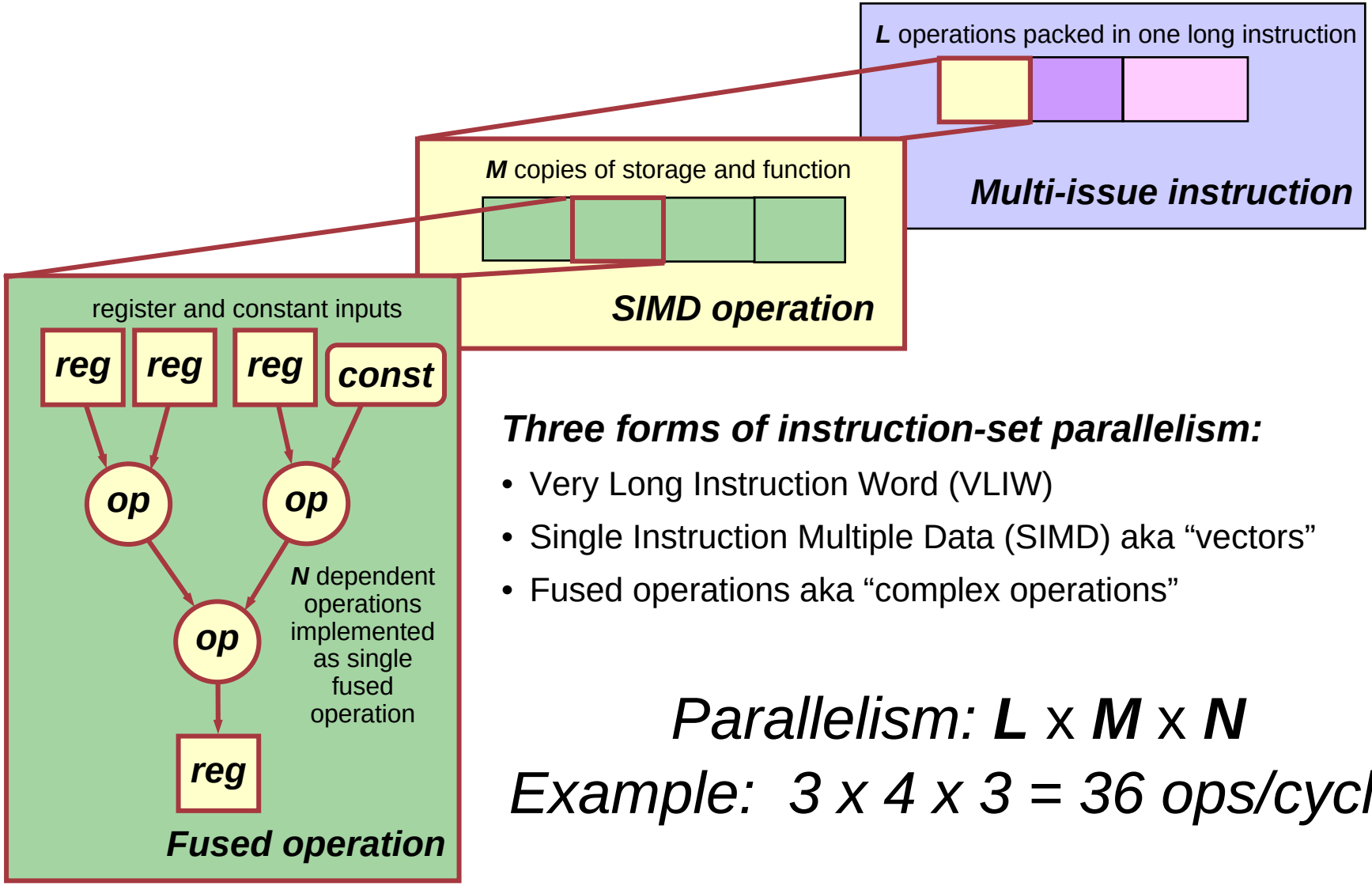
- FLIX™ – **F**lexible **L**ength **I**nstruction **X**tensions
- Multiple, concurrent, independent, compound operations per instruction
  - Modeless intermixing of 16, 24, and 32 or 64 bit instructions
  - Fast and concurrent code (concurrent execution) when needed
  - Compact code when concurrency / parallelism isn't needed
  - Full code compatibility with base 16/24 bit Xtensa ISA
- Minimal overhead
  - No VLIW-style code-bloat
  - ~2000 gates added control logic

Designer-Defined FLIX Instruction Formats with Designer-Defined Number of Operations



Parallelism at Three Levels in Extensible Instructions

Courtesy of Grant Martin, Chief Scientist, Tensilica



Three forms of instruction-set parallelism:

- Very Long Instruction Word (VLIW)
- Single Instruction Multiple Data (SIMD) aka “vectors”
- Fused operations aka “complex operations”

*Parallelism: **L** x **M** x **N***

Example: 3 x 4 x 3 = 36 ops/cycle

HW & SW automatically generated

Courtesy of Grant Martin, Chief Scientist, Tensilica

Xtensa Xplorer

- Integrated Development Environment
- TIE Development tools
- C Development tools
- Profiling & visualization tools

Hardware

- Synthesizable RTL
- Synopsys/Cadence flows

Software

- Scheduling assembler
- Xtensa C/C++ Compiler: vectorizing C/C++ compiler
- Xtensa Instruction Set Simulator – Pipeline accurate
- Debuggers
- XTMP: System Modeling API
- Bus Functional Model for HW/SW co-simulation model
- RTOS: VxWorks, Nucleus, XTOS

Design Flow

Courtesy of Grant Martin, Chief Scientist, Tensilica

Automation: Optimized Processor & Matching Software Tools

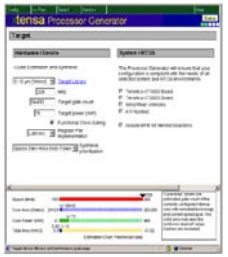
ANSI C/C++ Code

Source code

```
int main()
{
 int i;
 short c[100];
 for (i=0;i<N/2;i++)
 {
```

XPRES ***Compiler***

Optional Step
Runs in Minutes



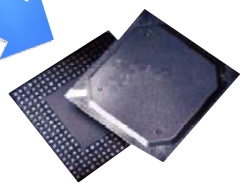
Electronic Specification
Configuration selection and custom-instruction description

Xtensa ***Processor Generator ****

Complete Hardware Design
Source RTL, EDA scripts, test suite

Processor Extensions

Customized Software Tools
C/C++ compiler
Debuggers Simulators
RTOSes



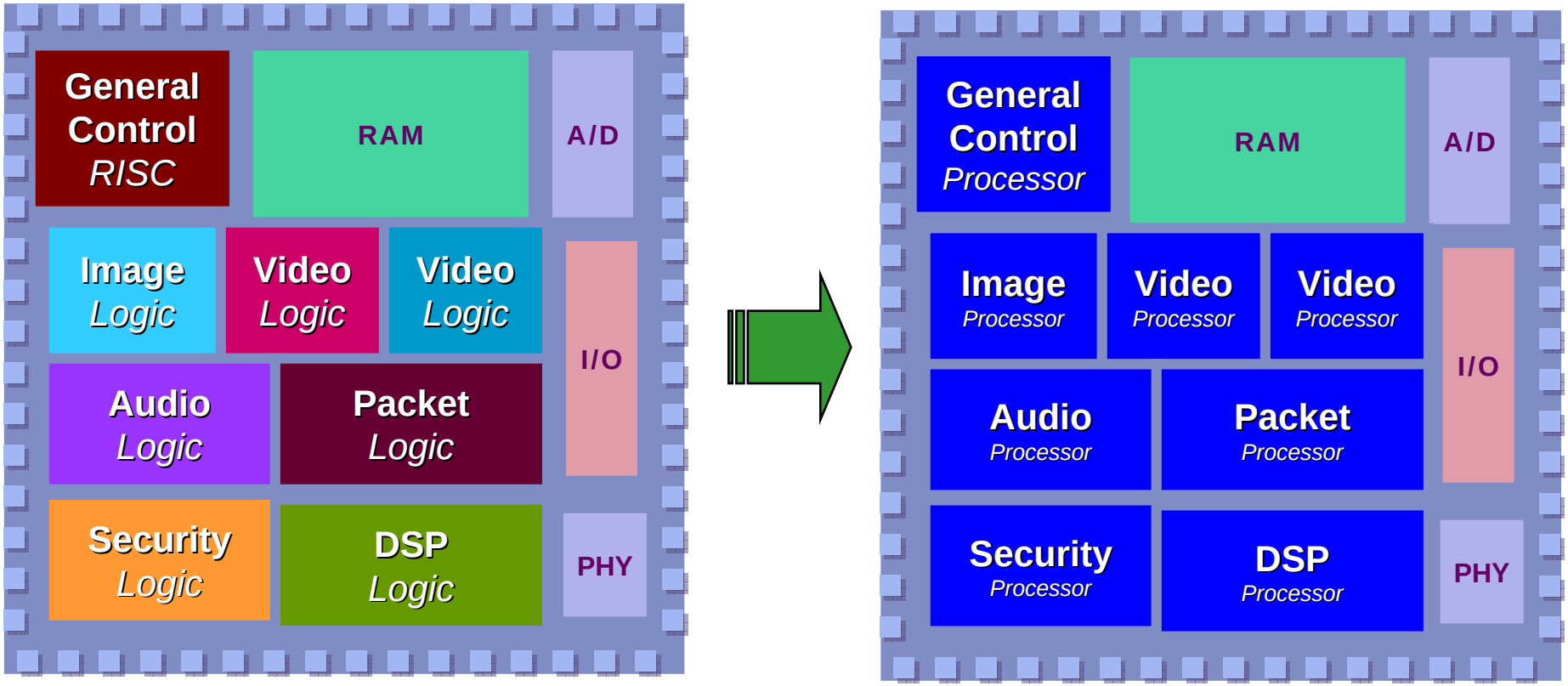
Use standard ASIC/COT design techniques and libraries for *any* IC fabrication process

Iterate in hours

** US Patent: 6,477,697*

Designing with many processors

Courtesy of Grant Martin, Chief Scientist, Tensilica

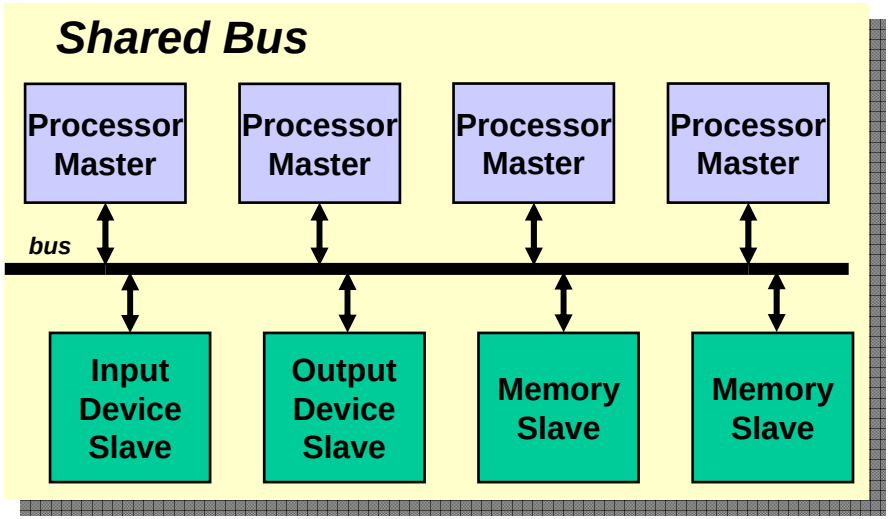


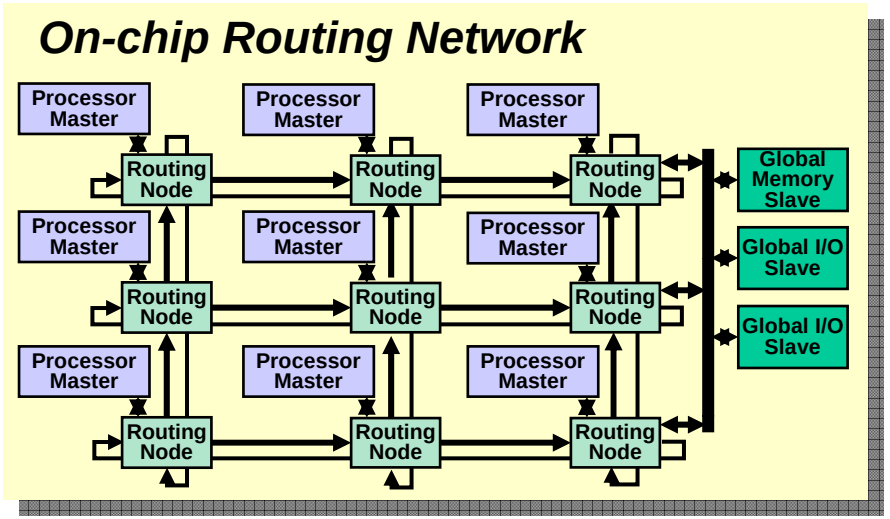
System-On-Chip (SOC)

