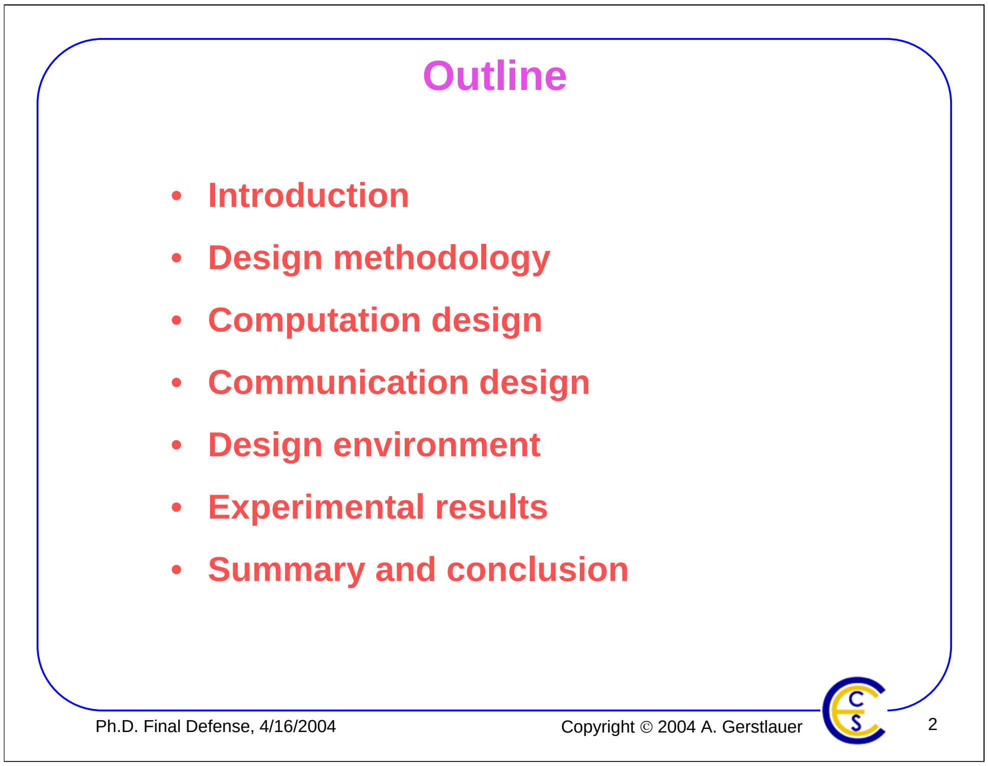
Modeling Flow for Automated System Design and Exploration

Andreas Gerstlauer
Center for Embedded Computer Systems
University of California, Irvine
`http://www.cecs.uci.edu/~gerstl`



The talk is outlined as follows:

I will start with an introduction in which I try to motivate the background for my research, provide an overview of system design in general and give a definition of the problem being solved in this work

After an overview of the overall system design methodology, I will then focus on describing the design steps that comprise computation and communication design tasks.

The design flow has been implemented in the form of a design environment and I want to give a brief overview of the environment and my contributions to it before showing some experimental results obtained by applying to flow to several industrial-strength design examples.

Finally, the talk concludes with a summary and a list of contributions.

Motivation and Goals

- **Productivity gap, increase in design complexity**
  - Raise level of abstraction
  - Intellectual property (IP) reuse

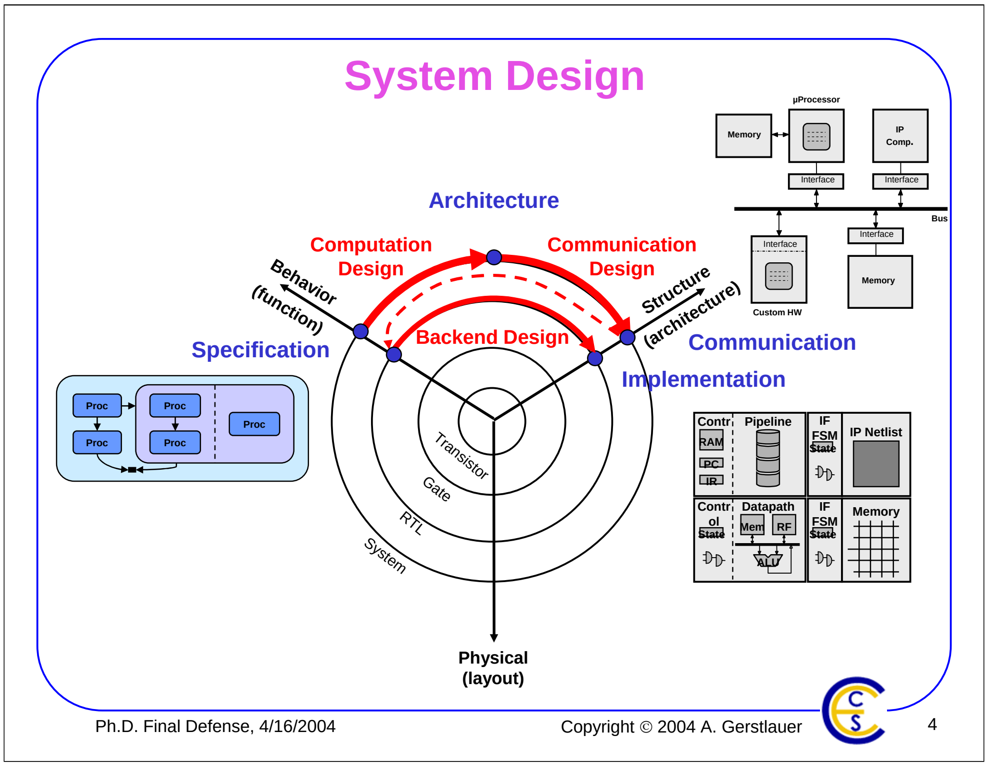
➢ **Well-defined, rigorous, structured design flow**
  - Unambiguous abstractions, models, transformations
  - Systematic flow from specification to implementation
  - Reliable feedback at early stages

  ➢ Design automation for synthesis, verification
  ➢ Rapid, early design space exploration

Ph.D. Final Defense, 4/16/2004 Copyright © 2004 A. Gerstlauer 3

Against the background of the well-known productivity gap in the design of SoCs and embedded computer systems in general, both raising the level of abstraction and massive reuse of intellectual property (IP) components have been proposed as solutions.

However, arbitrarily raising abstraction levels is not enough. In order to achieve the required productivity gains, a systematic, structure, and complete design flow from specification down to implementation with clear and unambiguous abstraction levels, models, and transformations is needed. Only a well-defined, formalized flow enables design automation for synthesis and verification. Furthermore, abstractions have to be defined such that critical issues can be addresses reliably to enable rapid, early design space exploration.



Using the Y-Chart for classification of design processes, design in general is the process of moving from a behavioral to a structural and eventually physical description where designs can be done at different levels of granularity from individual transistors up to complete systems.

System design starts with a purely functional system specification. Based on a separation of computation and communication, a system architecture and a bus-functional communication system are derived from the specification through computation and communication design tasks. Finally, in a backend design tasks, each of the components of the system is then brought down to a cycle-accurate implementation by implementing its behavior in hardware or software on top of the component's microarchitecture.

Problem Definition

- **Bridge semantic gap**
  - Split into manageable design steps
- **Define intermediate abstraction levels, design models**
  - Synthesizable representation of critical design issues
  - Abstract unnecessary implementation detail
- **Define design steps**
  - Design decisions, model transformations

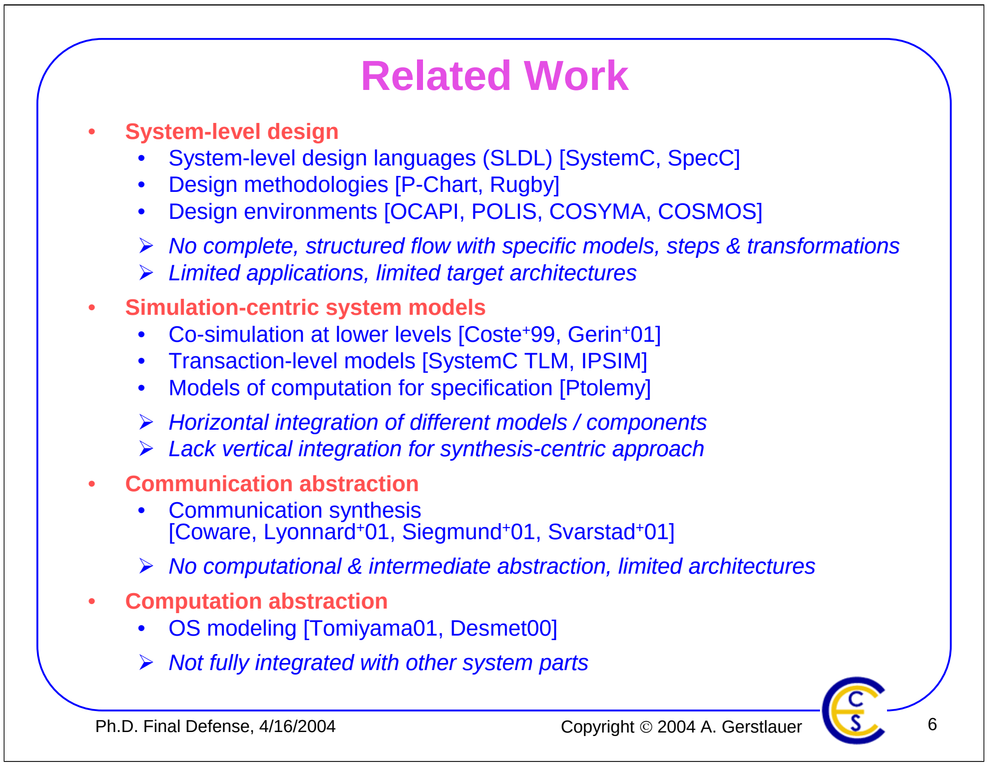
➢ **Enable design automation**
  ➢ Automated model refinement, decision making
➢ **Enable rapid, early design space exploration**
  ➢ Reliable feedback about critical issues at high levels
➢ **Support for realistic SoC designs**
  ➢ Wide range of applications, target architectures

Ph.D. Final Defense, 4/16/2004 5

In general, the semantic gap between specification and implementation is too big to be completed in one step. In order to bridge the gap, the design process therefore has to be broken down into smaller, manageable steps.

The problem is therefore to define such a flow of successive design steps and intermediate design models. Intermediate abstractions and corresponding design models have to be defined such that critical issues can be addressed early and reliably while unnecessary implementation details are abstracted away. Then, each design step has to be properly defined by formalizing the design decisions and model transformations necessary in that step.

All in all, the resulting design flow should support a wide variety of realistic system applications and target architectures. The formalized nature of the process should enable design automation for decision making and model refinement. Finally, together with design automation, high-level models should enable rapid, early design space exploration with fast turn-around times.



Related work in the area so far has been dealing with several aspects:

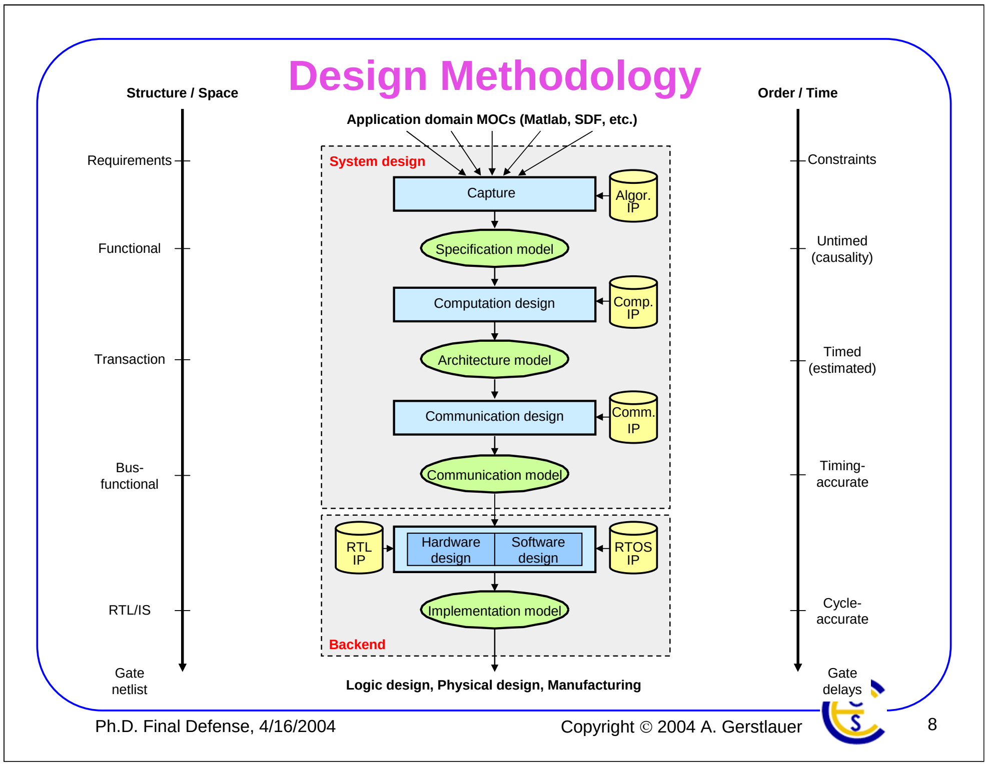
There are a number of system-level design languages, methodologies and design environments. However, none of these define an actual design flow with specific models, steps and transformations. Furthermore, some of these approaches only support limited applications or target architectures. For our work, we use the SpecC SLDL to describe all the design models in our flow. However, the concepts presented are independent of the language and can be equally applied to other SLDLs with support for system modeling.

