HW/SW Co-Design for Embedded Systems

P. Giusto, A. Jurecska
Magneti Marelli, Torino, Italy

K. Suzuki
Hitachi Res. Lab., Tokio, Japan

M. Chiodo
Alta Group, Sunnyvale, CA

L. Lavagno
Politecnico di Torino, Italy
Cadence Berkeley Labs, CA

F. Balarin, E. Sentovich
Cadence Berkeley Labs, CA

H. Hsieh, A. Sangiovanni-Vincentelli
University of California, Berkeley, CA

© ASV et al. 1996
Outline

- Reactive Real-Time Embedded Systems
- Specification Models and Languages
  - Data Flow
  - Extended Finite State Machines
- Proposed Design Methodology
  - System Specification
  - Validation
  - System Partitioning
  - Software, Hardware and Interface Synthesis
  - Real-time Operating System and Scheduling
- Summary
Embedded Systems

- An embedded system
  - uses a computer to perform some function, but
  - is not used (nor perceived) as a computer
- Software is used for features and flexibility
- Hardware is used for performance
- Typical characteristics:
  - it performs a single function
  - it is part of a larger (controlled) system
  - cost and reliability are often the most significant aspects
Embedded System Applications

- Consumer electronics
  (microwave oven, camera, ...)
- Telecommunication switching and terminal equipment
  (cellular phone, ...)
- Automotive, aero-spatial
  (engine control, anti-lock brake, ...)
- Plant control and production automation
  (robot, plant monitor, ...)
- Defense
  (radar, intelligent weapon, ...)
Reactive Real-time Systems

- Reactive Real-Time Systems
  - “React” to external environment
  - Maintain permanent interaction
  - Ideally never terminate
  - Are subject to external timing constraints (real-time)
Several protocols have been proposed for the implementation of in-vehicle networks. Among them:

- CAN, VAN (Europe)
- J1850 (USA)
Automotive networks come in three classes:

<table>
<thead>
<tr>
<th>Class</th>
<th>Application</th>
<th>Max. Latency</th>
<th>Bit Rate</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>Body electronics</td>
<td>20-50 ms</td>
<td>&lt; 10 Kbit/s</td>
</tr>
<tr>
<td>B</td>
<td>Information sharing</td>
<td>1-10 ms</td>
<td>10-20 Kbit/s</td>
</tr>
<tr>
<td>C</td>
<td>Real-time controls</td>
<td>&lt; 1 ms</td>
<td>0.125-1 Mbit/s</td>
</tr>
</tbody>
</table>

The implementation can be different for different classes of applications. For example:

- class A - CPU can handle communication + application
- class C - communication IC required
Embedded Controller Example:
Engine Control Unit (ECU)

Task: control the torque produced by the engine by timing fuel injection and spark

- Major constraints:
  Low fuel consumption
  Low exhaust emission
Engine Control Unit (ECU) - 2

Task: control injection time (3 sub-tasks)

- compute air flow
  - air flow
  - throttle position
  - engine speed
  - air temperature
  - air pressure

- compute injection time
  - look-up table
  - engine temperature

- drive actuators
  - injection time
  - PWM signals

© ASV et al. 1996
Engine Control Unit (ECU) - Option 1

CPU has to:
- process input data
- compute outputs
- control actuators

- Relatively easy to design
- May not meet timing requirements
Engine Control Unit (ECU) - Option 2

- CPU processes input data and computes outputs
- FPGA controls actuators

Analog inputs

A/D

Digital inputs

16 bit CPU

FPGA

Actuations (PWM)
Engine Control Unit (ECU) - Option 3

- DSP processes input data
- CPU computes outputs
- FPGA controls actuators

Analog inputs → A/D → DSP → 8 bit CPU → FPGA → Actuations (PWM)

Digital inputs
RRTES Implementations

- Possibly contain both hardware and software (ASIC plus embedded software)
- Past Design Methodology
  - Software implementation:
    - timing \(\rightarrow\) offload to hardware
  - Specify and design hardware and software separately
Problems with Past Design Method