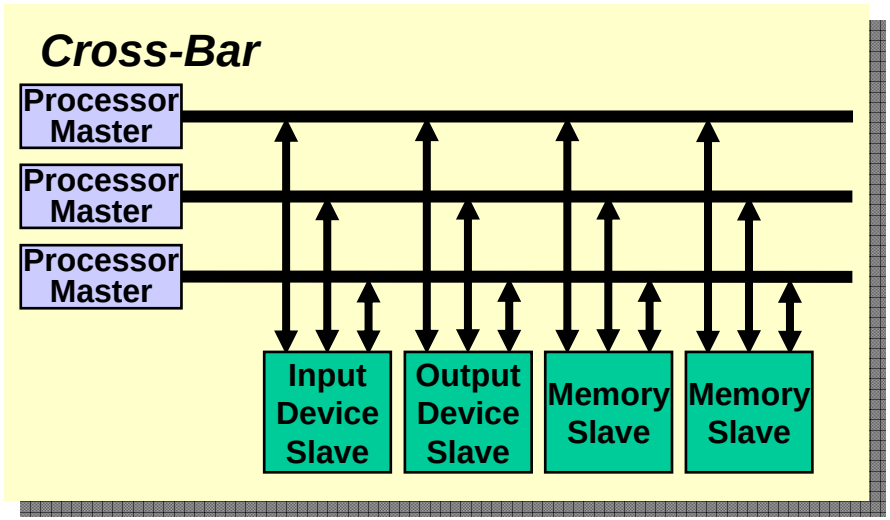
Advanced System-On-Chip (SOC)

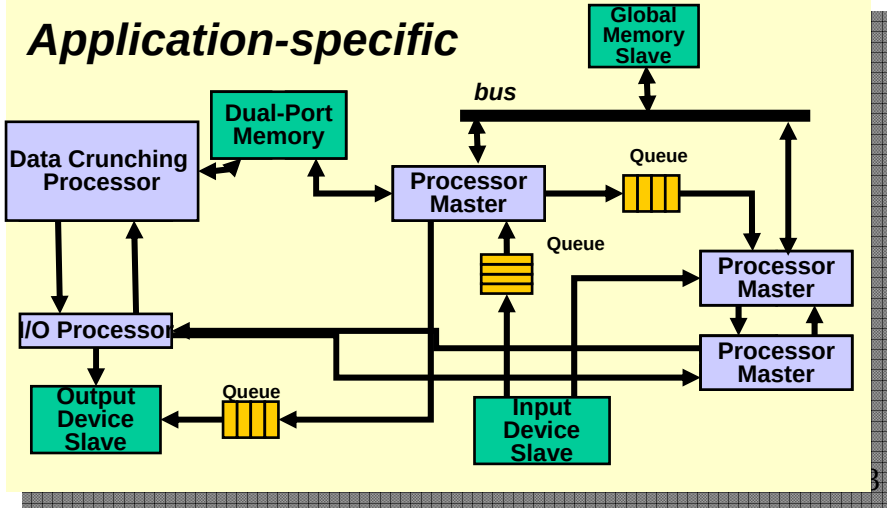
Exploiting MP: Many Possible Architectures