In terms of system design models, there are several approaches dealing with horizontal integration of different models at different levels of abstraction. However, none of these works deals with the vertical integration needed to provide a path to implementation.

Finally, there are some approaches that deal with automated synthesis of computation or communication. However, none of these are integrated into an overall system design flow, they don't provide intermediate models for rapid, early design space exploration, and they are often limited in their support for realistic applications or architectures.

Outline

- Introduction
- **Design methodology**
- Computation design
- Communication design
- Design environment
- Experimental results
- Summary and conclusion

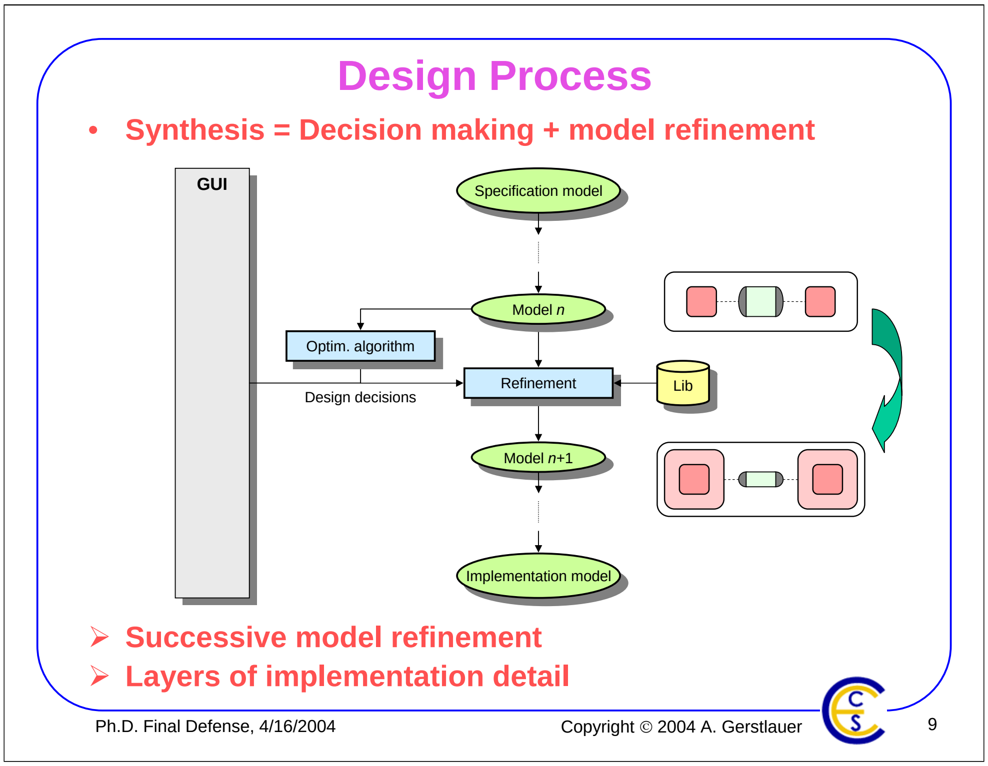


The overall design methodology is shown here. In general, as we move down in the level of abstraction from specification to implementation, more and more implementation detail in the form of structure and order is added.

System design starts with the specification model that captures and unifies domain-specific models of requirements and constraints. The specification model is a purely functional description of the desired system behavior and it is untimed, i.e. partially ordered based only on causality.

In the system design process, the specification is then mapped onto a set of system components connected by system busses or other communication structures through computation and communication design tasks. The intermediate architecture model describes the system as a virtual architecture of processors communicating via abstract channels, annotated with estimated processor execution delays. The communication flow at the end of system design is a bus-functional description of the system as a netlist of timing-accurate components connected by wires..

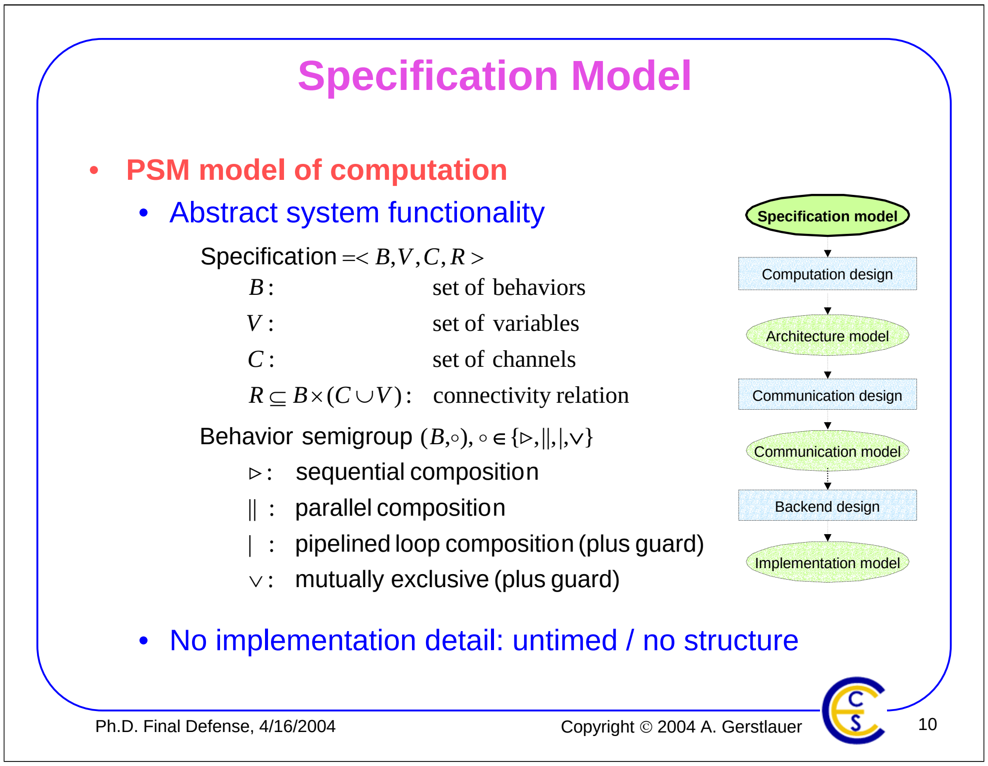
Finally, in the backend design process, components of the communication model are each brought down to a cycle-accurate implementation at the RTL or instruction-set level through hardware or software design tasks.



Each of the different design tasks is then further broken down into several successive design steps. With each step, a design model at a certain level of abstraction is synthesized into a model at the next lower level. The result is a flow with successive refinement of design models where a new layer of implementation detail is added to the design in each step.

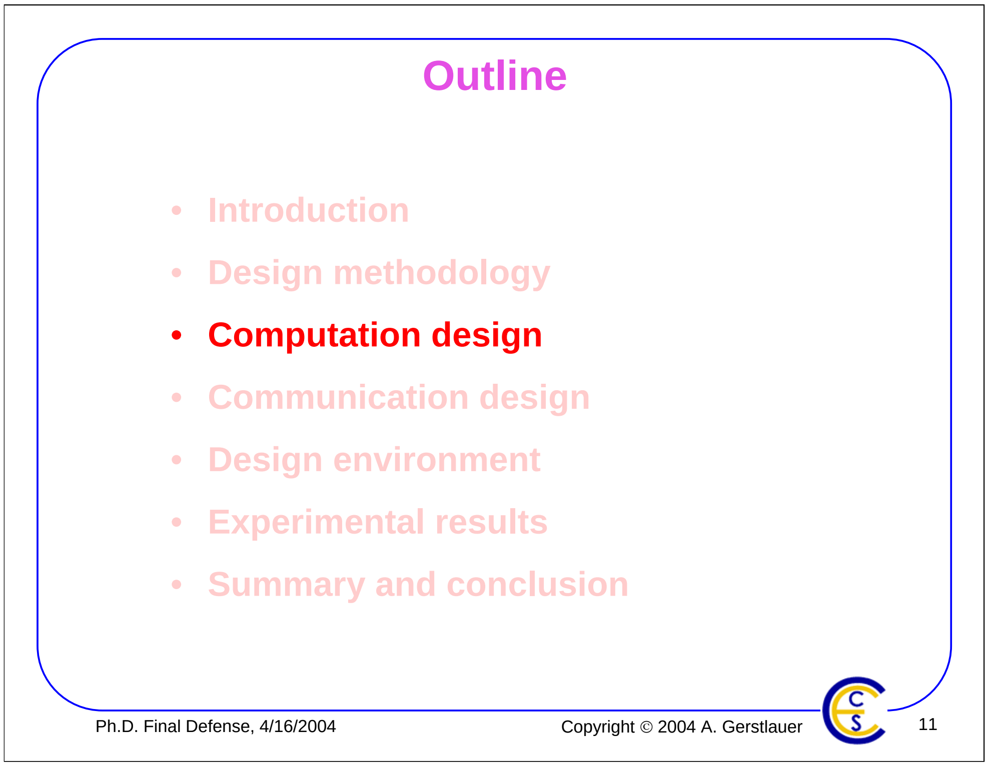
In general, synthesis of designs in each step can be separated into the two distinct parts: making design decisions on the one hand and refining the design model to represent the results of those decisions on the other hand.

Both parts can generally be manual or automated. To provide controllability, transparency and observability of the design process, decisions are made under the control of the designer through a graphical user interface or by selectively employing automated decision making algorithms. On the other hand, with the help of refinement tools that automatically generate design models from each other, there is no need for tedious, error-prone manual model rewriting.

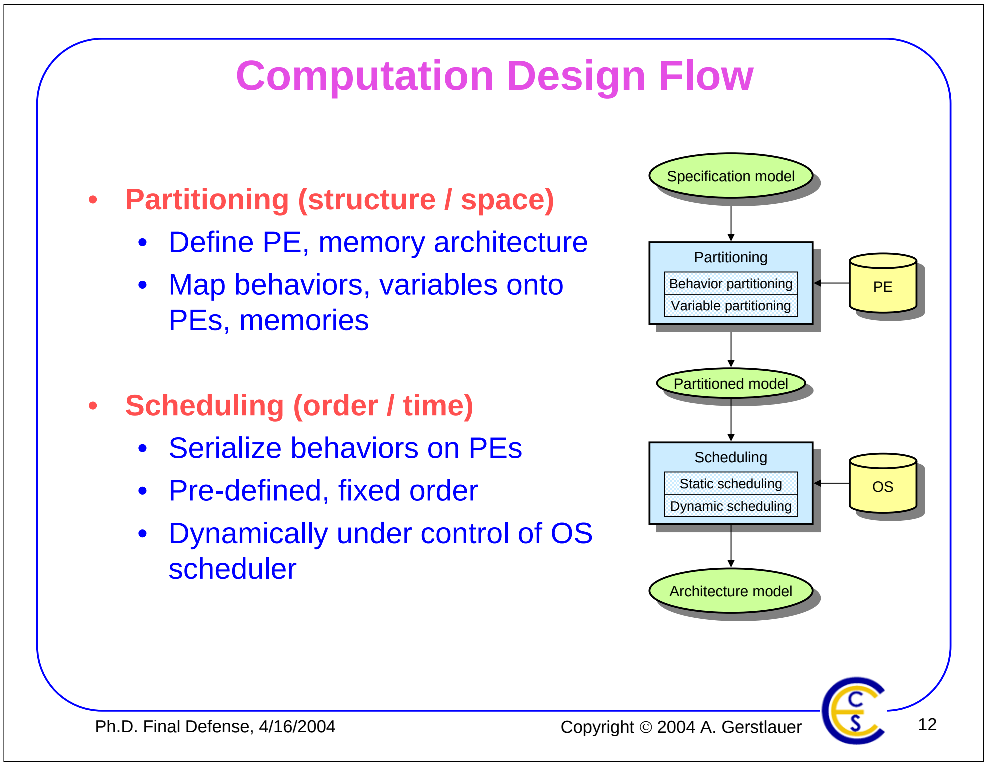


As mentioned previously, the starting point for system design is the system specification model. The specification model is a program state machine model of computation. System functionality is described as a set of behaviors that communicate through variables and abstract channels. Behaviors can be arranged hierarchically in a sequential, parallel, pipelined or state machine fashion. Behaviors the leaf of the hierarchy then contain basic algorithms in the form of C code.

In general, the specification model is free of any implementation detail and as such is untimed and its behaviors do not make any implications about the structure of the system architecture.