- Lack of unified hardware-software representation
- Partitions are defined *a priori*
  - Can't verify the entire system
  - Hard to find incompatibilities across HW-SW boundary
- Lack of well-defined design flow
  - Time-to-market problems
  - Specification revision becomes difficult

Need Hardware-Software Co-Design

© ASV et al. 1996
Hardware/Software Co-Design Goals and Requirements

- Unified design approach
  - Facilitates system specification
  - Easy HW-SW trade-off evaluation
  - Flexible HW-SW partitioning
- Implementation Independent
  - Stress system design issues
  - Allow different hardware and software styles
- Design/Implementation Verification
  - Formal Verification
  - Simulation
- Automatic Hardware and Software Synthesis
Outline

- Reactive Real-Time Embedded Systems
- Specification Models and Languages
  - Data Flow
  - Extended Finite State Machines
- Proposed Design Methodology
  - System Specification
  - Validation
  - System Partitioning
  - Software, Hardware and Interface Synthesis
  - Real-time Operating System and Scheduling
- Summary
System Specification Models

- Main purpose: provide clear and unambiguous description of system \textit{function}
  - documentation of initial design process
  - allow the application of Computer Aided Design:
    - design space exploration
    - partitioning
    - synthesis
    - validation
    - testing
  - ideally should not constrain the \textit{implementation}
System Specification Models

- Distinguish between *models* and *languages*
  (a language implies one or more models)
- Main models for embedded systems:
  - Data Flow Diagrams (Petri Nets)
  - Finite State Machines
  - Software models
  - Hardware simulation models
System Specification Models

- Model choice depends on:
  - Application domain
    - E.g. data flow for digital signal processing,
      finite state machines for control,
      simulation engine for hardware, ...

- Language choice depends on:
  - Available tools
  - Personal taste and/or company policy
  - Underlying model
    (the language must have a semantics in the chosen model)
Control versus Data Flow

- Fuzzy distinction, yet useful for:
  - specification (language, model, ...)
  - synthesis (scheduling, optimization, ...)
  - validation (simulation, formal verification, ...)

- Rough classification:
  - control:
    - don’t know when data arrive (quick reaction)
    - time of arrival often matters more than value
  - data:
    - data arrive in regular streams (samples)
    - value matters most
Control versus Data Flow

- Specification, synthesis and validation methods emphasize:
  - for control:
    - event/reaction relation
    - response time
      (Real Time scheduling for deadline satisfaction)
    - priority among events and processes
  - for data:
    - functional dependency between input and output
    - memory/time efficiency
      (data flow scheduling for efficient pipelining)
    - all events and processes are equal
Outline

- Reactive Real-Time Embedded Systems
- Specification Models and Languages
  - Data Flow
  - Extended Finite State Machines
- Proposed Design Methodology
  - System Specification
  - Validation
  - System Partitioning
  - Software, Hardware and Interface Synthesis
  - Real-time Operating System and Scheduling
- Summary
Data Flow Networks

- Collection of nodes (processes) connected by FIFOs
- Typical domains of application:
  - Digital Signal Processing
  - Performance estimation (queueing models)
- Very different models depending on node interpretation:
  - Uninterpreted
    (classical Petri Nets)
  - Arithmetic operators
    (classical DFGs)
  - Complex operators
    (queueing models, colored Petri Nets)
Data Flow Example

Process 1:
forever {
    x = 0; y = 0;
    while (x < 10) {
        y = y + get_data (B);
        x = x + 1;
    }
    put_data (A, y / 10);
}

Process 2:
forever {
    a = get_data (A);
    if (a < 0)
        put_data (C, a);
    else
        put_data (D, -a);
}
Data Flow Primitives

- $\mathbb{T}$: initial token (with value)
- $0$: constant
- $\mathbb{T} \mathbb{F}$: deterministic split/merge
- $\rightarrow$: data dependency (FIFO)
- $\oplus$ +1: operator
- $\mathbb{C}$: communication channel
- $\triangle$: flow duplication
Kahn networks introduced to develop semantics for λ calculus (’74):
- nodes communicate via unbounded FIFO channels
- nodes must block when reading from a FIFO (cannot test and continue)
- nodes can choose which FIFO(s) they read from and write to