Courtesy of Grant Martin, Chief Scientist, Tensilica



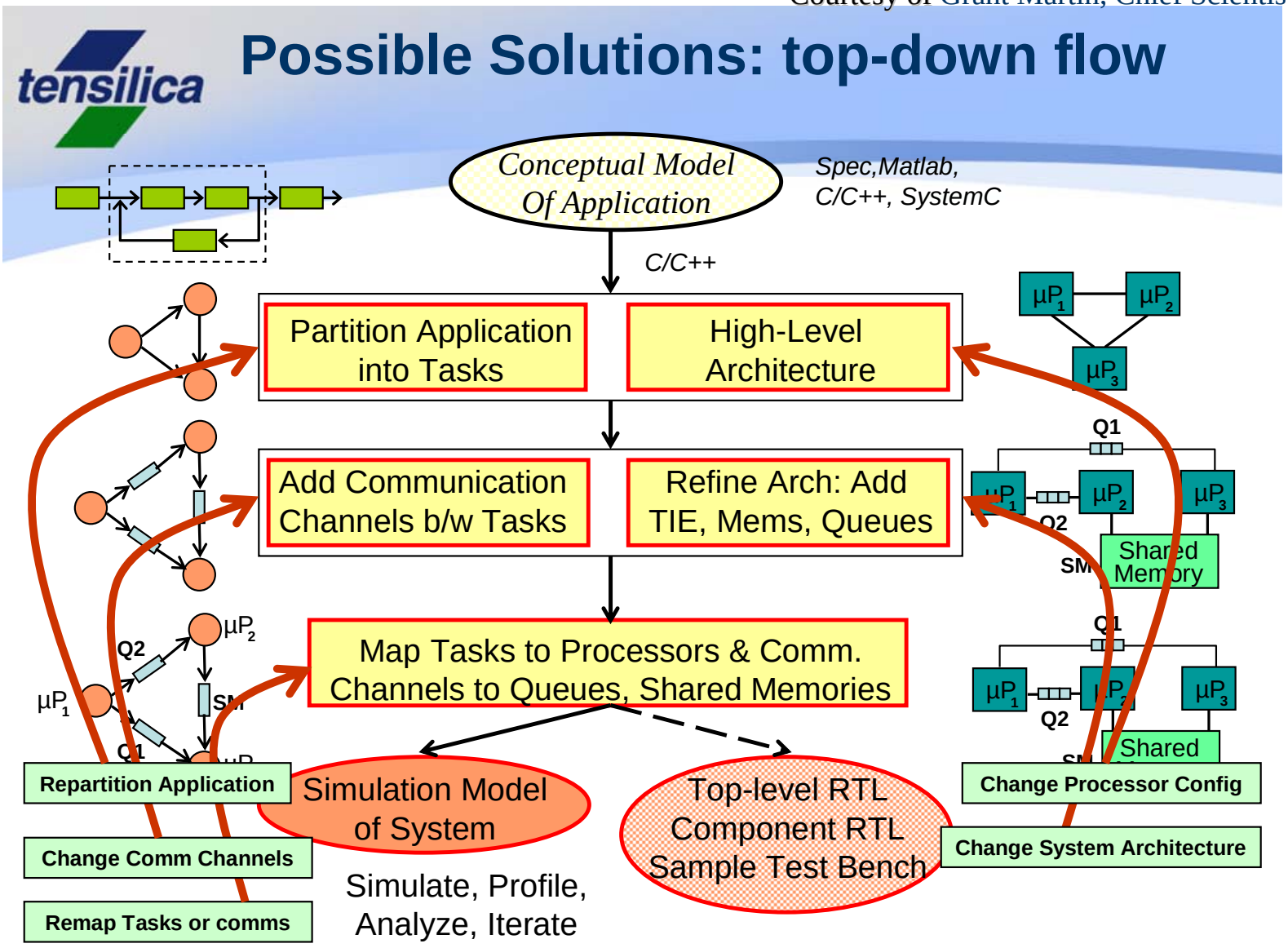






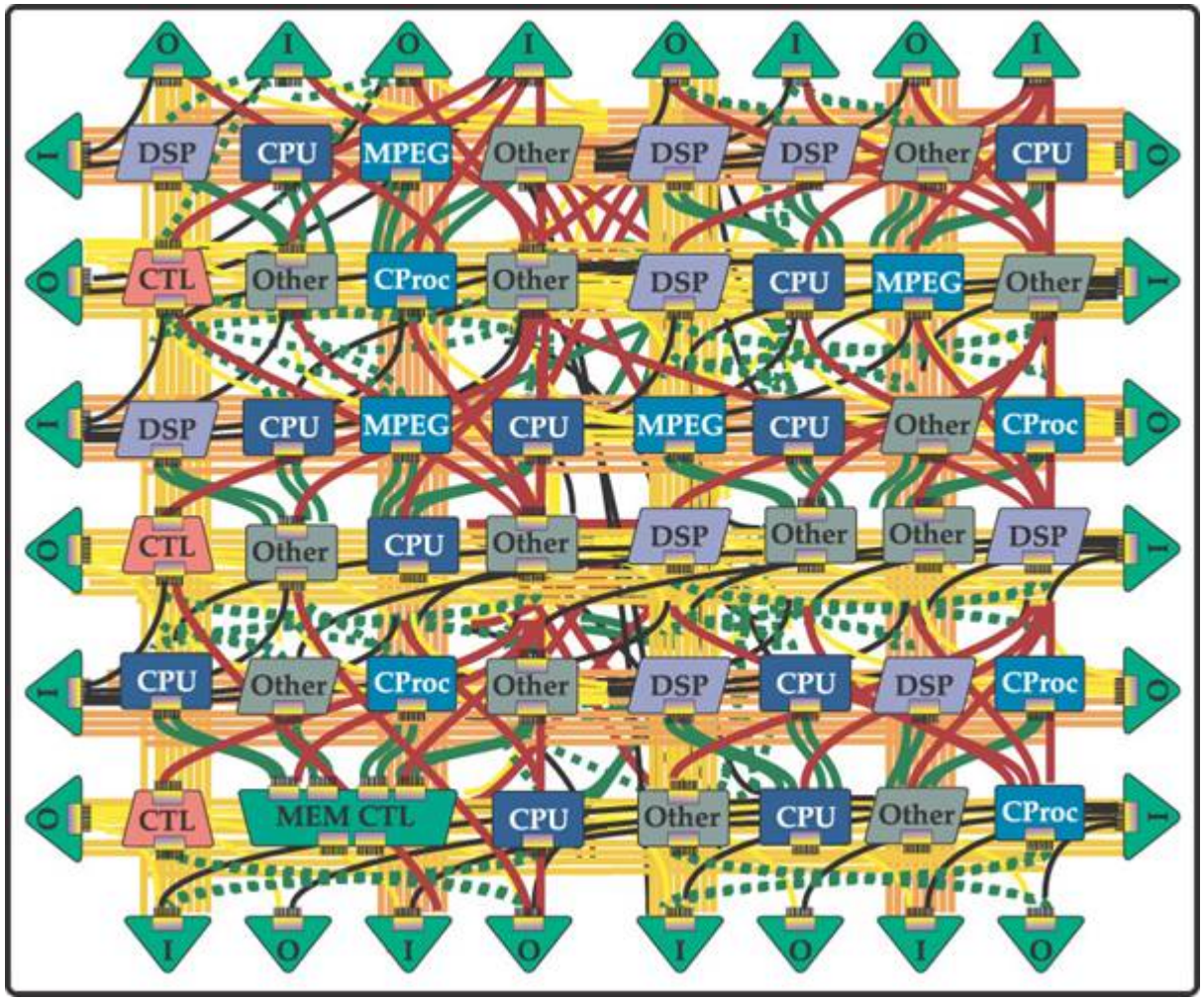
Multiprocessor Design Flow

Courtesy of Grant Martin, Chief Scientist, Tensilica



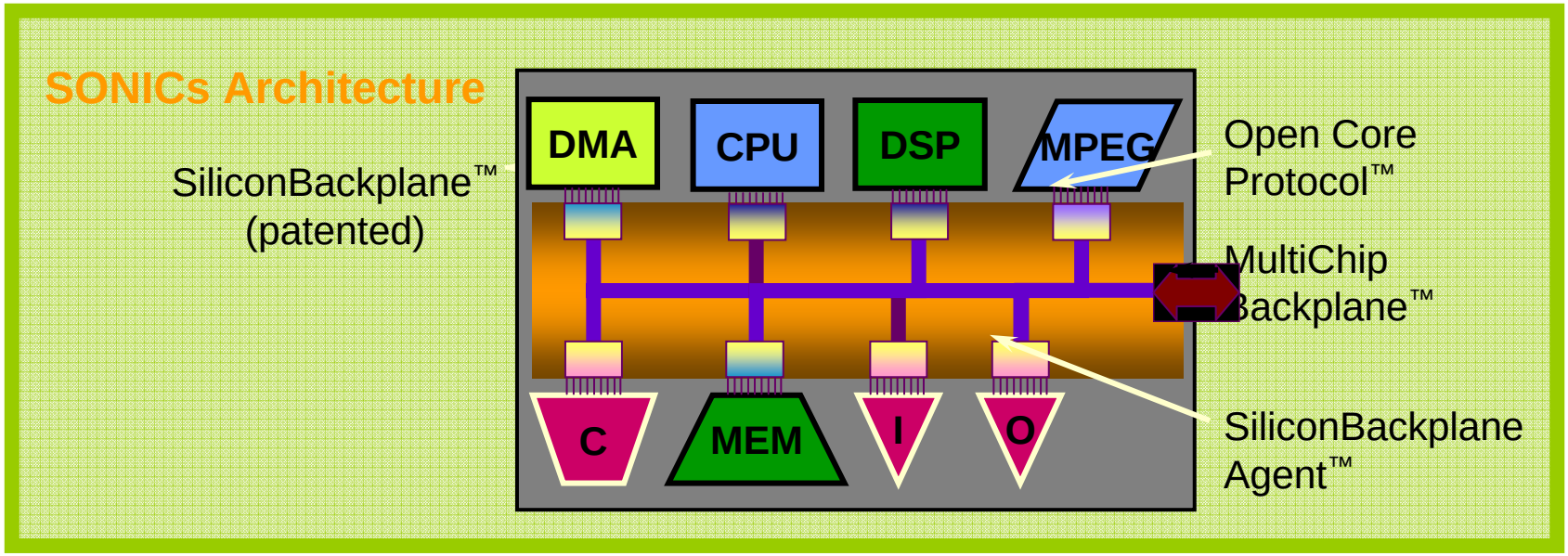
From unstructured connectivity to a …

Courtesy of SONICS



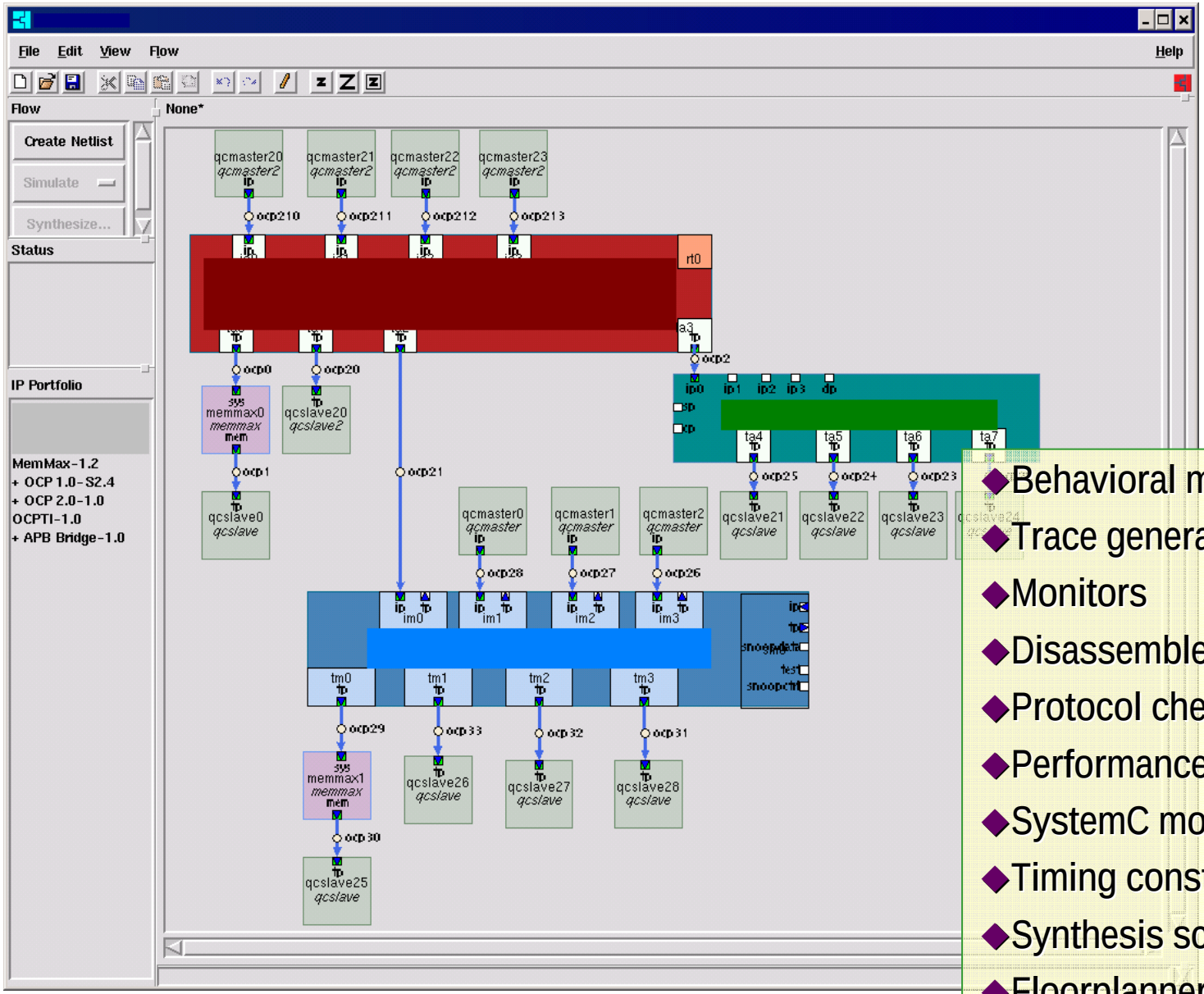
Communication Centric Design Flow

- "Communication Centric Platform"
  - SONIC, Palmchip
  - Concentrates on communication
    - Delivers communication framework plus peripherals
    - Limits the modeling efforts



SONICS Automated flow

Courtesy of SONICS



- Behavioral models
- Trace generation
- Monitors
- Disassemblers
- Protocol checkers
- Performance analysis
- SystemC models
- Timing constraint propagation
- Synthesis script generation
- Floorplanner interface

Outline

- Embedded System Applications
- Platform based design methodology
- Electronic System Level Design
  - Functions: MoC, Languages
  - Architectures: Network, Node, SoC
- Metropolis
- Conclusions

Metropolis: an Environment for System-Level Design

- Motivation
  - Design complexity and the need for verification and time-to-market constraints are increasing
  - Semantic link between specification and implementation is necessary
- Platform-Based Design
  - Meet-in-the-middle approach
  - Separation of concerns
    - Function vs. architecture
    - Capability vs. performance
    - Computation vs. communication
- Metropolis Framework
  - Extensible framework providing simulation, verification, and synthesis capabilities
  - Easily extract relevant design information and interface to external tools
- Released Sept. 15th, 2004

Metropolis: Target and Goals

- Target: Embedded System Design
  - Set-top boxes, cellular phones, automotive controllers, …
  - **Heterogeneity:**
    - computation: Analog, ASICs, programmable logic, DSPs, ASIPs, processors
    - communication: Buses, cross-bars, cache, DMAs, SDRAM, …
    - coordination: Synchronous, Asynchronous (event driven, time driven)
- Goals:
  - **Design methodologies:**
    - abstraction levels: design capture, mathematics for the semantics
    - design tasks: cache size, address map, SW code generation, RTL generation, …
  - **Tool set:**
    - synthesis: data transfer scheduling, memory sizing, interface logic, SW/HW generation, …
    - verification: property checking, static analysis of performance, equivalence checking, …

Metropolis Project

Participants:

- UC Berkeley (USA): methodologies, modeling, formal methods
- CMU (USA): formal methods
- Politecnico di Torino (Italy): modeling, formal methods
- Universita Politecnica de Catalunya (Spain): modeling, formal methods
- Cadence Berkeley Labs (USA): methodologies, modeling, formal methods
- PARADES (Italy): methodologies, modeling, formal methods
- ST (France-Italy): methodologies, modeling
- Philips (Netherlands): methodologies (multi-media)
- Nokia (USA, Finland): methodologies (wireless communication)
- BWRC (USA): methodologies (wireless communication)
- Magneti-Marelli (Italy): methodologies (power train control)
- BMW (USA): methodologies (fault-tolerant automotive controls)
- Intel (USA): methodologies (microprocessors)
- Cypress (USA): methodologies (network processors, USB platforms)
- Honeywell (USA): methodologies (FADEC)

Metropolis Framework

Function Specification

Design Constraints & Assertions

Architecture (Platform) Specification

Metropolis Infrastructure

- Design methodology
- Meta model of computation
- Base tools
  - Design imports
  - Meta model compiler
  - Simulation