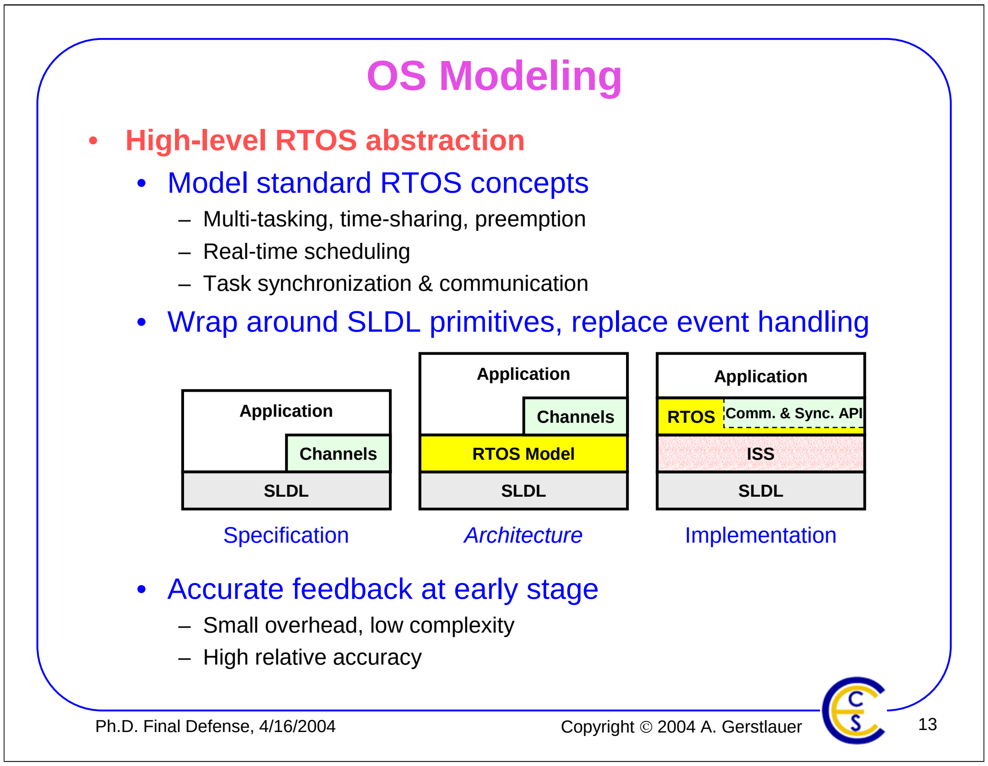
In the rest of presentation, starting with compuation design, I want to focus on compuation and communication design tasks as the main tasks of system design. Due to time reasons, details of the backend design task are not presented here.



The purpose of computation design is to implement the computation in the specification as represented by its behaviors operating on variables on a virtual architecture of processing elements and memories.

In a first part, the structure of the computation architecture is defined by partitioning behaviors and variables onto PEs and memories. This requires allocation of a set of PEs and a set of shared system memories out of the PE database. Then, behaviors and variables have to be mapped onto those PEs and memories.

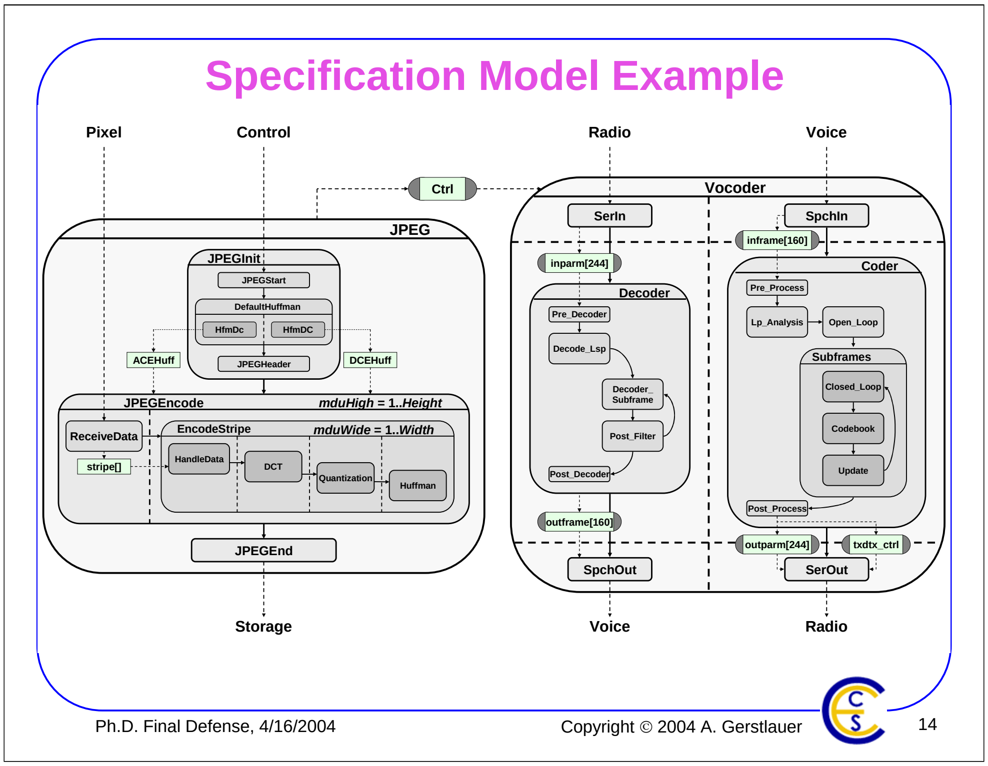
In the second part, the order of behavior execution on the inherently sequential PEs is determined. Behaviors can be scheduled statically or dynamically. In static scheduling, behaviors are arranged in a pre-defined, fixed order. In dynamic scheduling, the order of behaviors is determined dynamically under the control of an OS scheduling algorithm selected out of the OS database.



In order to properly model dynamic scheduling behavior at this high-level of abstraction, an abstracted model of the underlying RTOS is inserted into the design. The RTOS model describes expected RTOS behavior at a high-level without unnecessary implementation details. It supports all the standard RTOS concepts for multi-tasking, dynamic real-time scheduling including preemption and inter-task communication and synchronization..

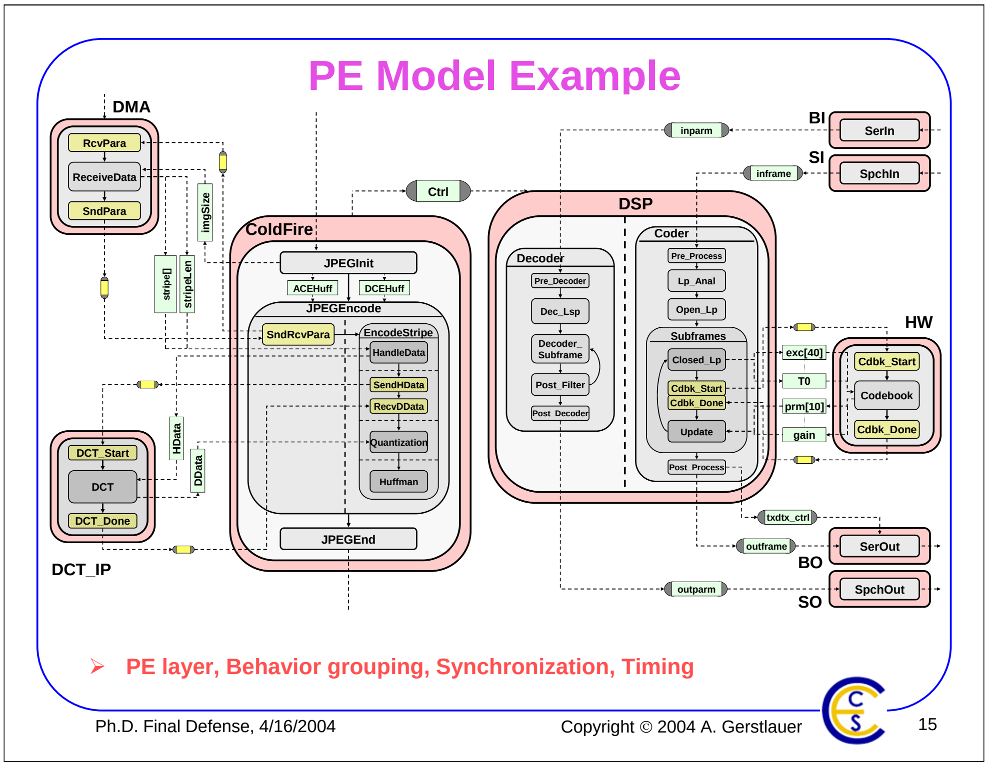
The RTOS model is implemented by wrapping around and replacing the basic event handling primitives of the underlying SLDL. As such, it is inserted as a layer between SLDL and application at the architecture level. As part of backend design, in the implementation, the RTOS model is then later replaced with a real RTOS on top of the processor's instruction set where application channels are mapped down onto RTOS communication primitives.

Using the RTOS model, therefore, accurate feedback about results of dynamic scheduling of behaviors can be obtained early at this high level. The RTOS only adds a small overhead while being able to provide relatively accurate results.

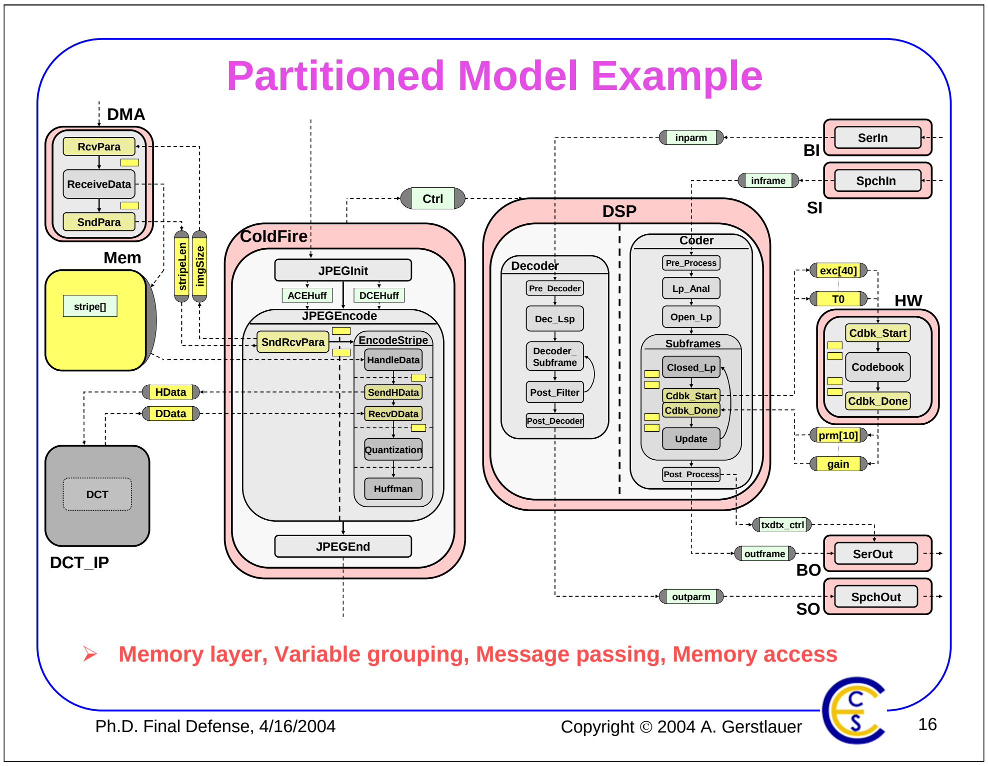


In the following, I want to illustrate the different steps of the computation design task using a system design example of a simplified mobile phone baseband processor. Due to time reasons, I will only provide an overview of the design decisions and model transformations required for each step. Details can be found in the dissertation or can be discussed if necessary.

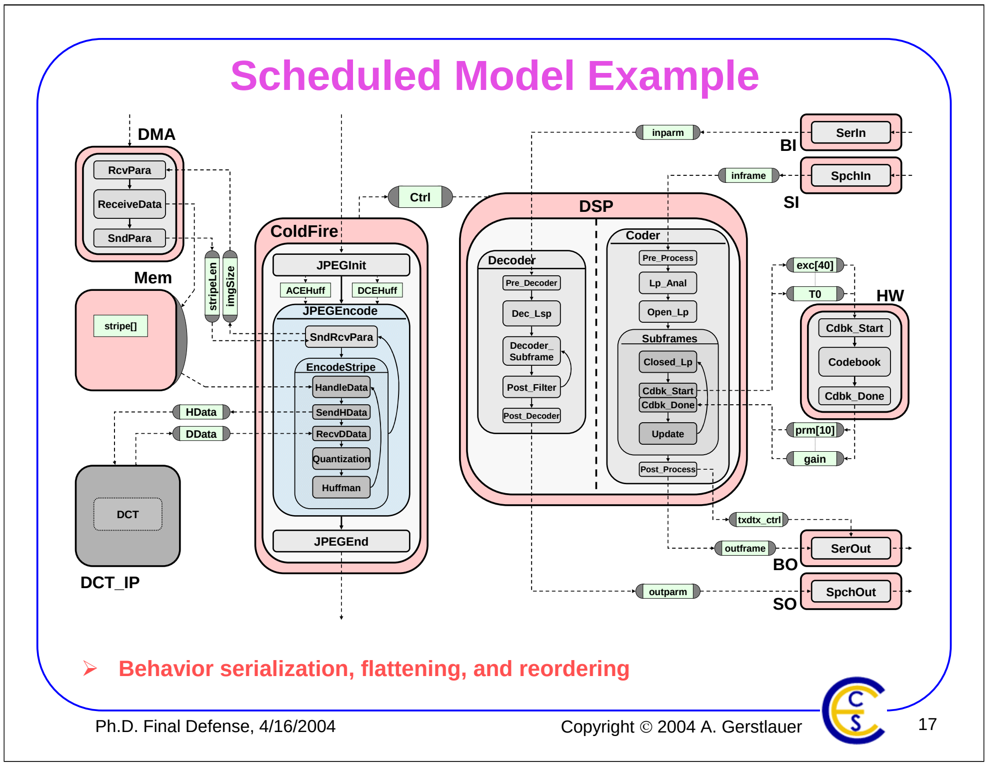
At the top level of its specification, the design examples runs concurrent blocks for JPEG encoding on the left side and voice encoding/decoding on the right. Without going into details, the JPEG encoder at its core encodes still pictures in a double-nested pipeline. The voice encoder/decoder (Vocoder), on the other hand, runs encoding and decoding tasks in parallel.