- Originally used to program data flow computers (Dennis ‘75)
- Recently used to specify algorithms for Digital Signal Processors (Lee ‘87, Buck ‘93)
Properties of Data Flow Networks

- Inherently *concurrent, asynchronous* computation model, *but...*
- Kahn Data Flow networks are *determinate*
  - the stream of values produced by each node does not depend on the execution (“firing”) order of the nodes
- Strong limitation (blocking read) implies strong result
- FSMs will need *synchronicity* to achieve the same objective
- Similar, but not identical to Petri nets (Petri ‘62)
Petri Nets

- Very powerful *uninterpreted* model
- Bipartite graph (transitions and places)
- Describes explicitly
  - causality
  - concurrency
  - choice
- Does not describe
  - computation
  - reason for choice
    (non-determinism)

© ASV et al. 1996
Petri Nets and Data Flow

● Similarities:
  ◆ distributed state (tokens in places, data in FIFOs)
  ◆ firing nodes move tokens around

● Differences:
  ◆ PN transitions cannot choose which successor place to mark,
    DF nodes can
  ◆ PN transitions can share predecessor places,
    DF nodes cannot
  ◆ uninterpreted PNs are (relatively) easy to analyze,
    DF networks are Turing-equivalent (undecidability)
Summary of Data Flow Networks

- **Advantages:**
  - Easy to use (graphical languages)
  - Powerful algorithms for
    - synthesis (scheduling and allocation)
    - verification (only PNs)
  - Explicit concurrency
- **Disadvantages:**
  - Efficient synthesis only for restricted models
    (no input or output choice)
  - Cannot describe reactive control (blocking read)
Outline

- Reactive Real-Time Embedded Systems
- Specification Models and Languages
  - Data Flow
  - Extended Finite State Machines
- Proposed Design Methodology
  - System Specification
  - Validation
  - System Partitioning
  - Software, Hardware and Interface Synthesis
  - Real-time Operating System and Scheduling
- Summary

© ASV et al. 1996
Finite State Machines

- Typical domains of application:
  - control functions
  - protocols (telecom, computers, ...)
- Different communication mechanisms:
  - synchronous
    - classical FSMs, Moore ‘64, Kurshan ‘90
  - asynchronous
    - CCS, Milner ‘80; CSP, Hoare ‘85
Informal specification:

*If the driver*

*turns on the key, and*

*does not fasten the seat belt within 5 seconds*

*then an alarm beeps*

*for 5 seconds, or*

*until the driver fastens the seat belt, or*

*until the driver turns off the key*
If no condition is satisfied, implicit self-loop in the current state
FSM Definition

- FSM = (I, O, S, r, δ, λ)
- I = { KEY_ON, KEY_OFF, BELT_ON, END_TIMER_5, END_TIMER_10 }
- O = { START_TIMER, ALARM_ON, ALARM_OFF }
- S = { OFF, WAIT, ALARM }
- r = OFF
- \( \delta : 2^I \times S \rightarrow S \)
  - e.g. \( \delta(\{\text{KEY_OFF}\}, \text{WAIT}) = \text{OFF} \)
- \( \lambda : 2^I \times S \rightarrow 2^O \)
  - e.g. \( \lambda(\{\text{KEY_ON}\}, \text{OFF}) = \{\text{START_TIMER}\} \)

Set of all subsets of I (implicit “and”)

All other inputs are implicitly absent
Non-deterministic FSMs

- $\delta$ and $\lambda$ may be *relations* instead of *functions*:
  - $\delta \subseteq 2^I \times S \times S$
    - e.g. $\delta(\{\text{KEY\_OFF, END\_TIMER\_5}\}, \text{WAIT}) = \{\{\text{OFF}\}, \{\text{ALARM}\}\}$
  - $\lambda \subseteq 2^I \times S \times 2^O$