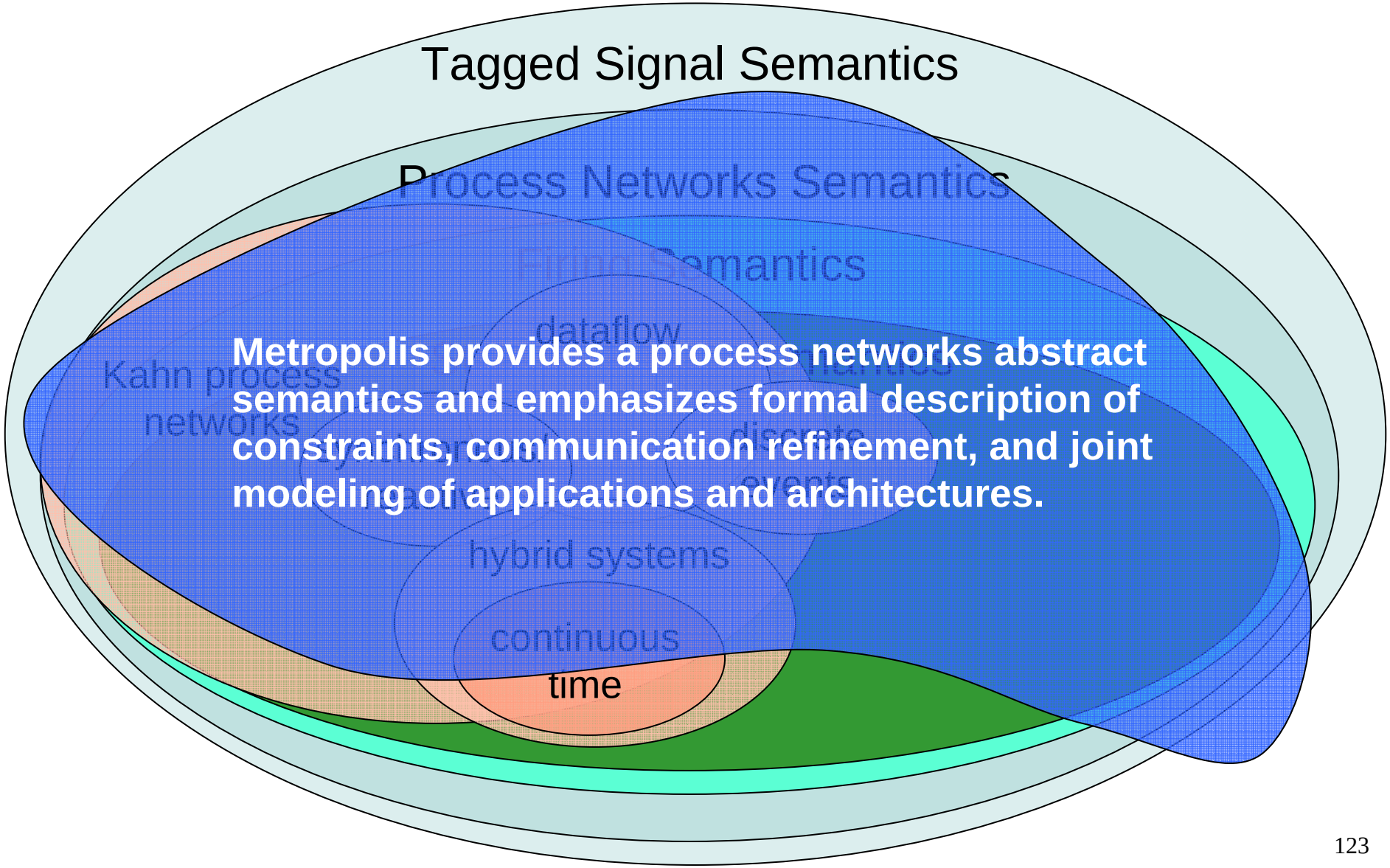
Synthesis/Refinement

- Compile-time scheduling of concurrency
- Communication-driven hardware synthesis
- Protocol interface generation

Analysis/Verification

- Static timing analysis of reactive systems
- Invariant analysis of sequential programs
- Refinement verification
- Formal verification of embedded software

Meta Frameworks: Metropolis



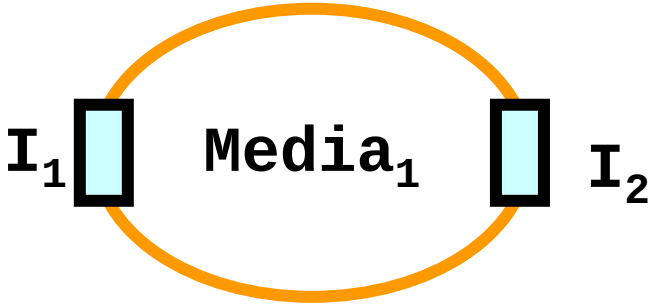
Metropolis Objects: adding quantity managers

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

• **Processes (Computation)**

Active Objects
Sequential Executing Thread

• **Media (Communication)**



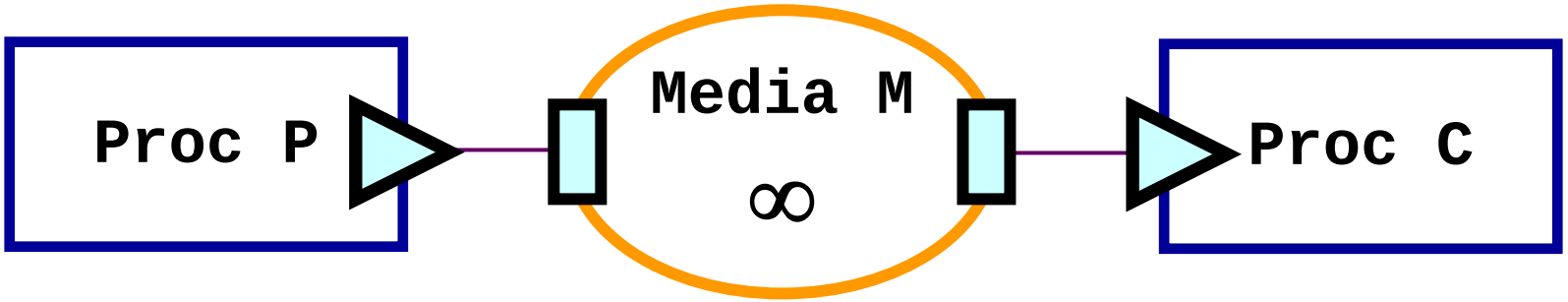
Passive Objects
Implement Interface Services

• **Quantity Managers (Coordination)**

Schedule access to
resources and quantities

A Producer–Consumer Example

- A process P producing integers
- A process C consuming integers
- A media M implementing the communication services

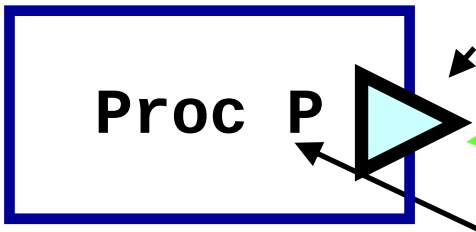


Writer: Process P (Producer)

- Writer.mmm: Port (interface) definition

```
package producers_consumer;
interface IntWriter extends Port{
    update void writeInt(int i);
    eval int nspace();
}
```

- P.mmm: Process behavior definition



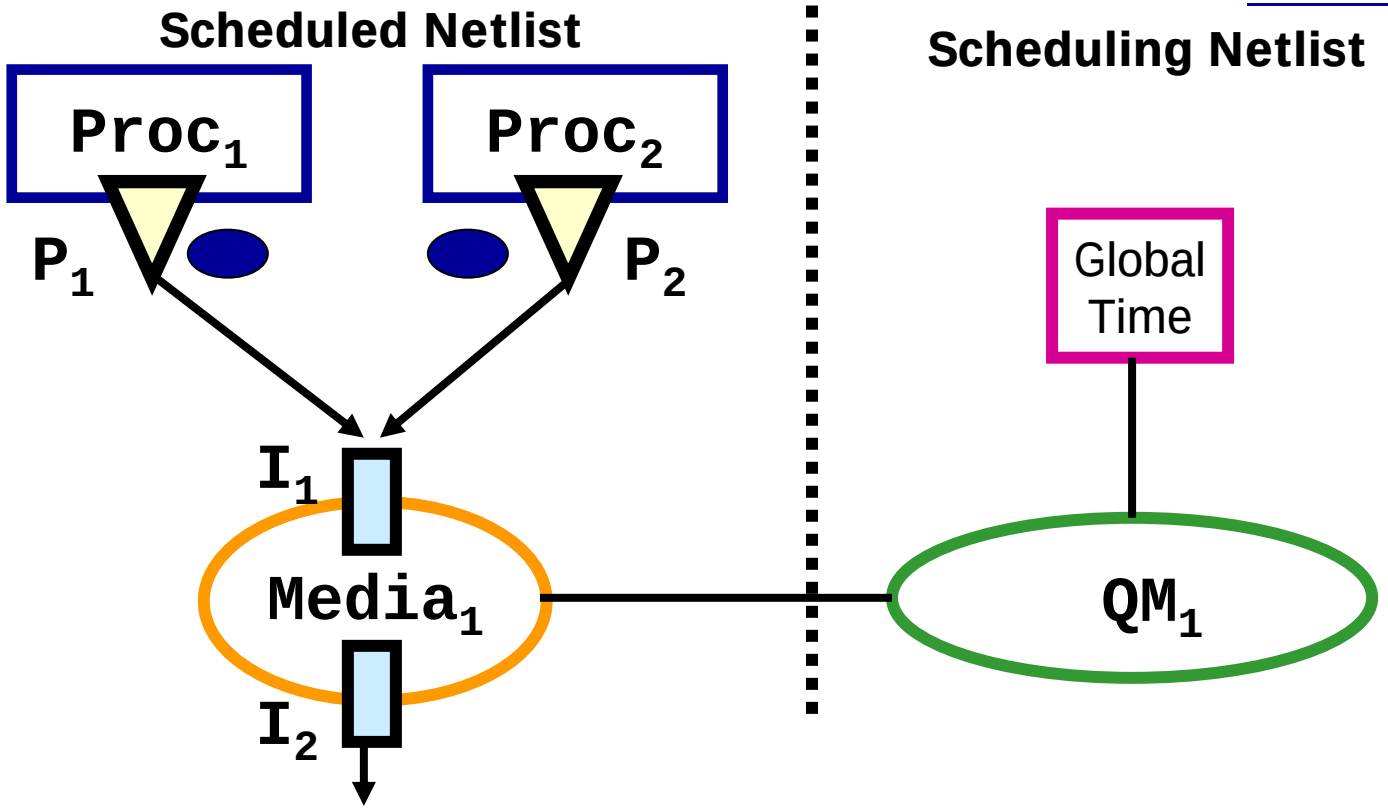

```
package producers_consumer;
process P {
    port IntWriter port_wr;
    public P(String name) {}
    void thread() {
        int w = 0;
        while (w < 30) {
            port_wr.writeInt(w);
            w = w + 1;
        }
}}
```

Metro. Netlists and Events

Metropolis Architectures are created via two netlists:

- Scheduled – generate **events**[1] for services in the scheduled netlist.
- Scheduling – allow these **events** access to the services and **annotate events** with **quantities**.

Related Work



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

Key Modeling Concepts

- An event is the fundamental concept in the framework
  - Represents a transition in the action automata of an object
  - An event is owned by the object that exports it
  - During simulation, generated events are termed as *event instances*
  - Events can be annotated with any number of quantities
  - Events can partially expose the state around them, constraints can then reference or influence this state
- A service corresponds to a set of sequences of events
  - All elements in the set have a common begin event and a common end event
  - A service may be parameterized with arguments

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

Action Automata

- Processes take *actions*.
  - statements and some expressions, e.g.

    y = z+port.f();, z+port.f(), port.f(), i < 10, …
  - only calls to media functions are *observable actions*

- An *execution* of a given netlist is a sequence of vectors of *events*.
  - *event* : the beginning of an action, e.g. B(port.f()),

    the end of an action, e.g. E(port.f()), or null N
  - the *i*-th component of a vector is an event of the *i*-th process