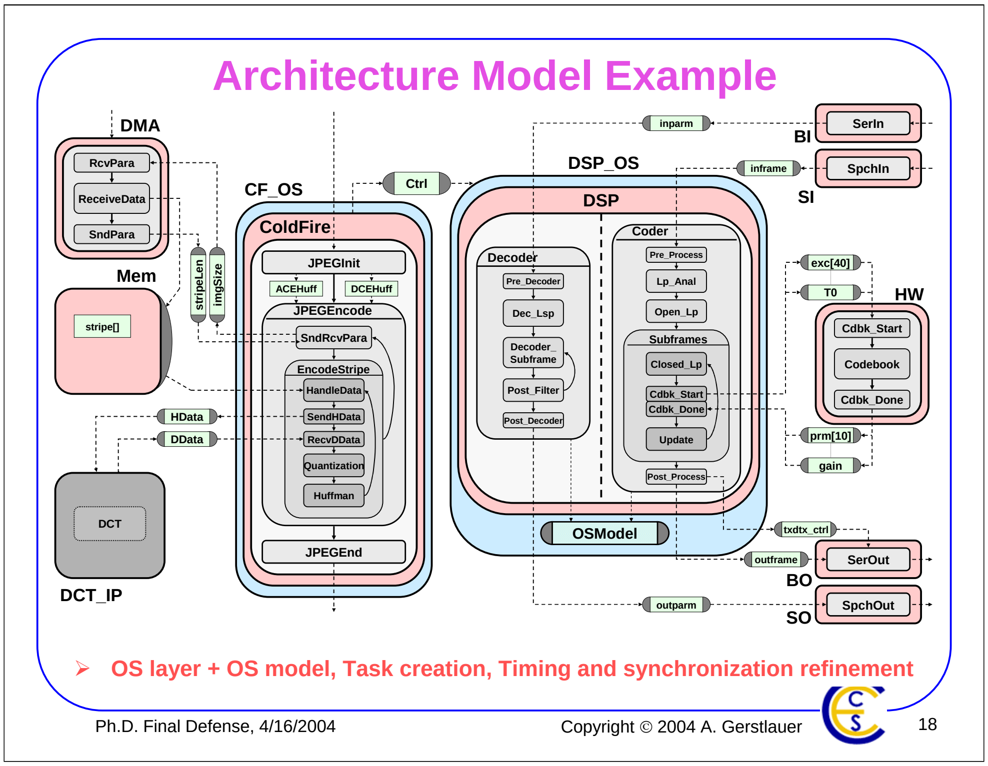
During behavior partitioning, an additional layer of behaviors representing allocated PEs (shown in red) is inserted, original specification behaviors are grouped under PE behaviors according the selected mapping, synchronization behaviors and channels are inserted to preserve execution semantics and leaf behaviors are annotated with estimated execution delays. Not that as a consequence, shared variables become system-global variables between PEs.



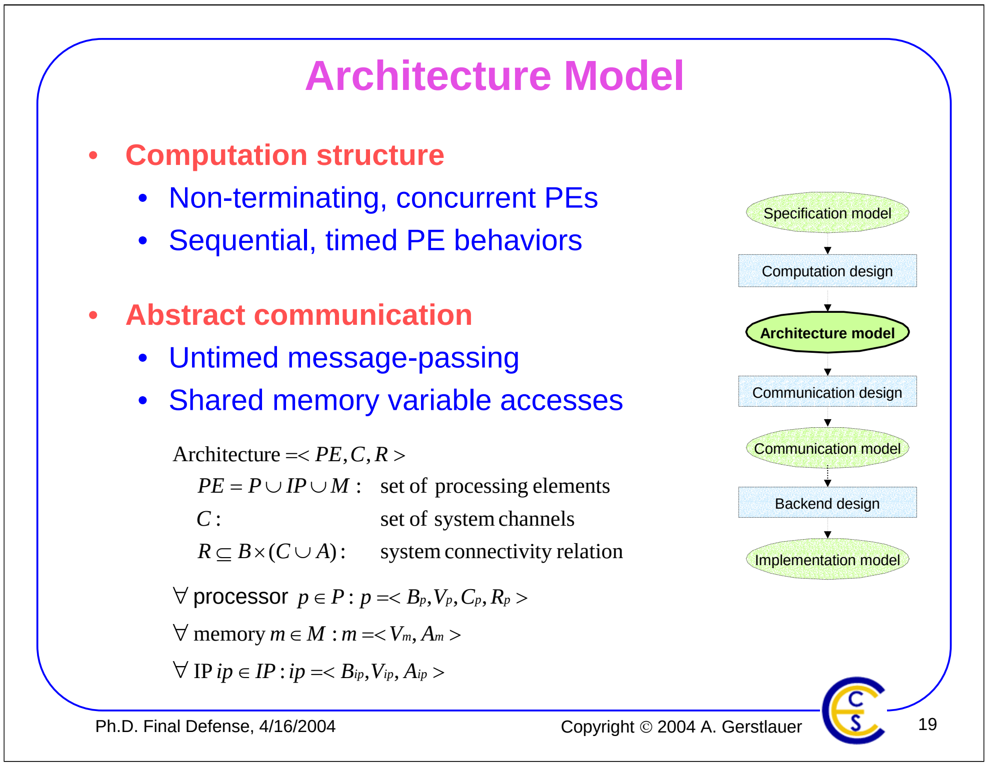
During variable partitioning, a new layer of memory behaviors is inserted, variables are grouped under the memories according to the selected mapping, variable accesses are refined into memory accesses and remaining global variables are distributed into local PE memories and synchronization is updated to exchange updated data values via message passing.



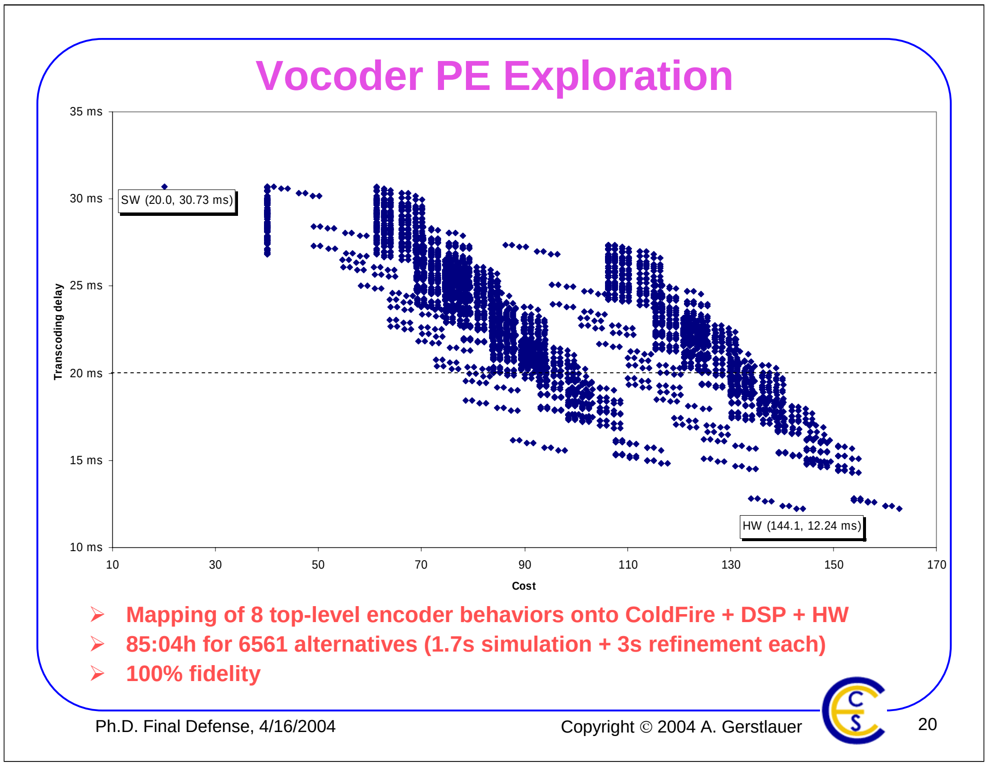
During static scheduling, selected concurrent behaviors are serialized, if necessary parts of the behavior hierarchy are flattened and child behaviors are re-arranged in the selected execution order.



Finally, during dynamic scheduling, an OS layer that includes the selected OS model is inserted around each programmable PE, any remaining concurrent behaviors are turned into OS tasks, and timing and synchronization inside tasks is replaced with corresponding OS model primitives.



The result of computation design is the architecture model. The architecture model describes the system computation structure as a virtual architecture of non-terminating, concurrent PEs communicating via abstract channels or shared memory accesses. Each PE in turn is described as a set of local behaviors connected by local variables and channels.



In order to demonstrate the effectiveness of the architecture model for computation design space exploration, we did several experiments to explore different aspects of the design space for the voice encoder/decoder that is part of the design example.

The graph here shows exploration of the Vocoder PE design space. Using the the scripting capabilities of the design environment, we ran an exhaustive search of all possible mappings of the 8 top-level encoder behaviors onto a Motorola Coldfire processor, a Motorola DSP and a custom hardware co-processor. We assumed fixed costs for the processors and a linear cost function for the hardware. For each alternative, an architecture model was generated and simulated to obtain results for the transcoding delay. The complete generation and analysis for all 6561 alternatives was finished within a few days, showing that even exhaustive, brute-force searches become possible.

As expected, a pure software solution is the cheapest but slowest design whereas a pure hardware solution is the fastest design at a high cost. Comparing exploration results to estimates of an actual implementation of the design, results show a 100% relative accuracy, so-called fidelity. Using the results, we can therefore prune large parts of the design space and focus further design efforts on the pareto-optimal solutions near the transcoding delay constraint.

Vocoder OS Exploration

	Modeling		Simulation	
	Lines of code	Simulation Time	Context switches	Transcoding delay
Partitioned	12,601	16.7 s	0	9.37 ms
Round-robin	13,920	18.1 s	64	10.9 ms
Decoder > Encoder	13,939	17.8 s	2	10.2 ms
Encoder > Decoder	13,939	17.8 s	8	11.3 ms
Implementation	~ 115,500	~ 5 h	2	10.7 ms

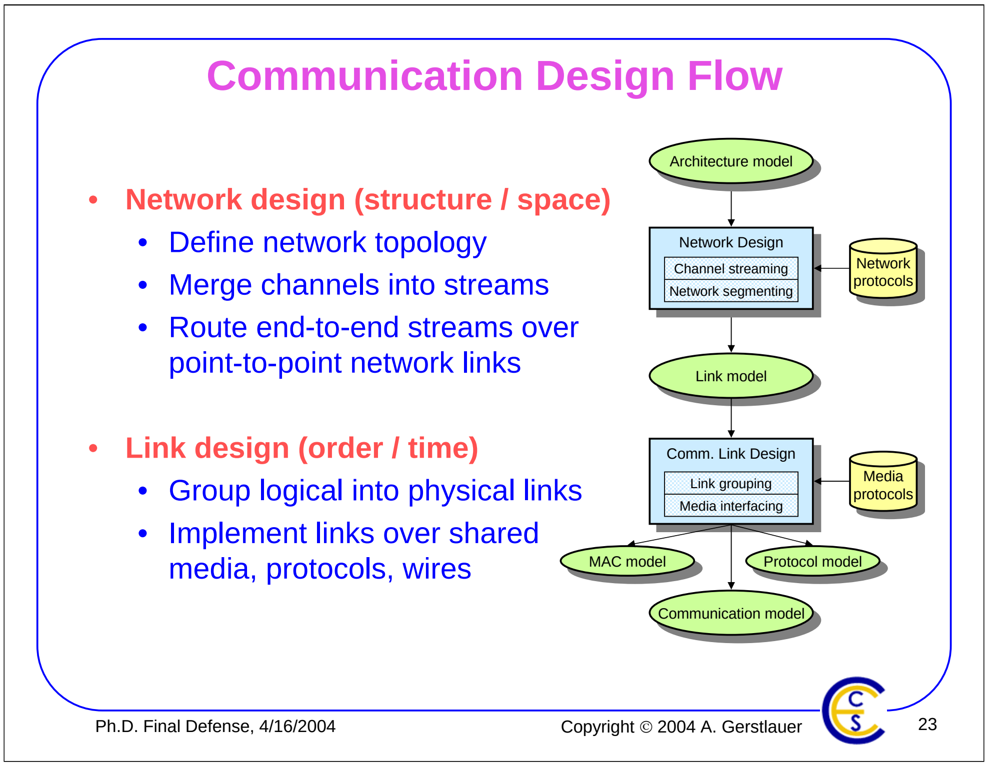
- Modeling effort
  - Automatic scheduling refinement: seconds
  - Manual scheduling refinement: < 1 h
    - 104 lines of code added / changed (< 1%)
  - Implementation: weeks

In another experiment, we evaluated different dynamic scheduling strategies for the encoding and decoding tasks in the Vocoder. Using the OS model, we created architecture models with round-robin and priority-based schedulers with different relative priorities. Results confirm expectations that round-robin scheduling results in low latencies while incurring a lot of context switches. On the other hand, since the decoder has a lower complexity than the encoder, a scheduling strategy in which the decoder has a higher priority than the encoder has the lowest latency and the lowest number of contect switches as it corresponding to a shortest-job-first.