- Non-determinism can be used to describe:
  - an unspecified behavior
    - (incomplete specification)
  - an unknown behavior
    - (environment modeling)
● E.g. error checking first partially specified:

\[
\begin{align*}
0 & \rightarrow 1 & \rightarrow \ldots & \rightarrow 7 & \rightarrow 8 \\
\text{BIT or not BIT } & \rightarrow & \text{BIT or not BIT } & \rightarrow \text{BIT or not BIT } & \rightarrow \text{ERR} \\
\text{SYNC } & \rightarrow & \text{BIT } & \rightarrow & \text{not BIT } & \rightarrow \text{BIT } & \rightarrow \text{not BIT } & \rightarrow \text{ERR} \\
\text{not BIT } & \rightarrow & \text{BIT } & \rightarrow & \text{not BIT } & \rightarrow \text{BIT } & \rightarrow \text{not BIT } & \rightarrow \text{ERR} \\
\text{BIT } & \rightarrow & \text{not BIT } & \rightarrow & \text{BIT } & \rightarrow & \text{not BIT } & \rightarrow \text{ERR} \\
\text{not BIT } & \rightarrow & \text{BIT } & \rightarrow & \text{not BIT } & \rightarrow \text{BIT } & \rightarrow \text{not BIT } & \rightarrow \text{ERR} \\
\text{SYNC } & \rightarrow & \text{BIT } & \rightarrow & \text{not BIT } & \rightarrow \text{BIT } & \rightarrow \text{not BIT } & \rightarrow \text{ERR} \\
\text{BIT } & \rightarrow & \text{not BIT } & \rightarrow & \text{BIT } & \rightarrow & \text{not BIT } & \rightarrow \text{ERR} \\
\text{not BIT } & \rightarrow & \text{BIT } & \rightarrow & \text{not BIT } & \rightarrow \text{BIT } & \rightarrow \text{not BIT } & \rightarrow \text{ERR} \\
\text{SYNC } & \rightarrow & \text{BIT } & \rightarrow & \text{not BIT } & \rightarrow \text{BIT } & \rightarrow \text{not BIT } & \rightarrow \text{ERR} \\
\end{align*}
\]

● Then completed as even parity:

● Could be implemented as CRC later
NDFSM: unknown behavior

- Modeling the environment
- Useful to:
  - optimize (don’t care conditions)
  - verify (exclude impossible cases)
- E.g. driver model:

  ![Diagram](s0 => KEY_ON or KEY_OFF or BELT_ON)

- Can be refined
  E.g. introduce timing constraints
  (minimum reaction time 0.1 s)
NDFSM: time range

- Special case of unspecified/unknown behavior, but so common to deserve special treatment for efficiency
- E.g. undetermined delay between 6 and 10 s
NDFSMs and FSMs

- Formally FSMs and NDFSMs are equivalent (Rabin-Scott construction, Rabin ‘59)
- In practice, NDFSMs are often more compact (exponential blowup for determinization)
● Bridle complexity via hierarchy: *FSM product yields an FSM*

● Fundamental hypothesis:
  all the FSMs change state together (*synchronicity*)

● System state = Cartesian product of component states
  (state explosion may be a problem...)

● E.g. seat belt control + timer

```
START_TIMER =>

SEC =>

END_10_SEC
```

```
START_TIMER =>

SEC =>

END_5_SEC
```

```
SEC =>

SEC =>

SEC =>

SEC =>
```

© ASV et al. 1996
KEY_ON and START_TIMER \implies START_TIMER

must be coherent

OFF, 0 \rightarrow WAIT, 1

not SEC and (KEY_OFF or BELT_ON) \implies

not SEC and (KEY_OFF or BELT_ON) \implies

SEC and (KEY_OFF or BELT_ON) \implies

OFF, 1

WAIT, 2

SEC and (KEY_OFF or BELT_ON) \implies

OFF, 2

etc. etc.
FSM Composition

● product of FSM₁ and FSM₂
● I = I₁ ∪ I₂
● O = O₁ ∪ O₂
● Assume e.g. that o₁ ∈ I₂, o₁ ∈ O₁ (communication)
● S = S₁ × S₂
● δ e λ are such that, e.g., for each pair:
  ◆ δ₁(\{i₁\}, s₁) = t₁, \ λ₁(\{i₁\}, s₁) = \{o₁\}
  ◆ δ₂(\{i₂, o₁\}, s₂) = t₂, \ λ₁(\{i₂\}, s₂) = \{o₂\}
we have:
  ◆ δ(\{i₁, i₂, o₁\}, (s₁, s₂)) = (t₁, t₂)
  ◆ λ(\{i₁, i₂, o₁\}, (s₁, s₂)) = \{o₁, o₂\}
Summary of Finite State Machines