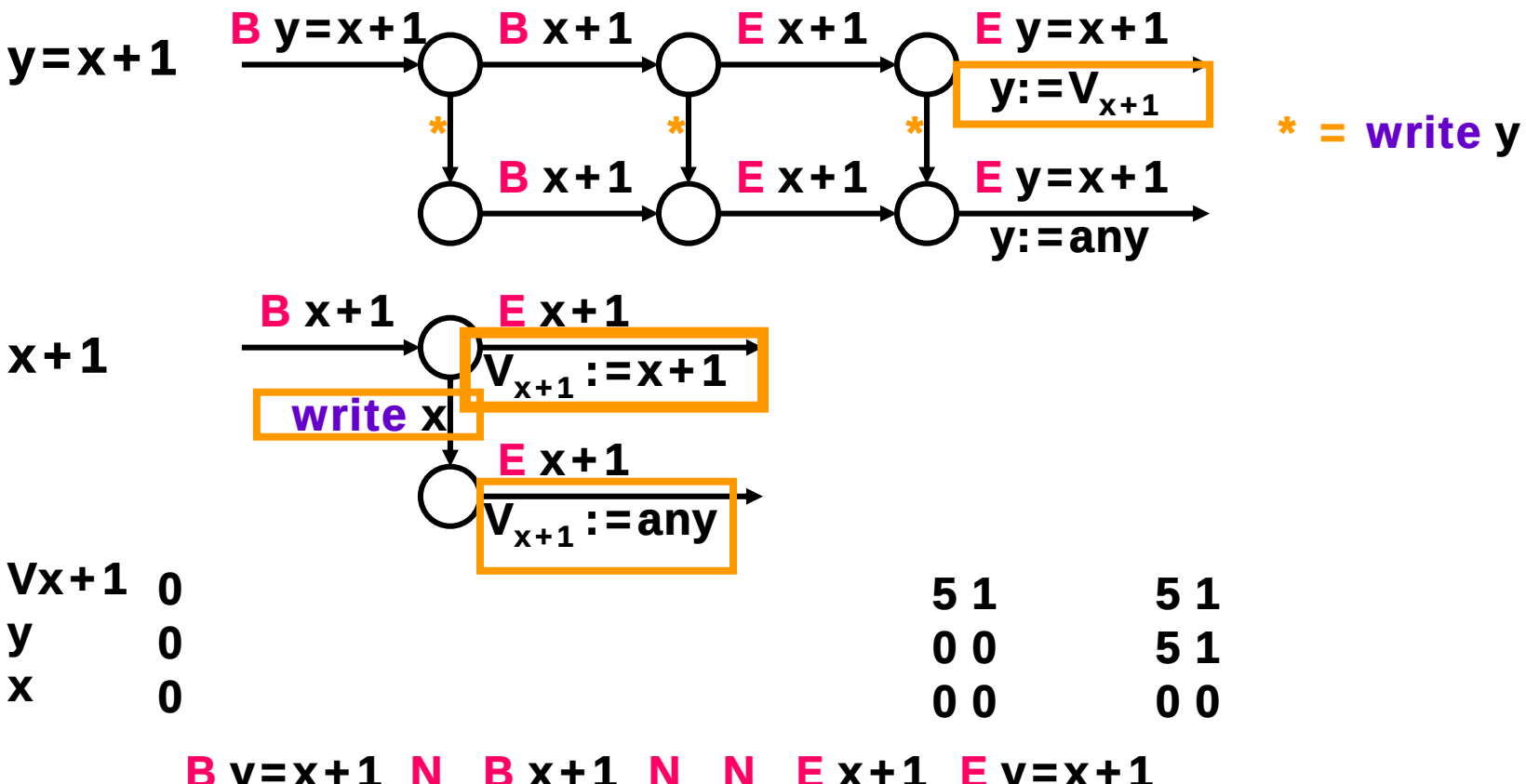
- An execution is *legal* if
  - it satisfies all coordination constraints, and
  - it is accepted by all action automata.

Execution semantics

- Action automaton:
  - one for each action of each process
    - defines the set of sequences of events that can happen in executing the action
  - a transition corresponds to an event:
    - it may update shared memory variables:
      - process and media member variables
      - values of actions-expressions
    - it may have guards that depend on states of other action automata and memory variables
  - each state has a self-loop transition with the null N event.
  - all the automata have their alphabets in common:
    - transitions must be taken together in different automata, if they correspond to the same event.

Action Automata

- y=x+1;



Return

Semantics summary

- Processes run sequential code concurrently, each at its own arbitrary pace.
- Read-Write and Write-Write hazards may cause unpredictable results
  - atomicity has to be explicitly specified.
- Progress may block at synchronization points
  - awaits
  - function calls and labels to which awaits or constraints refer.
- The legal behavior of a netlist is given by a set of sequences of event vectors.
  - multiple sequences reflect the non-determinism of the semantics:

    concurrency, synchronization (awaits and constraints)

Constraints

Two mechanisms are supported to specify constraints:

1. Propositions over temporal orders of states
   - execution is a sequence of states
   - specify constraints using linear temporal logic
   - good for scheduling constraints, e.g.

     "if process P starts to execute a statement s1, no other process can start the statement until P reaches a statement s2."

2. Propositions over instances of transitions between states
   - particular transitions in the current execution: called "*actions*"
   - annotate actions with quantity, such as time, power.
   - specify constraints over actions with respect to the quantities
   - good for real-time constraints, e.g.

     "any successive actions of starting a statement s1 by process P must take place with at most 10ms interval."

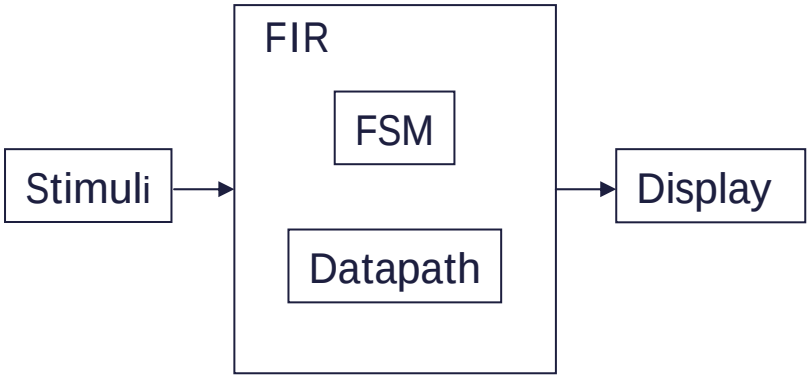
Logic of Constraints (LOC)

- A transaction-level quantitative constraint language
- Works on a sequence of events from a particular execution trace
- The basic components of an LOC formula:
  - Boolean operators: $\neg$ (not), $\vee$ (or), $\wedge$ (and) and $\rightarrow$ (imply)
  - Event names, e.g. “in”, “out”, “Stimuli” or “Display”
  - Instances of events, e.g. “Stimuli[0]”, “Display[10]”
  - Annotations, e.g. “t(Display[5])”
  - Index variable i, the only variable in a formula, e.g. “Display[i-5]” and “Stimuli[i]”

LOC Constraints


```
Stimuli : 0 at time 9
Display : 0  at time 13
Stimuli : 1 at time 19
Display : -6  at time 23
Stimuli : 2 at time 29
Display : -16  at time 33
Stimuli : 3 at time 39
Display : -13  at time 43
Stimuli : 4 at time 49
Display : 6  at time 53
   ⋮
```

FIR Trace



(SystemC2.0 Distribution)

Throughput: “at least 3 *Display* events will be produced in any period of 30 time units”.

$$t\,(Display[i+3]) - t\,(Display[i]) <= 30$$

Other LOC constraints

- Performance: rate, latency, jitter, burstiness
- Functional: data consistency

Meta-model: architecture components

An architecture component specifies *services*, i.e.

- what it *can* do:

  interfaces, methods, coordination (awaits, constraints), netlists

- how much it *costs:*

  quantities, annotated with events, related over a set of events

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

```
medium Bus implements BusMasterService …{
  port BusArbiterService Arb;
  port MemService Mem; …
  update void busRead(String dest, int size) {
     if(dest== … ) Mem.memRead(size);
   }

  …
```

Meta-model: quantities

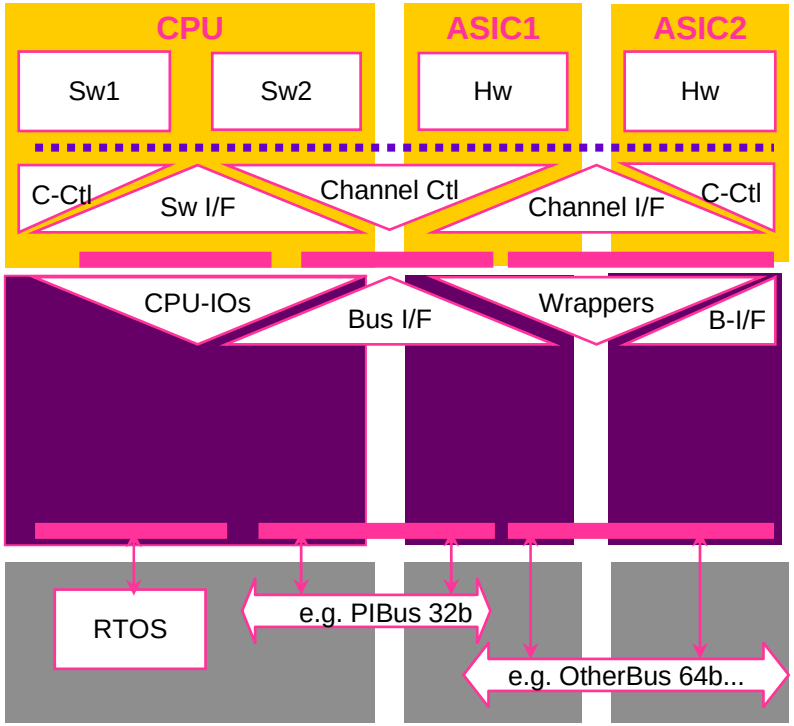
- The domain D of the quantity, e.g. *real* for the global time,
- The operations and relations on D, e.g. subtraction, <, =,
- The function from an event instance to an element of D,
- Axioms on the quantity, e.g.

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

```
class GTime extends Quantity {
        double t;
        double sub(double t2, double t1){...}
        double add(double t1, double t2){...}
        boolean equal(double t1, double t2){ ... }
        boolean less(double t1, double t2){ ... }
        double A(event e, int i){ ... }
        constraints{
          forall(event e1, event e2, int i, int j):
          GXI.A(e1, i) == GXI.A(e2, j) -> equal(A(e1, i), A(e2, j)) &&
          GXI.A(e1, i) < GXI.A(e2, j) -> (less(A(e1, i), A(e2, j)) || equal(A(e1, i), A(e2. j)));
}}
```

Meta-model: architecture components

- This modeling mechanism is generic, independent of services and cost specified.
- Which levels of abstraction, what kind of quantities, what kind of cost constraints should be used to capture architecture components?
  - depends on applications: *on-going research*



Transaction:

Services:

- fuzzy instruction set for SW, execute() for HW

- bounded FIFO (point-to-point)

Quantities:

- #reads, #writes, token size, context switches

Virtual BUS:

Services:

- data decomposition/composition

- address (internal v.s. external)

Quantities: same as above, different weights

Physical:

Services: full characterization

Quantities: time

Quantity resolution

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

1. quantity requests

for each process *Pi*, for each event *e* that *Pi can* take, find all the quantity constraints on *e*.

In the meta-model, this is done by explicitly requesting quantity annotations at the relevant events, i.e. Quantity.request(event, requested quantities).

2. quantity resolution

find a vector made of the candidate events and a set of quantities annotated with each of the events, such that the annotated quantities satisfy:

- all the quantity requests, and
- all the axioms of the Quantity types.

In the meta-model, this is done by letting each Quantity type implement a resolve() method, and the methods of relevant Quantity types are iteratively called.