Again, using automatic model refinement tools, all three design alternatives could be generated and simulated within seconds. Compared to actual implementations of the Vocoder on top of real RTOSes which would require weeks to implement, we can explore a much larger part of the design space in a much shorter amount of time.

Outline

- Introduction
- Design methodology
- Computation design
- **Communication design**
- Design environment
- Experimental results
- Summary and conclusion

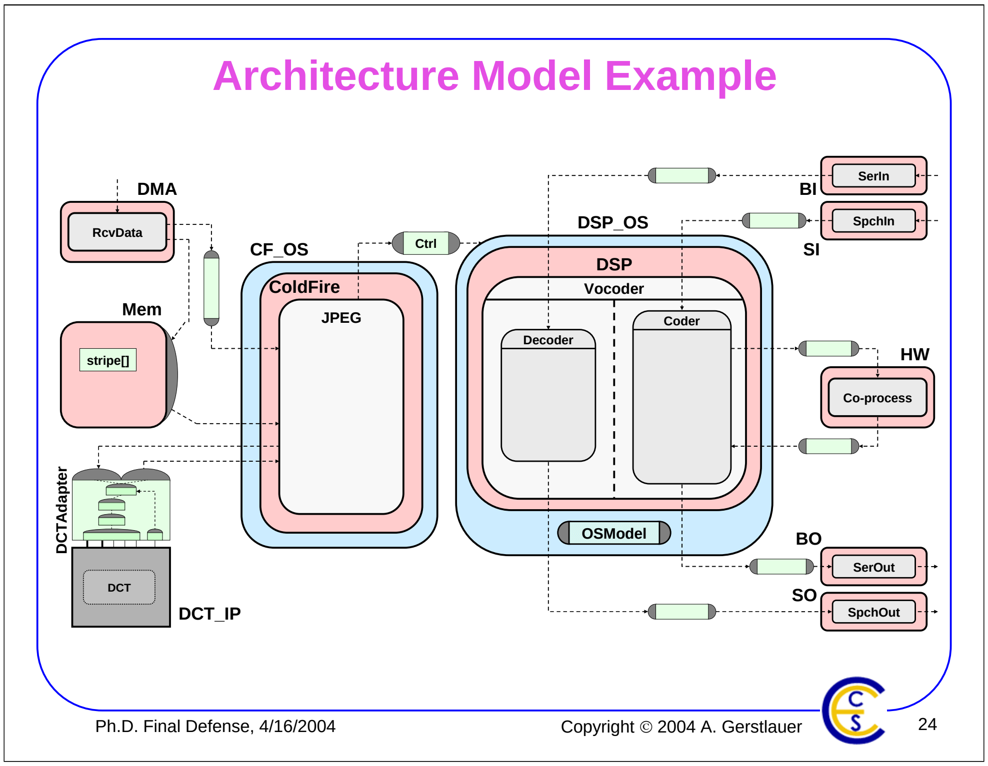


After computation design, communication design deals with the implementation of abstract communication channels over actual busses or other communication structures.

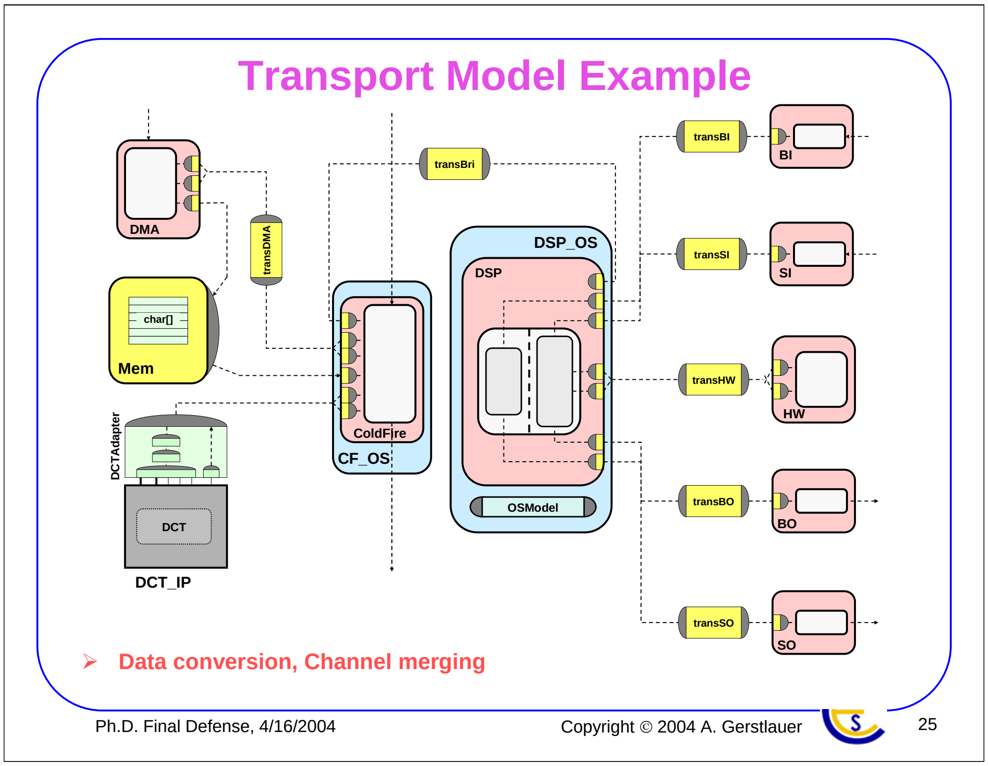
Similar to computation design, communication design consist of two parts. First, the topology of the communication network is defined and the end-to-end channels are merged into untyped byte streams and routed over the network of point-to-point logical links.

In the second part of communication design, logical point-to-point links between network stations are then implemented over shared physical media by grouping them into physical links and by implementing media, protocol and finally wire level interfaces for each physical link in each network station.

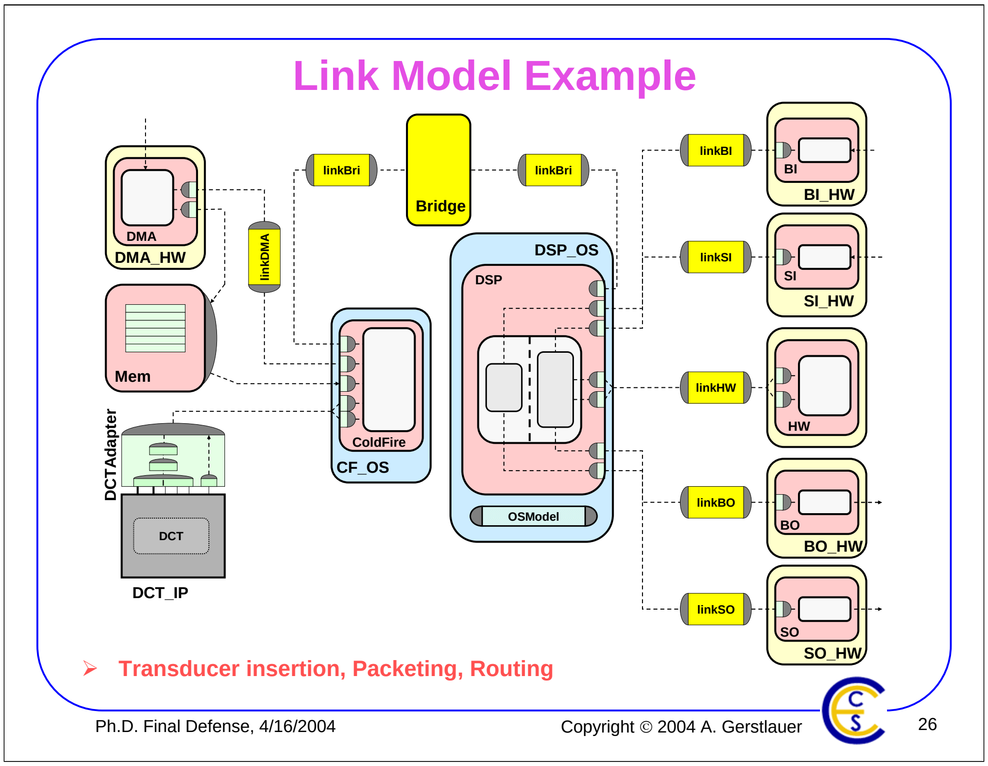
In addition to the final communication model that is handed off to the backend design task, communication design can output intermediate, transaction-level media access and protocol models. As will be shown later, abstract TLMs allow rapid design space exploration by trading off model accuracy and model complexity.



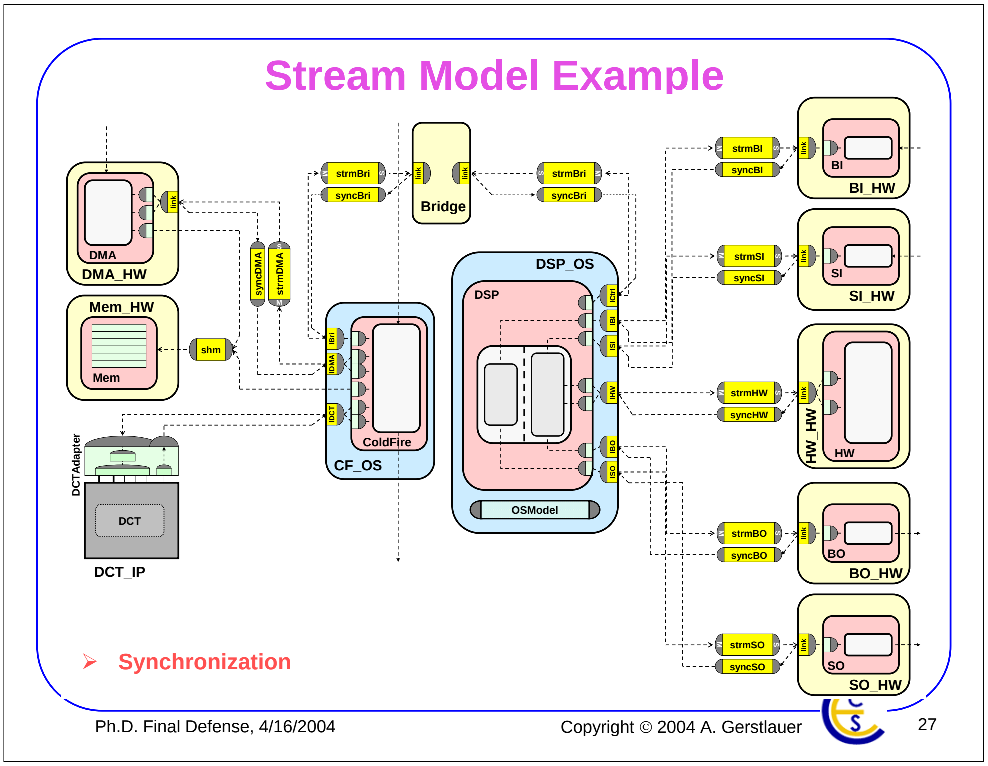
Recalling the architecture model at the output of computation design, communication design starts at this point.



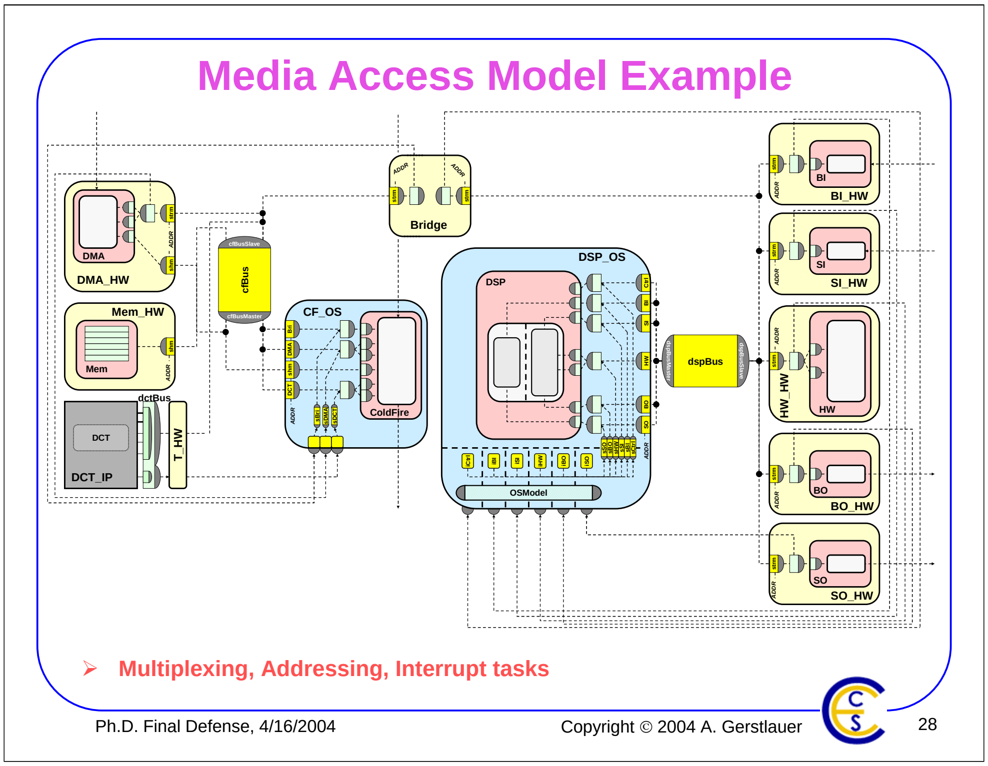
During channel streaming, presentation and session layers are inserted to implement conversion of abstract data types into network bytes and to merge channel into a set of streams between PEs. As part of data conversion, memory behaviors are refined to a byte-accurate representation of their data layout.