- Advantages:
  - Easy to use (graphical languages)
  - Powerful algorithms for
    - synthesis (SW and HW)
    - verification

- Disadvantages:
  - Sometimes overspecify implementation
    (sequencing is fully specified)
  - Numerical computations cannot be specified compactly
    (need extended FSMs)
Extended FSM Example

Process 1:
forever {
    x = 0; y = 0;
    while (x < 10) {
        y = y + get_data(B);
        x = x + 1;
    }
    put_data(A, y / 10);
}

Process 2:
forever {
    a = get_data(A);
    if (a < 0)
        put_data(C, a);
    else
        put_data(D, -a);
}
Extended FSM Example

s1
x = 0
y = 0

proc. 1

s2
y = y + get_data (B)
x = x + 1

s3
put_data (A, y / 10)

proc. 2

t1
a = get_data (A)

s
a < 0

N
S

N
S

N
S

put_data (D, -a)

put_data (C, a)
Formal Extended FSM Example

proc. 1

s1

< 10

s2

x

+1

/s10

B

y

+1

A

N

S

ds3

proc. 2

(t1

< 0

S

N

(t3

(t2

C

D

a

-
Extended FSM primitives

- initial state
- state
- decision
- transition
- FSM/data connection
- constant
- variable
- communication channel
- data dependency
- operator
- loading

© ASV et al. 1996
Communication models

- Synchronous:
  all FSMs make a transition simultaneously

- Asynchronous:
  communication is mediated by “channels”:
  - blocking write/blocking read
    (rendez-vous: both partners must be ready)
  - non-blocking write/blocking read
    (FIFOs)
  - non-blocking write/non-blocking read
    (shared variables)
Communication models

● Synchronous:
  ◆ predictable (determinacy)
  ◆ highly constraining

● Asynchronous:
  ◆ unpredictable (result depends on scheduling/timing)
  ◆ does not constrain the implementation
    (good for heterogeneous embedded systems)
  ◆ blocking write: difficult to implement correctly
  ◆ non-blocking write: needs unbounded buffers
    (or may lose events)
  ◆ non-blocking read: consistency problems

© ASV et al. 1996
Outline

- Reactive Real-Time Embedded Systems
- Specification Models and Languages
  - Data Flow
  - Extended Finite State Machines
- Proposed Design Methodology
  - System Specification
  - Validation
  - System Partitioning
  - Software, Hardware and Interface Synthesis
  - Real-time Operating System and Scheduling
- Summary
Software Models

- Advantages:
  - Executable model
  - Object-oriented analysis:
    - Natural hierarchical decomposition
    - Inheritance
      (simplifies extension and re-use)
    - Method invocation as communication primitive
- Disadvantages:
  - Strongly biased towards SW
  - (Almost) impossible to verify formally
Object-Oriented Methodologies

- Various CASE methodologies use an Object-Oriented paradigm
  (Shlaer-Mellor ‘88)
- Languages are widely known (C++, Java, ...)
- Typical domains of application:
  - Rapid prototyping
  - Complex (mainly SW) systems
  - Network-wide programming
- The Object-oriented ideas are not limited to software models !!!
• Advantages:
  ◆ Powerful algorithms for
    ◆ synthesis
    ◆ verification
  ◆ (Almost) standard languages
    (VHDL, Verilog, UDL/I)
  ◆ Timing is handled explicitly
  ◆ Software-like extensions (e.g. VHDL, Verilog)

• Disadvantages:
  ◆ Strongly biased towards HW
  ◆ Not really formal...
Hardware Simulation Models

- Typical domain of application:
  - Hardware design
- Can be considered a least common denominator among SW and HW
- With some constraints, can have an EFSM-based semantics
  - “synthesizable subsets”
  - cycle-based simulation
Outline