- theory of fixed-point computation

Quantity resolution

- The 2-step approach is same as how schedulers work, e.g. OS schedulers, BUS schedulers, BUS bridge controllers.
- Semantically, a scheduler can be considered as one that resolves a quantity called *execution index*.
- Two ways to model schedulers:

1. As processes:
   - explicitly model the scheduling protocols using the meta-model building blocks
   - a good reflection of actual implementations
2. As quantities:
   - use the built-in request/resolve approach for modeling the scheduling protocols
   - more focus on resolution (scheduling) algorithms, than protocols: suitable for higher level abstraction models

Programmable Arch. Modeling

- Computation Services

PPC405 | MicroBlaze | SynthMaster | SynthSlave

Computation Services
Read (addr, offset, cnt, size), Write(addr, offset, cnt, size),
Execute (operation, complexity)

- Communication Services

Processor Local Bus (PLB) | On-Chip Peripheral Bus (OPB) | BRAM

Communication Services
addrTransfer(target, master)
addrReq(base, offset, transType, device)
addrAck(device)

dataTransfer(device, readSeq, writeSeq)
dataAck(device)

- Other Services

OPB/PLB Bridge | Mapping Process

Task Before Mapping
Read (addr, offset, cnt, size)

Programmable Arch. Modeling

- Coordination Services

PPC Sched

MicroBlaze Sched

PLB Sched

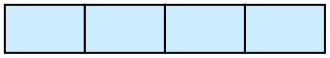
OPB Sched

BRAM Sched

General Sched

Request (event e)

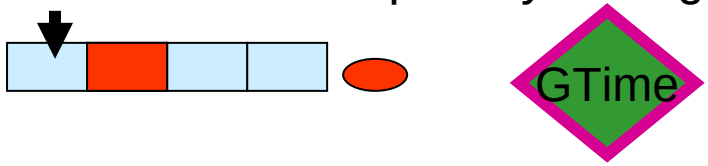
-Adds event to pending queue of requested events



PostCond()
Resolve()

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

-Uses algorithm to select an event from the pending queue



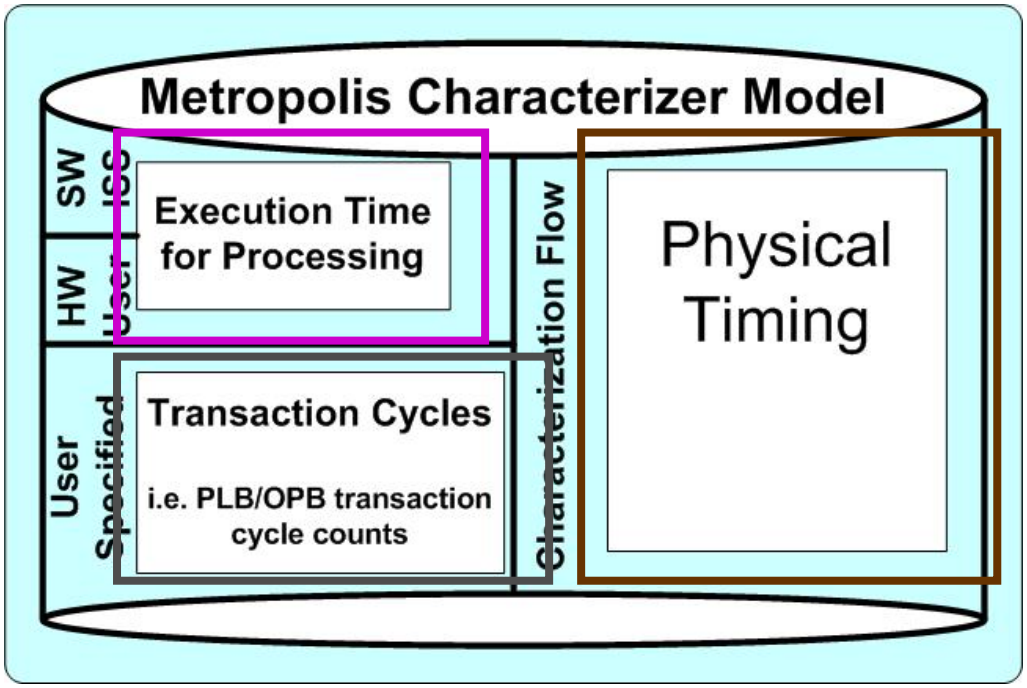
Prog. Platform Characterization

Create database **ONCE** prior to simulation and populate with independent (modular) information.

1. Data detailing performance based on physical implementation.

2. Data detailing the composition of communication transactions.

3. Data detailing the processing elements computation.



From Char Flow Shown

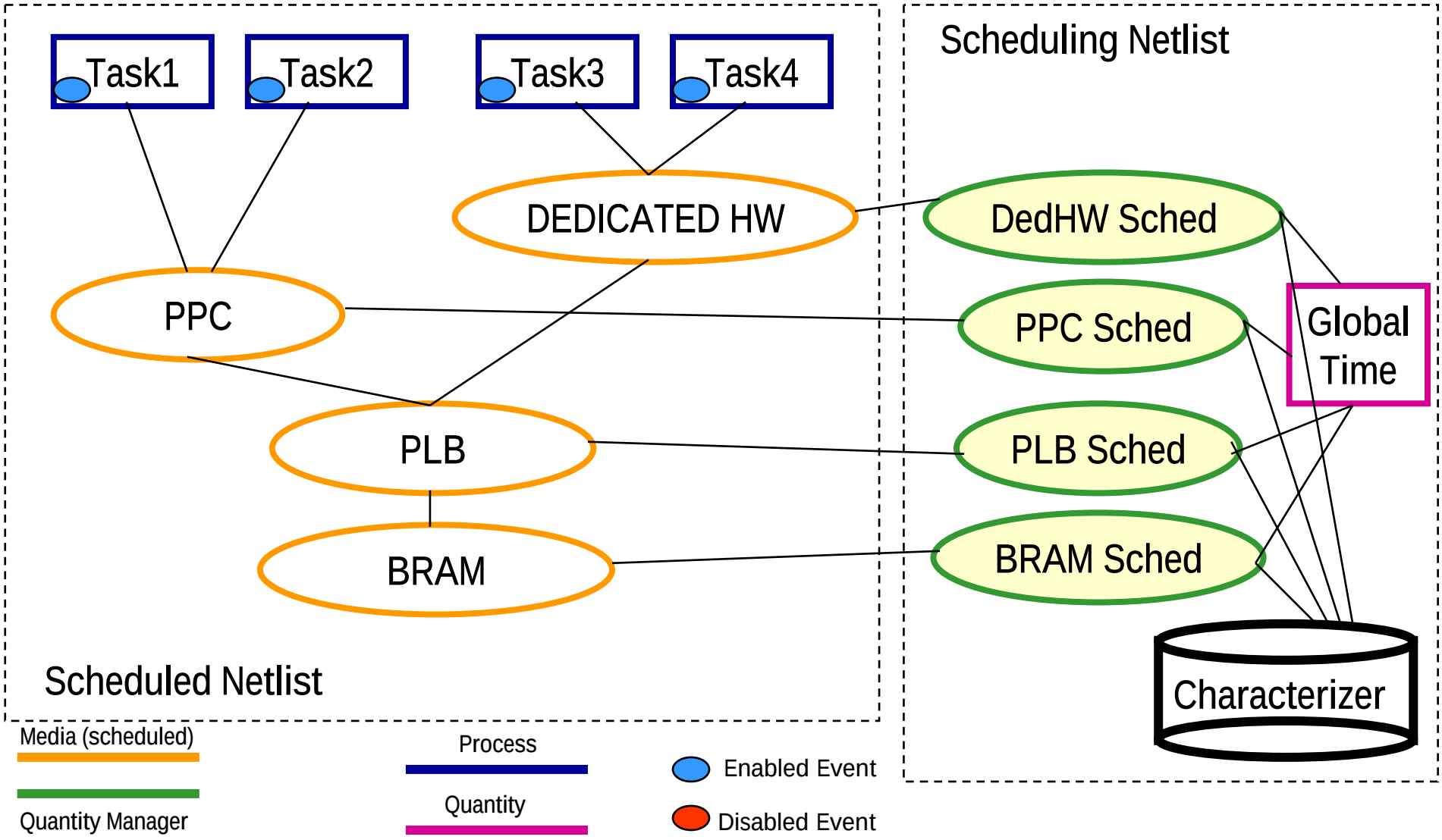
From Metro Model Design

From ISS for PPC

Work with Xilinx Research Labs

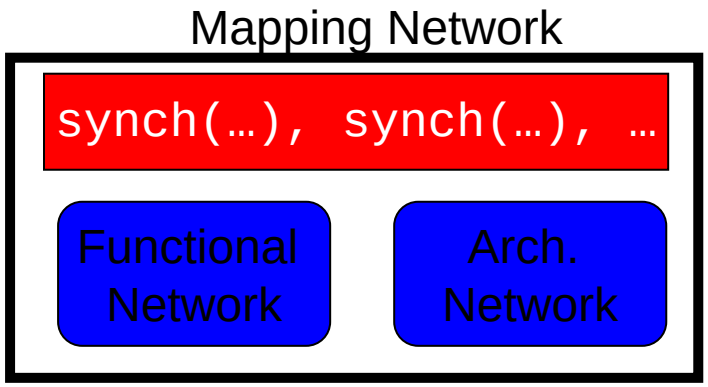
1. Douglas Densmore, Adam Donlin, A.Sangiovanni-Vincentelli, *FPGA Architecture Characterization in System Level Design*, Submitted to CODES 2005.
2. Adam Donlin and Douglas Densmore, *Method and Apparatus for Precharacterizing Systems for Use in System Level Design of Integrated Circuits*, Patent Pending.

Modeling & Char. Review



Mapping in Metropolis

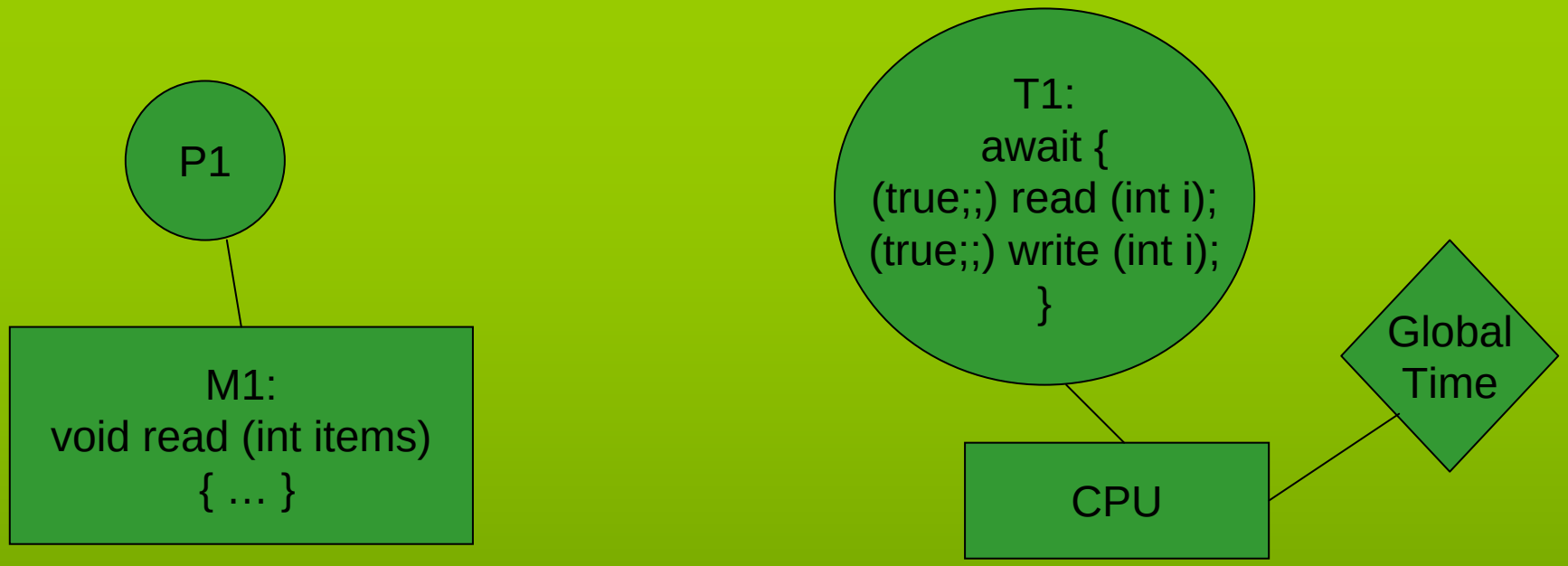
- Objectives:
  - Map a functional network with an architectural network without changing either of the two
    - Support design reuse
  - Specify the mapping between the two in a formal way
    - Support analysis techniques
    - Make future automation easier
- Mechanism:
  - Use declarative synchronization constraints between events
  - One of the unique aspects of Metropolis