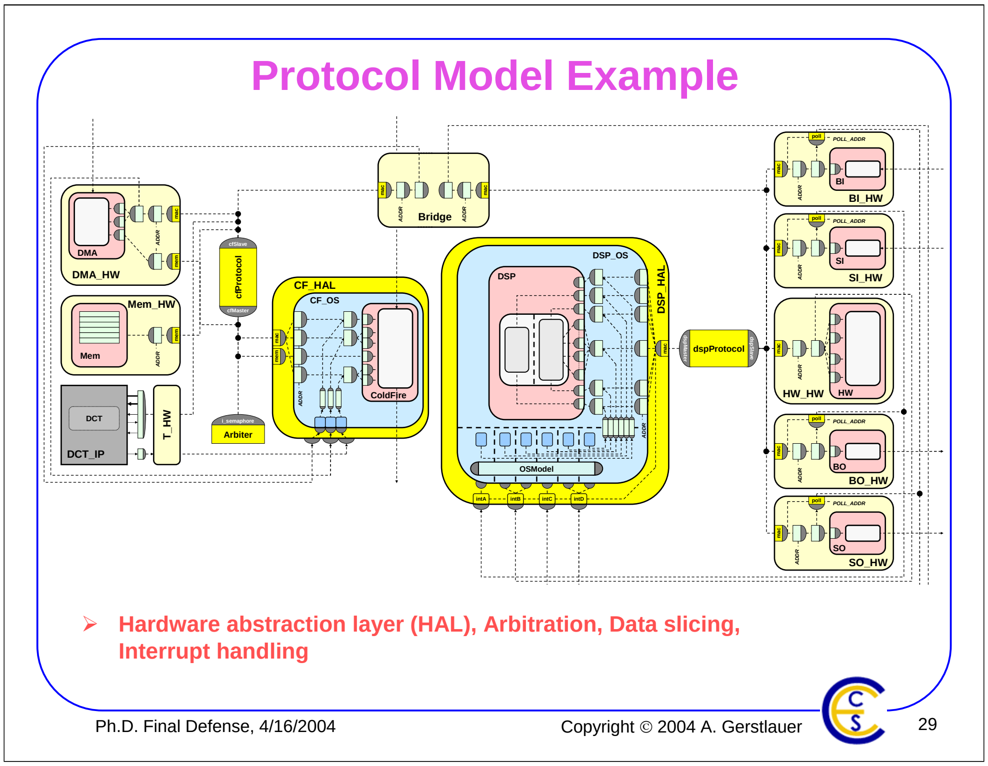
During network segmenting, transducers are inserted to divide and bridge the network into several segments, splitting end-to-end channels into point-to-point links as necessary. Inside PEs and transducers, transport and network layers are inserted to perform the necessary packeting and routing.



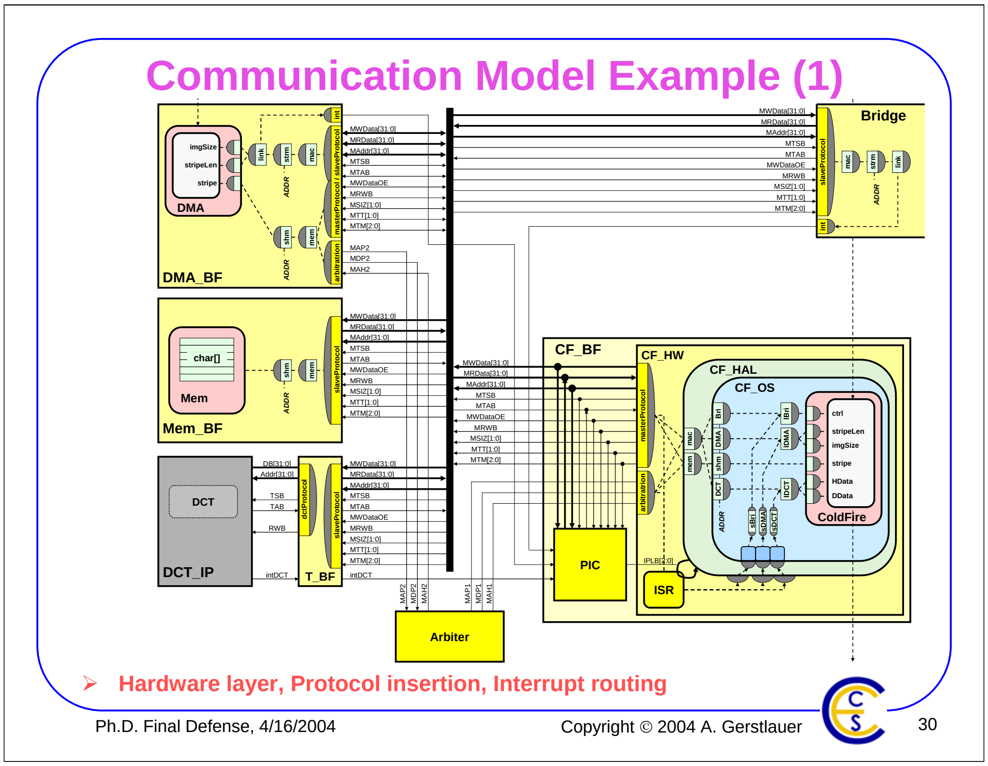
As a first step of link grouping, links are split into separate control and data streams based on the type of bus interface of each link. Link layers that implement synchronization over control channels around each data transaction are inserted.



In the second step of link grouping, data streams in each segment are then multiplexed over a shared medium channel. Stream layers that perform the necessary media addressing are inserted into the components. In addition, interrupt tasks that communicate with bus drivers through semaphores are inserted to implement control transactions.

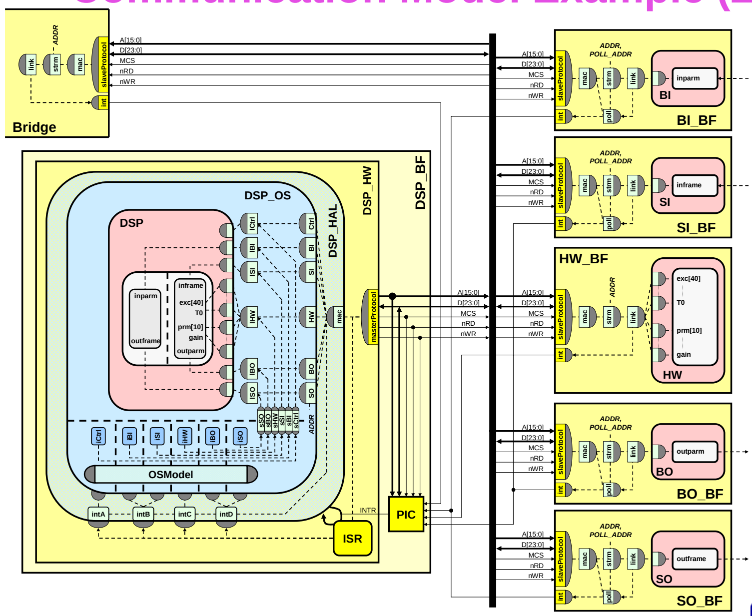


In the first step of implementing media interfaces, media access layers are inserted into components to implement arbitration through arbitration channels and slicing of data packets into bus words/frames transactions over protocol channels. For programmable PEs, media access layers become part of a newly added hardware abstraction layer that will mark the boundary between the PE's software and hardware. Finally, interrupt handlers are created inside the hardware abstraction layers to implement low-level control transactions including slave polling in case of interrupt sharing.



Finally, timing-accurate bus-protocol implementations are inlined into the system components, exposing the underlying bus wires. For programmable PEs, hardware models are inserted that accurately describe the PE's interrupt handling behavior. Finally, bus-functional arbiter and interrupt controller models are inserted and connected.

Communication Model Example (2)



Communication Model

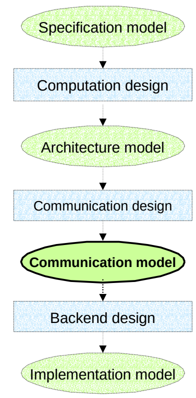
- System architecture
  - Computation & communication structure
  - Timed, bus-functional

$\text{Communication} = < PE \cup CE, W, c >$

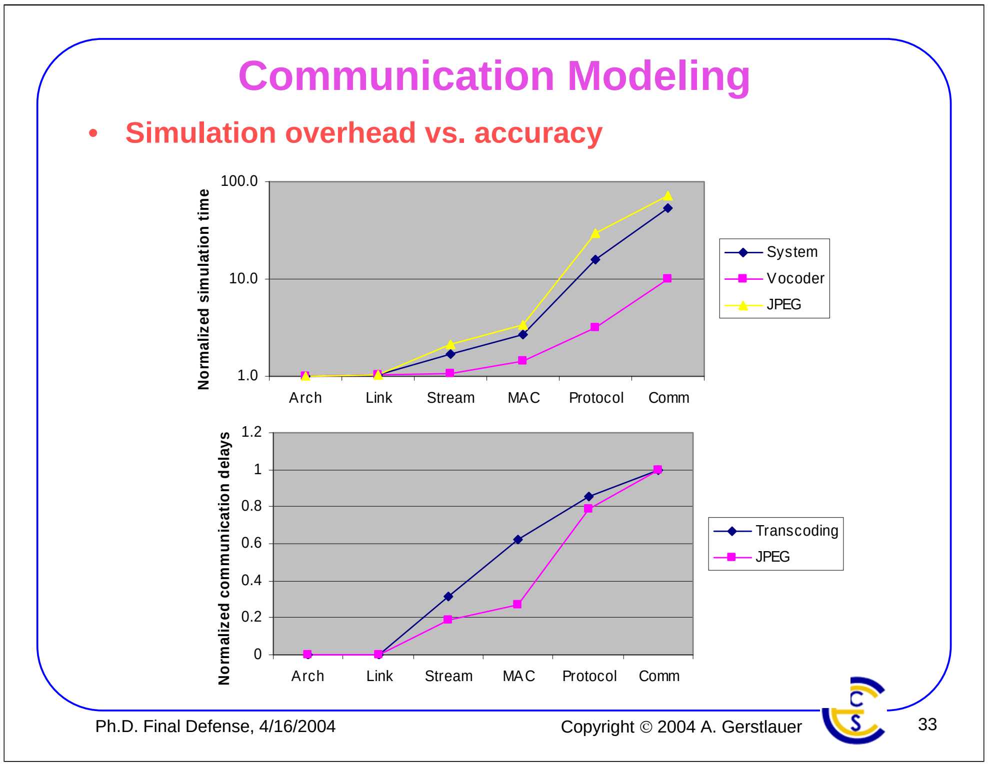
$PE = P \cup IP \cup M$:	set of processing elements
$CE = T \cup A \cup IC$:	set of communication elements
W :	set of bus wires
$c : \bigcup_{p \in (PE \cup CE)} O_p \mapsto W$	port mapping function

$\forall$ bus-functional processor $p \in P : p = < B_p, V_p, C_p, D_p, O_p, R_p >$

B_p :	set of behaviors
C_p :	set of local channels
D_p :	set of bus drivers
O_p :	set of PE ports
$R_p \subseteq B_p \times (C_p \cup D_p)$:	local connectivity relation



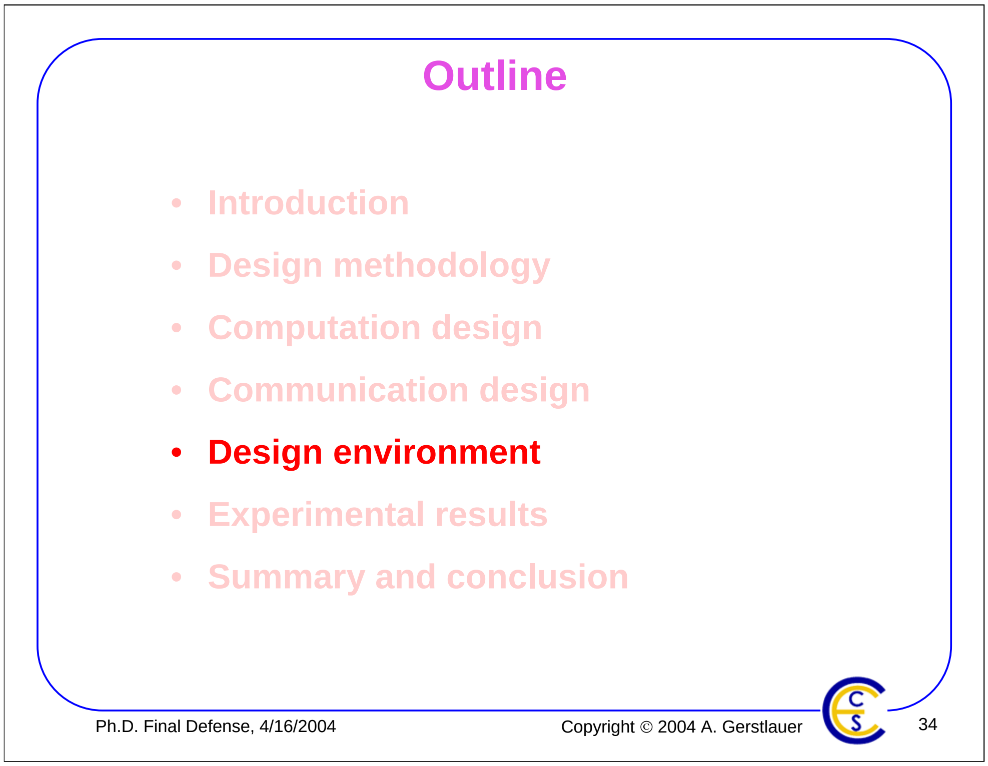
The result of communication design is the communication model. It is a bus-functional, timing-accurate description of the complete computation and communication system architecture. The communication model is a netlist of processing and communication elements connected via bus wires. Each bus-functional component in turn is described as a set of local behaviors, variables, channels, bus drivers and ports.