- Reactive Real-Time Embedded Systems
- Specification Models and Languages
  - Data Flow
  - Extended Finite State Machines
- Proposed Design Methodology
  - System Specification
  - Validation
  - System Partitioning
  - Software, Hardware and Interface Synthesis
  - Real-time Operating System and Scheduling
- Summary
Reactive Specification Languages

- Main aspects of language choice:
  - Ease of use (domain-specific)
  - Availability of tools/methodologies:
    - a tool/methodology implies one or more languages (and models)
    - graphical capabilities
  - (e.g. structured analysis uses DF and FSMs)
  - Standards/regulations
  - Tradition ...
Graphical FSM Languages

- StateCharts, BetterCharts, SpeedCharts, ...
  (Har’el ‘90)
- Easy to use for control-dominated systems
- Simulation (animated), SW and HW synthesis
- Extended with arithmetics
- Hierarchical states necessary for complex reactive system specification
Synchronous Languages

- Assumptions:
  - the system continuously reacts to internal and external events by emitting other events
  - events can occur only at discrete instants
  - zero (negligible) reaction time
- Both control (Esterel) and data flow (Lustre, Signal)
- Very simple syntax and clean semantics (based on FSMs)
- Deterministic behavior
- Simulation, software and hardware synthesis, verification

© ASV et al. 1996
ESTEREL

- Designed at INRIA by Berry et al.
- Concurrent modules:
  - interface signals, possibly with values
  - local signals and variables
  - statements, e.g.:
    - await (single or multiple signals)
    - do stmt1 watching signal [timeout stmt2]
      (instantaneous killing of stmt1)
    - trap exception in stmt1 [handle do stmt2]
      (allow stmt1 to terminate)
  - allows “external” procedures and functions
module counter:

input go, reset, req; output ack(integer);

var t:integer in

loop do
  t:=0;
  every go do
    t:=t+1;
    await req; emit ack(t)
  end

watching reset
end end.
Summary of Models/Languages

- Models/languages for control and data:
  - same object (embedded computation), yet...
  - different specification, different optimization, different validation
- Currently: need to pick style at the beginning, and hope for the best
- Future:
  - at least, mix styles freely
  - at best, decouple specification and optimization styles
    (unified underlying model)
Outline

● Reactive Real-Time Embedded Systems
● Specification Models and Languages
  ◆ Data Flow
  ◆ Extended Finite State Machines
● Proposed Design Methodology
  ◆ System Specification
  ◆ Validation
  ◆ System Partitioning
  ◆ Software, Hardware and Interface Synthesis
  ◆ Real-time Operating System and Scheduling
● Summary

© ASV et al. 1996
Main Codesign Methods and Tools

- CHINOOK (Chou, Ortega, Borriello et al. ‘92-...)
- COSYMA (Ernst, Henkel et al. ‘92-...)
- MEIJE (Berry, Gonthier, Halbwachs, Caspi, Benveniste, Le Guernic et al. ‘91-...)
- POLIS (Chiodo, Lavagno, Sangiovanni et al. ‘92-...)
- PTOLEMY (Kalavade, Buck, Lee et al. ‘92-...)
- VULCAN (Gupta, Coelho, De Micheli et al. ‘92-...)
- ...

© ASV et al. 1996
CHINOOK (U. of Washington)

- Specification: Verilog HDL
- Internal representation: Event Graph (CDFG)
- Validation: none specific (Verilog simulation)
- Partitioning: directed by scheduling constraints
- Scheduling: aimed at satisfying timing constraints
  ("modes" allow complex constraints)
- Synthesis: Verilog to C translator
- Main emphasis on interface synthesis
  (port assignment and driver synthesis)
COSYMA (U. of Braunschweig)

- Specification: C* (C++ extended with concurrency)
- Internal representation: ES graph (CDFG)
- Validation: none specific (C++ execution)
- Partitioning: two nested loops
  - outer: hand-driven, uses synthesis and profiling for cost estimation
  - inner: simulated annealing, uses quick estimator
- Scheduling: none specific
- Synthesis: hardware extraction from (subset of) ES graph
- Main emphasis on partitioning for hardware accelerators

© ASV et al. 1996
MEIJE (INRIA and others)