Synchronization constraints

- Synchronization constraint between two events e1 and e2:
  - ltl synch(e1, e2)
  - e1 and e2 occur simultaneously or not at all during simulation
- Optional variable equality portion:
  - ltl synch(e1, e2: var1@e1 == var2@e2)
  - The value of *var1* in the scope of e1 is equal to the value of *var2* when e1 and e2 occur
  - Can be useful for “passing” values between functional and architectural models

Metropolis Example




```
e1 = beg(P1, M1.read);
e2 = beg(T1, T1.read);
ltl synch(e1, e2: items@e1 == i@e2);
e3 = end(P1, M1.read);
e4 = end(T1, T1.read);
ltl synch(e3, e4);
```

Meta-model: mapping netlist

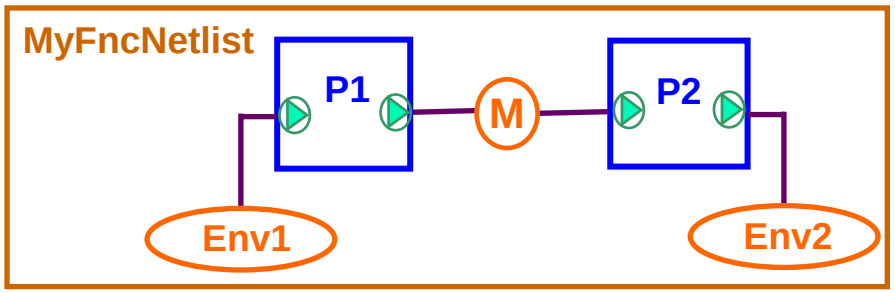
MyMapNetlist

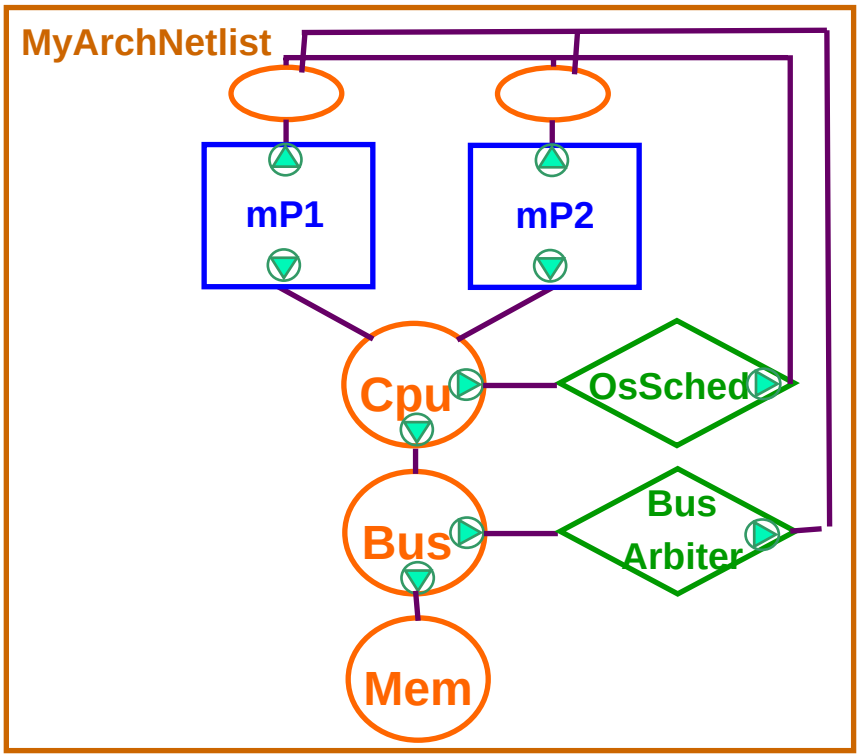
B(P1, M.write) <=> B(mP1, mP1.writeCpu); E(P1, M.write) <=> E(mP1, mP1.writeCpu);

B(P1, P1.f) <=> B(mP1, mP1.mapf); E(P1, P1.f) <=> E(mP1, mP1.mapf);

B(P2, M.read) <=> B(P2, mP2.readCpu); E(P2, M.read) <=> E(mP2, mP2.readCpu);

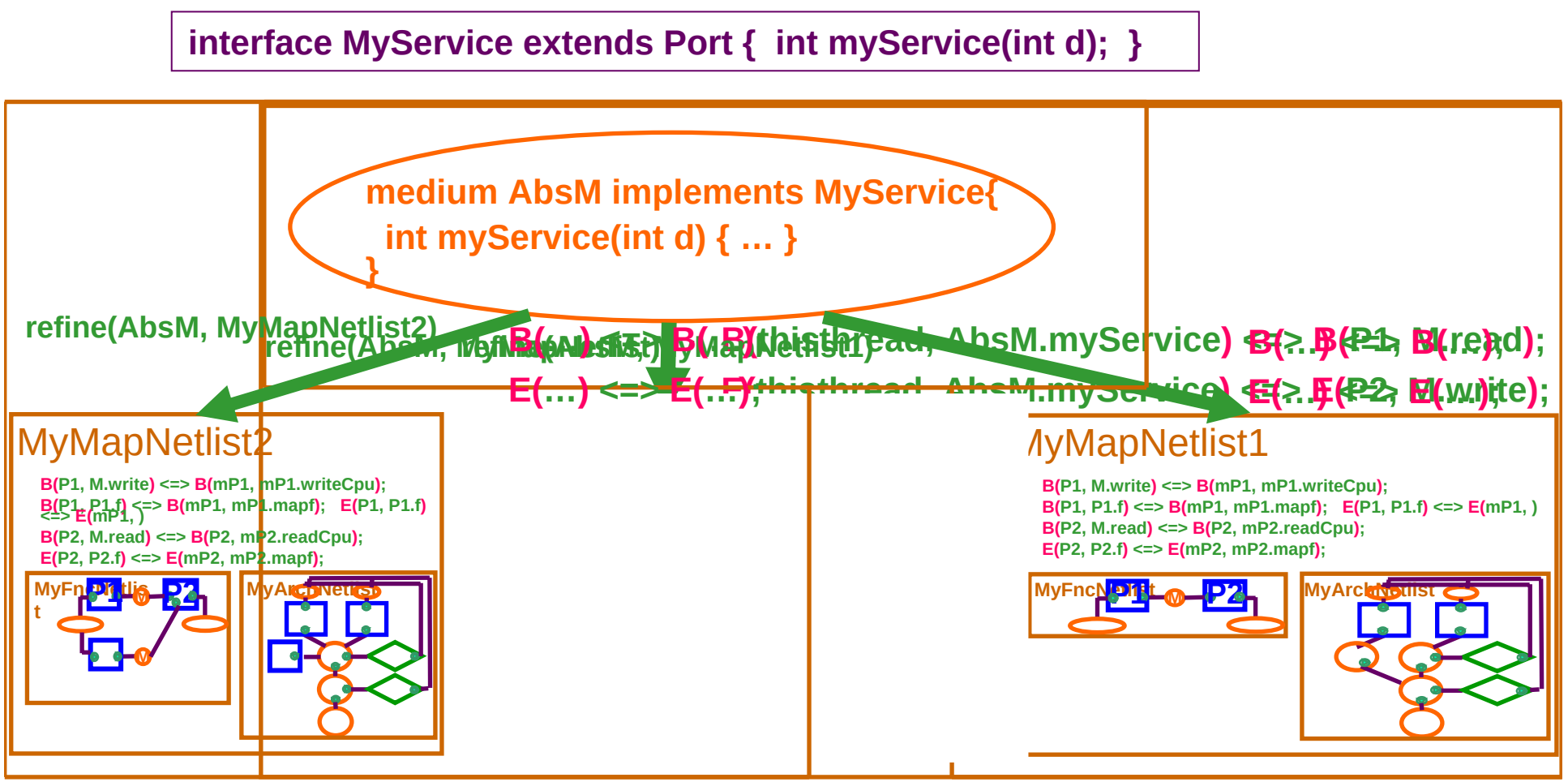
B(P2, P2.f) <=> B(mP2, mP2.mapf); E(P2, P2.f) <=> E(mP2, mP2.mapf);



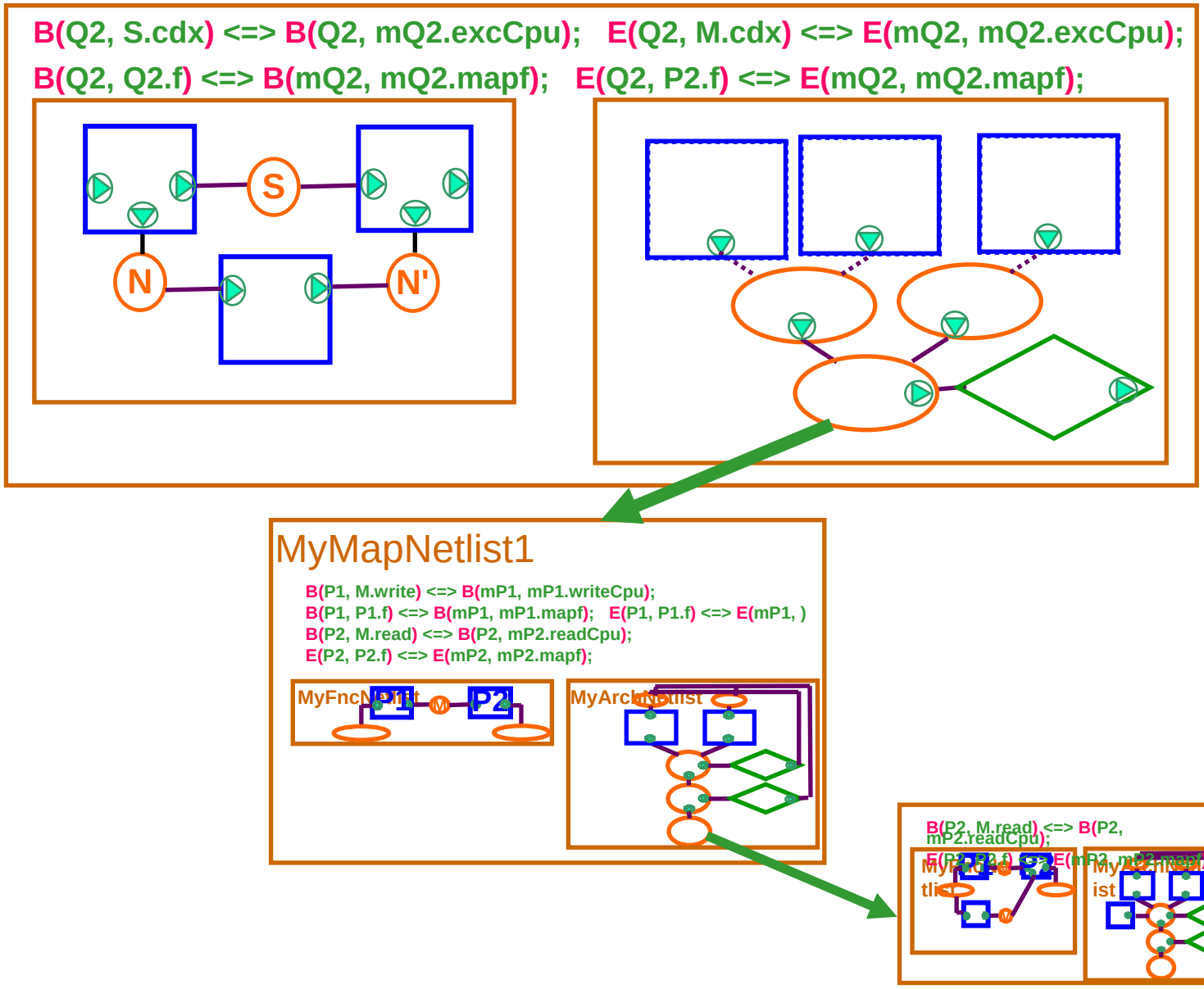


Meta-model: platforms

A set of mapping netlists, together with constraints on event relations to a given interface implementation, constitutes a ***platform*** of the interface.