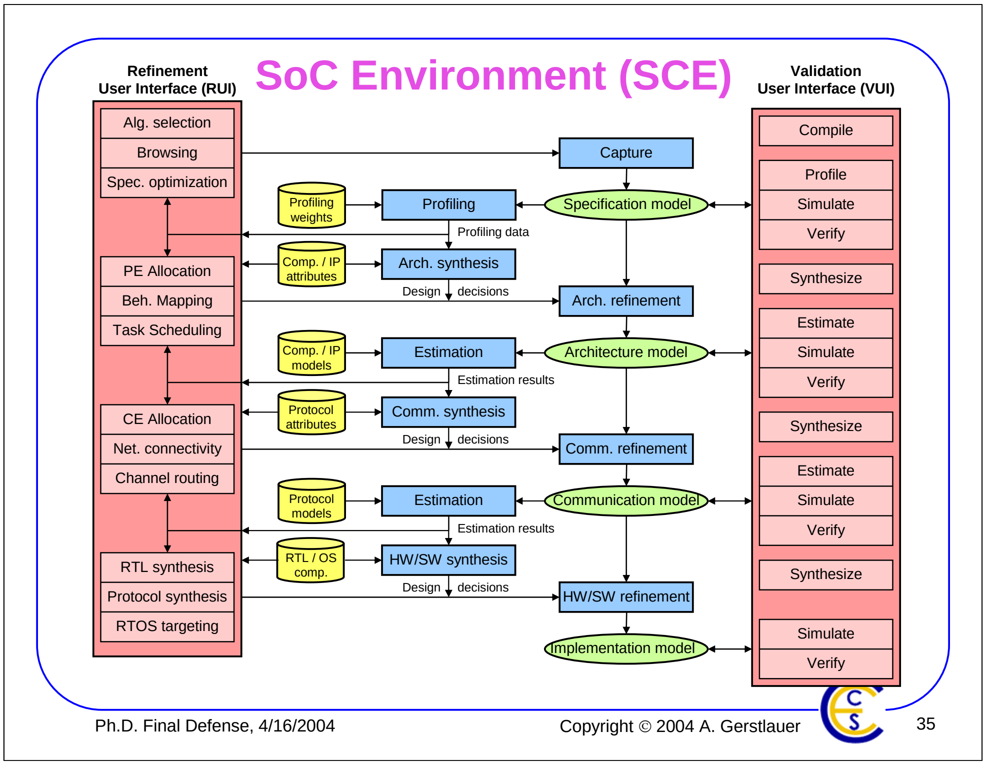
Benefits and trade-offs in terms of model complexities vs. model accuracies for communication models at different levels of abstraction are shown here. The graphs show simulation runtimes on a logarithmic scale and communication delays measured by setting computation delays to zero and normalized against the communication delay in the final implementation.

As can be expected, generally simulation runtimes grow exponentially whereas accuracies grow linearly with lower levels of abstraction, clearly demonstrating the benefits of high-level models. As can bee seen, the protocol model can provide up to 80% accuracy at significantly higher simulation speeds. The protocol model includes data slicing and bus arbitration needed to accurately model delays in the presence of interleaved transactions of multiple masters on the bus. On the other hand, if no arbitration is present, as in the case of the Vocoder subsystem, the MAC model can potentially provide relatively accurate data. In the Vocoder case, delay inaccuracies in the MAC model are introduced due to the fact that slave polling is not included in the MAC model. If the design does neither require arbitration nor slave polling, the MAC model would be even more accurate. Note that since the MAC model lumps several all bus transfers within a packet into a single transaction, simulation speeds are disproportionally higher compared to the protocol model.

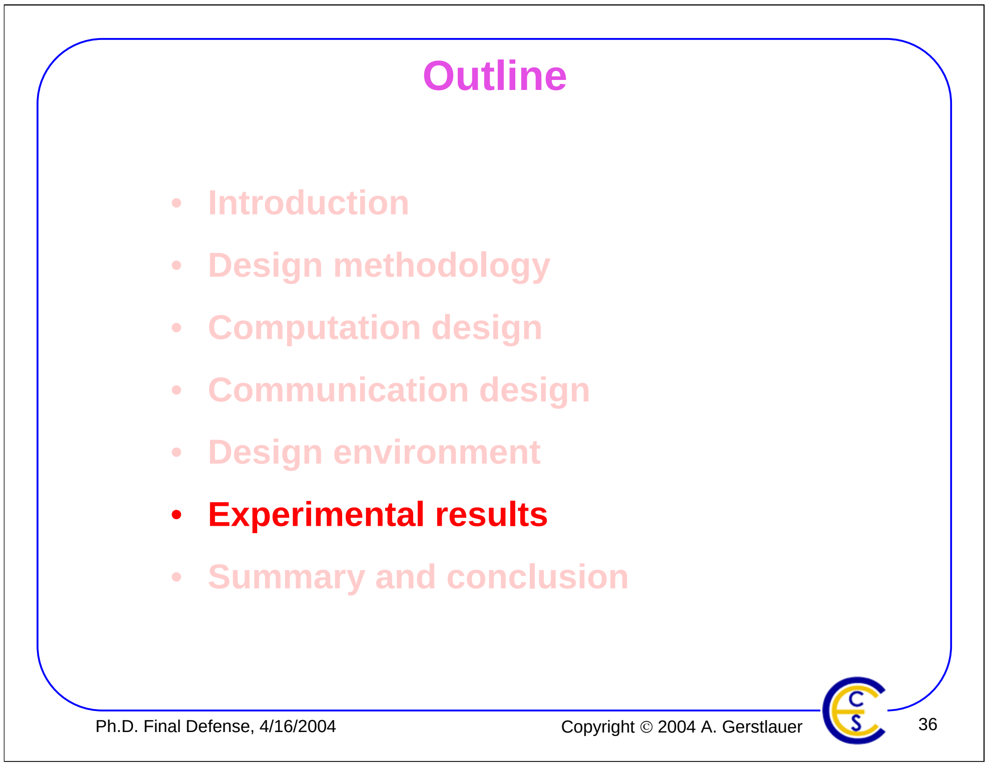
All in all, results confirm the choice of both MAC and protocol models for communication design space exploration depending on the selected target architecture.



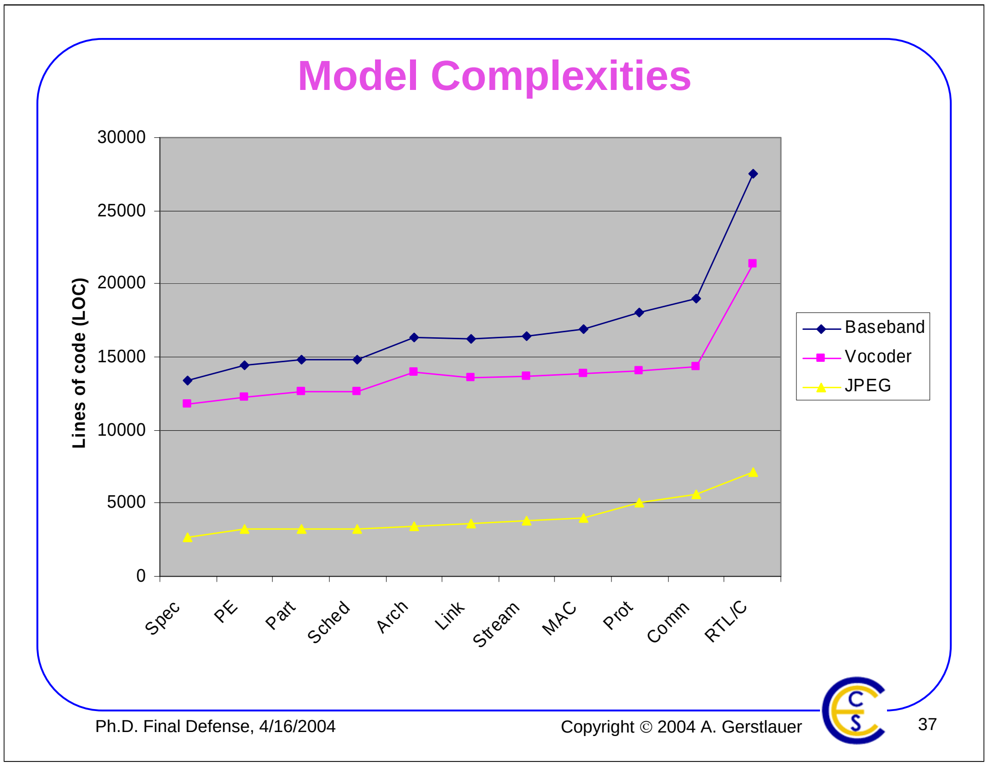
The design flow has been implemented in the form of the SoC design environment.



The overall architecture of the SoC design environment is shown here. As part of this work, the design environment's general framework including architecture, tool flow, databases, and interfaces has been developed. Tools for automatic model refinement have been integrated into the design environment, enabling generation of complete designs within minutes. The design environment supports automated decision making through a plug-in mechanism such that the design can selectively apply algorithms to all or part of a design at any time. Finally, graphical user interfaces for model visualization and decision entry have been developed that aid and steer the designer in the exploration process.

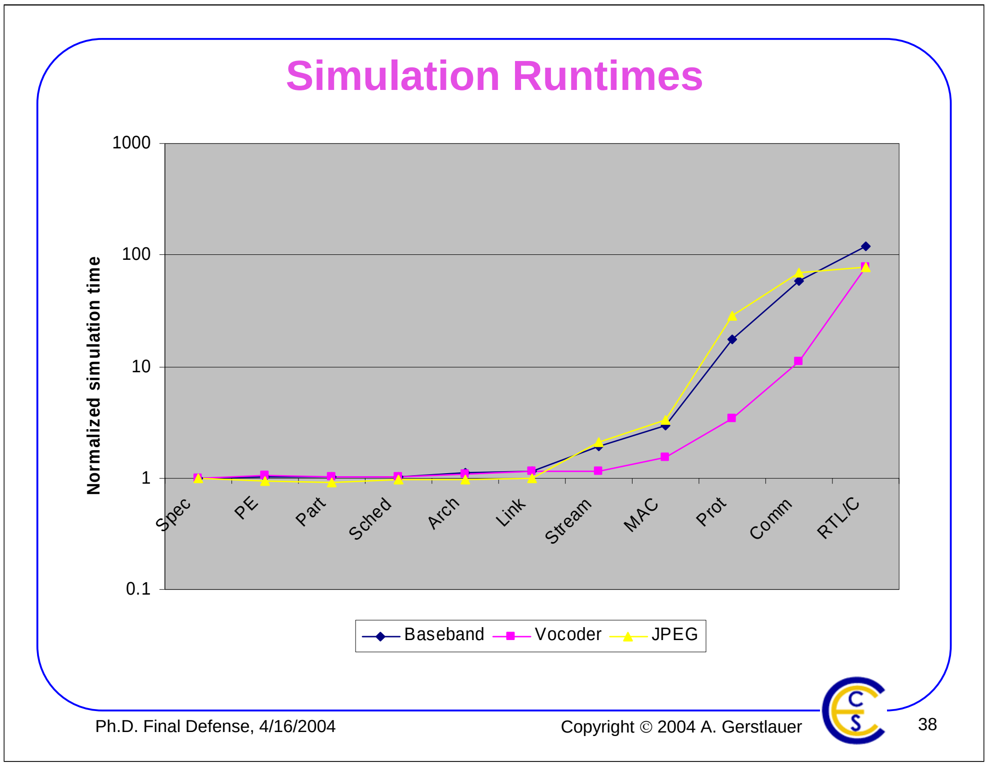


In the following, I will show results obtained by applying the design flow to the example design presented throughout this presentation. In general, results have been obtained for the overall system and for both Vocoder and JPEG encoder subsystems design separately.

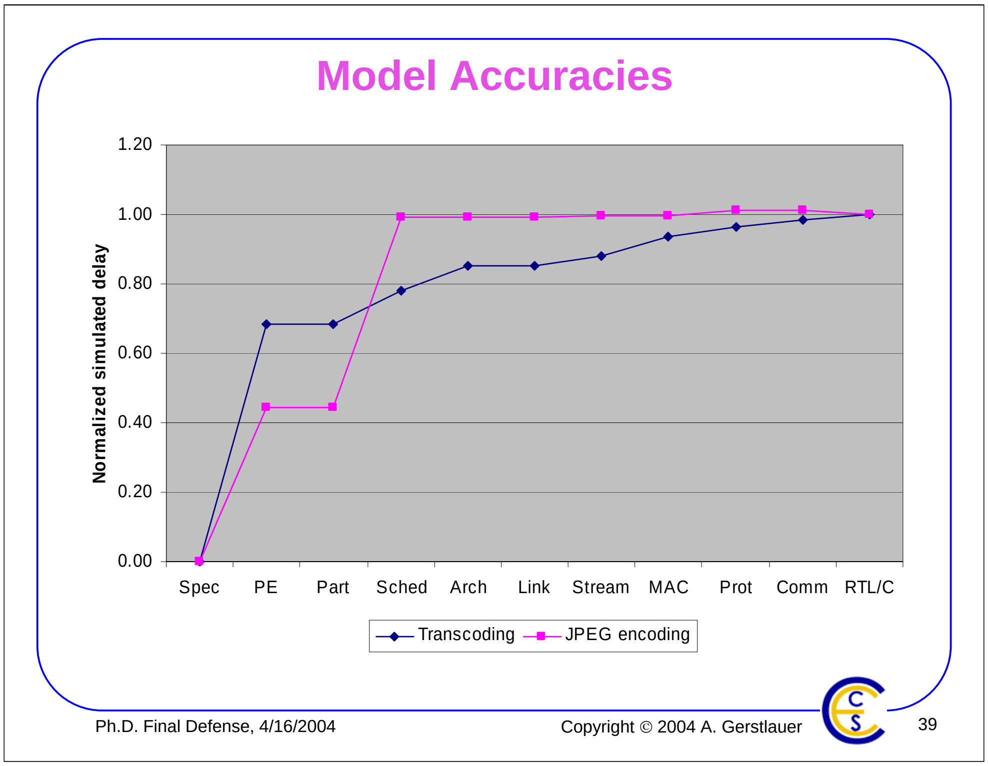


Model complexities as measure by the number of lines of code for models of different designs at different levels are shown here. As expected, models generally grow linearly with lower levels of abstraction. At the RTL level, however, model sizes grow exponentially due to the high overhead necessary for cycle-accurate state machine modeling where growth depends to a large extend on the size of the hardware part.

Note that model growth does not depend on the size of the original specification. Rather, model complexities grow depending on the complexity of the target architecture and hence the necessary implementation detail to be added.



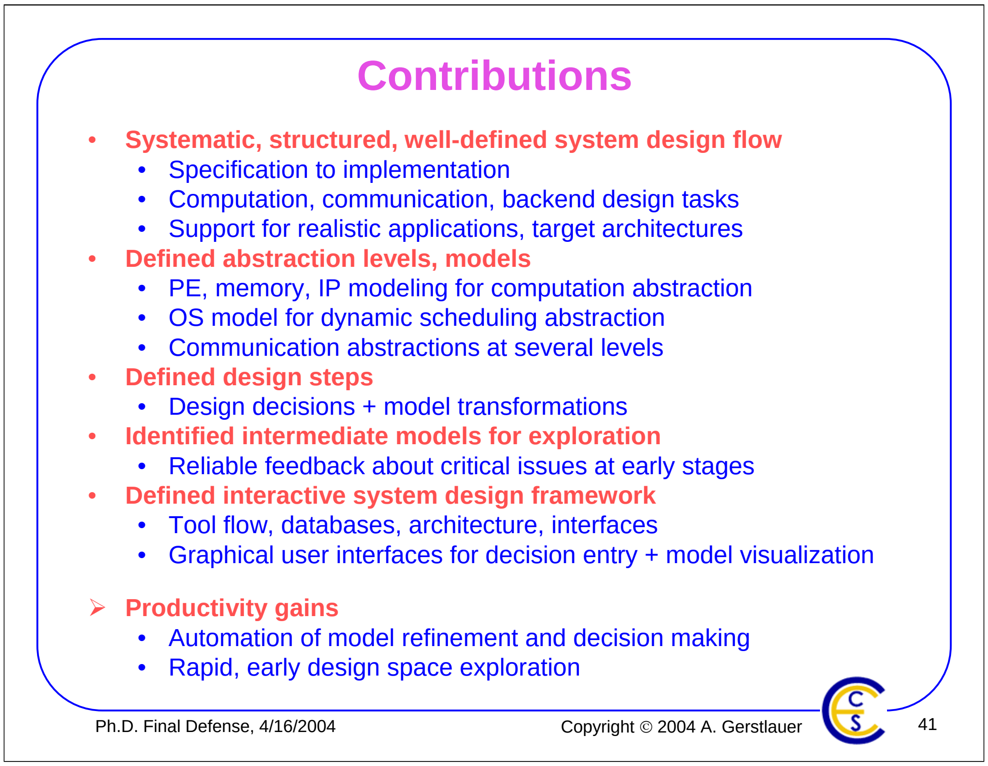
In terms of simulation overhead, it can be seen that all throughout computation design, almost no additional overhead is introduced. Only in the link design phase, simulation runtimes start growing exponentially as explained earlier during communication modeling. Again, exponential growth of runtimes during backend design depends exclusively on the relative size of the hardware part in the design.



Finally, looking at accuracies of models at different levels of abstraction, results confirm the choice of the architecture model as intermediate model for exploration, especially considering the fact that no additional simulation overhead is introduced up to this point. For the designs shown here, the architecture model is over 80% accurate. PE and partitioned models are generally not accurate enough as they ignore the effects of sequential execution on PEs.

Outline

- Introduction
- Computation design
- Communication design
- Design environment
- Experimental results
- **Summary and conclusion**



In summary, the main contribution of this work is the definition of a complete system design flow in a structured, systematic manner. Starting from an abstract, functional specification, a cycle-accurate implementation is derived through computation, communication and backend design tasks. The flow supports a wide variety of realistic applications and target architectures.

We defined abstraction levels and corresponding design models breaking the design flow into individual steps. PE, memory, IP and OS models for computation abstraction have been developed. Communication abstractions at several levels have been defined.

For each design step, necessary design decisions and model transformations have been defined. Furthermore, intermediate models for reliable, rapid and early design space exploration have been identified.

The design flow has been implemented in the form of a SoC design environment. The general framework of the design environment including tool flow, databases, architecture and interfaces has been defined. Furthermore, graphical user interfaces for decision entry and model visualization have been developed.

In conclusion, following this design flow, required productivity gains can be achieved. Steps have been defined such that decision making and model refinement can be automated. Together with design automation, abstract models at high levels enable rapid exploration of large parts of the design space in short amounts of time.

Selected Publications

- **Books**
  - A. Gerstlauer, R. Dömer, J. Peng, D. Gajski, "***System Design: A Practical Guide with SpecC***", Kluwer, 2001.
  - D. Gajski, J. Zhu, R. Dömer, A. Gerstlauer, S. Zhao, "***SpecC: Specification Language and Methodology***", Kluwer, 2000.
- **Book chapters**
  - A. Gerstlauer, H. Yu, D. Gajski, "**RTOS Modeling for System-Level Design**", *Embedded Software for SoC*, Kluwer, 2003.
  - A. Rettberg, F. Rammig, A. Gerstlauer, D. Gajski, W. Hardt, B. Kleinjohann, "**The Specification Language SpecC within the PARADISE Design Environment**", *Architecture and Design of Distributed Embedded Systems*, Kluwer, 2001.
- **Conference Papers**
  - L. Cai, A. Gerstlauer, D. Gajski, "**Retargetable Profiling for Rapid, Early System-Level Design Space Exploration**", *DAC* 2004.
  - A. Gerstlauer, H. Yu, D. Gajski, "**RTOS Modeling for System-Level Design**", *DATE* 2003.
  - A. Gerstlauer, D. Gajski, "**System-Level Abstraction Semantics**", *ISSS* 2002.
  - W. Mueller, R. Dömer, A. Gerstlauer, "**The Formal Execution Semantics of SpecC**", ISSS 2002.
  - A. Gerstlauer, S. Zhao, D. Gajski, A. Horak, "**SpecC System-Level Design Methodology Applied to the Design of a GSM Vocoder**", SASIMI 2000.

Backup Slides

Behavior Partitioning

- **Design decisions**
  - PE allocation and selection
    - $PE =$ set of $(name, type)$ tuples
  - Behavior mapping
    - mapping function $m_b : B \mapsto PE$
- **Model transformations**
  - PE layer
    - Additional layer of behavior hierarchy representing PEs
  - Grouping
    - Group behaviors under PEs according to mapping
  - Synchronization
    - Insert synchronization to preserve transition semantics
  - Timining refinement
    - Annotate behaviors with estimated execution delays

Variable Partitioning

- **Design decisions**
  - Memory allocation and selection
    - M = set of $(name, type)$ tuples
  - Variable mapping
    - mapping function $m_v : V_s \subseteq V \mapsto M$
- **Model transformations**
  - Memory layer
    - Insert behaviors representing shared memories
  - Grouping
    - Group global variables under shared memories according to mapping
  - Message passing
    - Distribute unmapped global variables, insert message passing
  - Memory accesses
    - Create memory interface, update shared variables accesses

Static Scheduling

- Design decisions
  - Behavior order
    - $\forall b \in B_s, B_s \subseteq B : \text{schedule } S_b = \text{totally ordered set of children}$
- Model transformations
  - Serialization
    - Sequentialize concurrent behavior compositions
  - Flattening
    - Move children into parent behavior as requested
  - Reordering
    - Arrange behaviors in selected execution order

Dynamic Scheduling

- **Design decisions**
  - Scheduling algorithm selection
    - OS selection function $os : PE \mapsto$ set of algorithms OS
  - Task priority assignment
    - task priority function $p : B_t \subseteq B \mapsto Z^+$
- **Model transformations**
  - OS layer
    - Additional OS layer around programmable PEs
    - Insert abstract OS model for selected scheduling strategy
  - Task creation
    - Turn concurrent behaviors into OS tasks
  - Task refinement
    - Replace delay primitives
  - Synchronization refinement
    - Replace event handling primitives

Channel Streaming

- **Design decisions**
  - Network byte layout
    - layout function l over data types $d \in D$:
      $D \mapsto Z^* \times Z^* \times \{b, l\},\ l(d) = (size, alignment, endianess)$
  - Channel merging
    - merging function m_c : set of channels $C_s \mapsto$ set of streams S
- **Model transformations**
  - Presentation layer
    - Conversion of abstract data types into network bytes
    - Memory data byte layout
  - Session layer
    - Merge channels into message streams

Network Segmenting

- **Design decisions**
  - Transducer allocation
    - T = set of $(name, type)$ tuples
  - Channel routing & packeting
    - $\forall$ stream $s \in S$: route R_s = ordered set of hops $r \in (PE \cup T)$
    - packet function $p : S \mapsto Z^+$, $p(s) = packet\ size$
- **Model transformations**
  - Transport layer
    - Splitting of message streams into packet streams
    - Flow control, error correction
  - Network layer
    - Insert transducers and links
    - Routing of packets over links between PEs and transducers

Link Grouping

- **Design decisions**
  - Bus/protocol allocation
    - BUS = set of $(name, type)$ tuples
  - Station connectivity
    - connectivity relation $N \subseteq (PE \cup T) \times BUS$
    - connection type function $if : N \mapsto$ interface types IF
  - Link parameters
    - $\forall l \in$ links L, parameter function m:
      $L \mapsto N \times N \times Z^{+} \times Z^{*}, m(l) = (src, dst, addr, intr)$
- **Model transformations**
  - Link layer
    - Splitting of links into control and data transactions
  - Stream layer
    - Multiplexing of data over media transaction via media addressing
    - Implementation of interrupt tasks for control transactions

Media Interfacing

- **Design decisions**
  - Arbiter allocation, bus master priority assignment
    - A = set of $(name, type)$ tuples, connectivity : $A \mapsto BUS$
    - bus master priority function $a : MASTER \subseteq N \mapsto Z^*$
  - Interrupt controller allocation, bus slave interrupt assgn.
    - IC = set of $(name, type)$ tuples, connectivity $IC \mapsto PE$
    - bus slave interrupt function $i : SLAVE \subseteq N \mapsto$ set of interrupts

- **Model transformations**
  - Media access layer, hardware abstraction layer (HAL)
    - Slicing of data packets into media words/frames, media arbitration
    - Implementation of interrupt handlers, slave polling
  - Protocol layer, hardware layer
    - Protocol transaction timing for sampling/driving wires
    - Insert programmable PE interrupt hardware model
    - Insert, connect arbiters and interrupt controllers