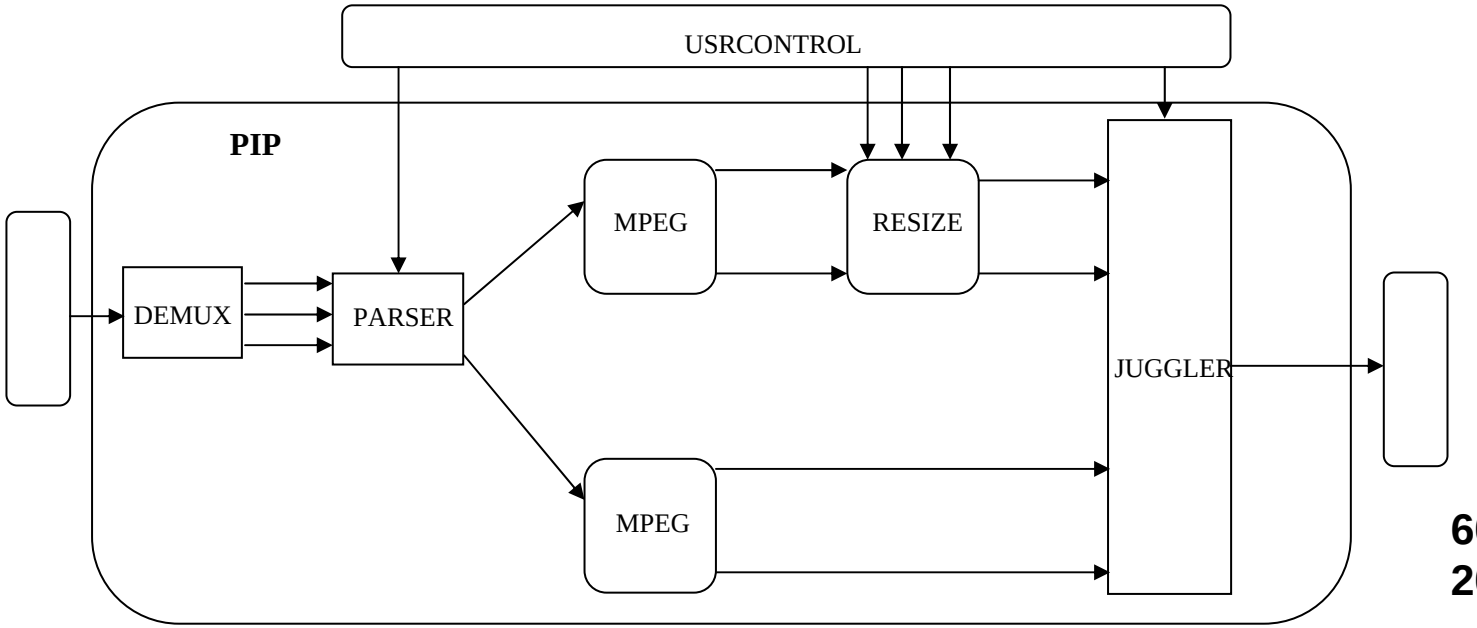
Meta-model: recursive paradigm of platforms



Metropolis Driver: Picture-in-Picture Design Exercise

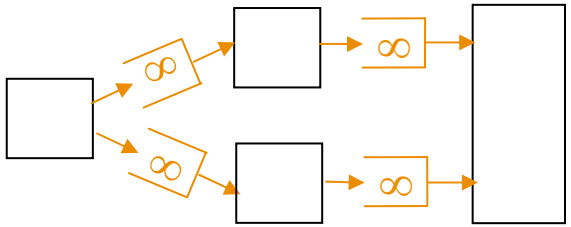
Evaluate the methodology with formal techniques applied.

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

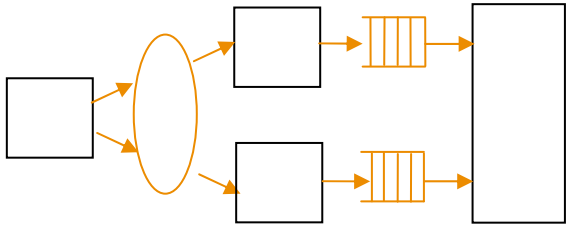


60 processes with 200 channels

Multi-Media System: Abstraction Levels

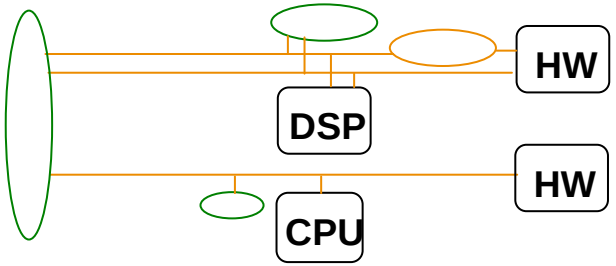


- Network of processes with sequential program for each
- Unbounded FIFOs with multi-rate read and write

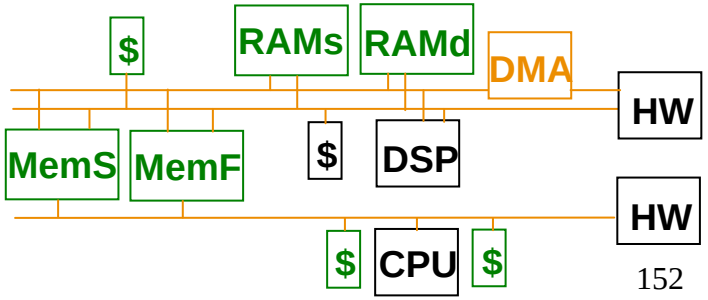


- Communication refined to bounded FIFOs and shared memories with finer primitives (called TTL API):
  allocate/release space, move data, probe space/data

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

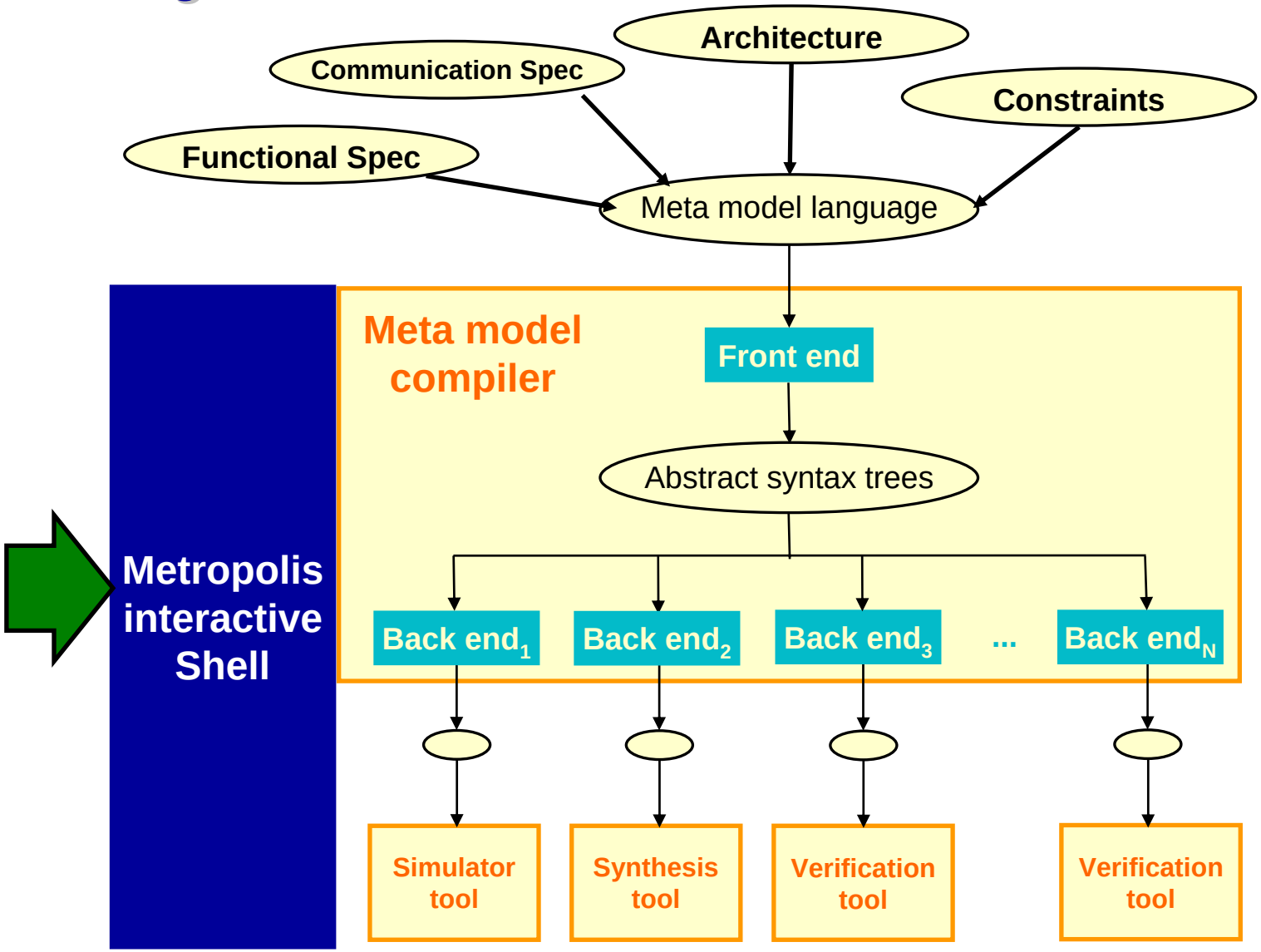


- Cycle-accurate services and performance models



Metropolis design environment

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



Backend Point Tools

- Synthesis/refinement:
  - Quasi-static scheduling
  - Scheduler synthesis from constraint formulas
  - Interface synthesis
  - Refinement (mapping) synthesis
  - Architecture-specific synthesis from concurrent processes for:
    - Hardware (with known architecture)
    - Dynamically reconfigurable logic
- Verification/analysis:
  - Static timing analysis for reactive processes
  - Invariant analysis of sequential programs
  - Refinement verification
  - Formal verification for software

Conclusions

- The trade-off between hardware and software starts long before the RTL design of an SoC
- Starting from the system specification:
  - Functionality, i.e., WHAT the system is required to do
  - Constraints, i.e., the set of requirements that restrict the design space by taking into consideration non functional aspects of the design such as cost, power consumption, performance, fault tolerance and physical dimensions.
  - Architecture, i.e., the set of available components from which the designer can decide HOW she can implement the functionality satisfying the constraints
- The PBD methodology progresses towards the implementation of the design "mapping" the functionality of the design to the available components.
  - The library of available components (they can be already fully designed or they can be considered virtual components) is called a platform.
  - Mapping implies the selection of the components, of their interconnection scheme and of the allocation of the functionality to each
  - Several models and methods are applied to achieve the final implementation

Acknowledgment

- Prof. Alberto S.Vincentelli
  - A lot of material from his course at University of California at Berkeley
- My collaborators at the PARADES Research Labs
  - L.Mangeruca, M.Baleani, M.Carloni, A.Balluchi, L.Benvenuti, T.Villa and others
- Grant Martin, Chief Scientist at Tensilica
  - Who provided all the slides on configurable and extendible cores
- Researchers at Cadence Berkeley Lab:
  - Yoshi Watanabe, Felice Balarin
- Researchers at United Technology (Carrier/OTIS)
  - Clas Jacobson, and others
- Sonics
  - slides on communication centric flow

Thank you!!