- Specification: synchronous languages for control and data flow (Esterel, Lustre, Signal)
- Internal representation: OC (EFSM)
- Validation:
  - synchronous simulation
  - formal verification
- Partitioning: none
- Scheduling: not needed (synchronous hypothesis)
- Synthesis: hardware from EFSM, software from hardware
- Main emphasis on determinate reaction to events
- Specification: FSM-based languages (Esterel, ...)
- Internal representation: CFSM network
- Validation:
  - high-level co-simulation
  - FSM-based formal verification
- Partitioning: by hand, based on co-simulation estimates
- Scheduling: classical RT algorithms
- Synthesis:
  - S-graph-based code synthesis for software
  - logic synthesis for hardware
- Main emphasis on unbiased verifiable specification
PTOLEMY (U. C. Berkeley)

- Specification: Data Flow graph
- Internal representation: DFG
- Validation: multi-paradigm co-simulation
  (DF, discrete events, ...)
- Partitioning: greedy, based on scheduling
- Scheduling: linear, sorting blocks by “criticality”
  (bit-level in HW, memory-intensive in SW)
- Synthesis:
  ◆ DSP code stitching for software
  ◆ custom DSP synthesis (LAGER) for hardware
- Main emphasis on heterogeneous computation models
VULCAN (Stanford U.)

- Specification: Hardware C
- Internal representation: CDFG
- Validation: custom co-simulator
- Partitioning: greedy, based on scheduling
- Scheduling: timing-driven
  - each I/O operation or unbounded loop initiates a thread
- Synthesis: high-level synthesis (OLYMPUS) for hardware
- Main emphasis on timing-driven scheduling of threads

© ASV et al. 1996
Outline

- Reactive Real-Time Embedded Systems
- Specification Models and Languages
  - Data Flow
  - Extended Finite State Machines
- Proposed Design Methodology
  - System Specification
  - Validation
  - System Partitioning
  - Software, Hardware and Interface Synthesis
  - Real-time Operating System and Scheduling
- Summary
Codesign Finite State Machines

- We have chosen an FSM model for
  - uncommitted
  - synthesizable
  - verifiable
  
  HW/SW specification

- Translators from state diagrams, Esterel, HDLs into a single FSM-based language

- Need efficient hw/sw communication primitive:
  - Event broadcasting

- Software response could take a long time:
  - Unbounded delay assumption
Communication primitive: event

- One-way data communication
- Need efficient implementation
  (interrupts, buffers...)
- No mutual synchronization requirement, but...
  ➤ Building block for higher-level synchronization primitives
- Examples:
  - *valued event*: temperature sample
  - *pure event*: excessive temperature alarm
Introducing a CFSM...

- A Finite State Machine
- Input events, output events and state events
- Initial values (for state events)
- A transition function
  - Transitions may involve complex, memory-less, instantaneous arithmetic and/or Boolean functions
  - All the state of the system is under form of events
- Need rules that define the CFSM behavior
CFSM Rules: phases

- Four-phase cycle:
  1. Idle
  2. Detect input events
  3. Execute one transition
  4. Emit output events

- Discrete time
  - Sufficiently accurate for synchronous systems
  - Feasible formal verification

- Model semantics: *Timed Traces* i.e. sequences of events labeled by time of occurrence
CFSM Trace Semantics

(i1==v1) and i3 and i4 => \{o2,o3\}

(i1==v2) and i2 => \{o2,o3\}

i4 and i5 => \{o4\}

© ASV et al. 1996
CFSM Rules: phases

- Implicit *unbounded delay* between phases
- *Non-zero* reaction time (avoid *inconsistencies* when interconnected): minimum delay is 1 time unit
- *Causal* model based on *partial order* (potential verification speed-up)
- Phases may overlap
CFSM Rules: events

- Hw is always “ready”
- Software may not be ready
  - Events may be “lost”
  - Implicit depth-1 buffer associated with every event
- Event: basic tool to implement synchronization
  - Trigger event can cause at most 1 transition
  - All output events of a transition must be emitted
CFSM Rules: additional constraints

- What if some event may not be lost?
- Tag some event as “critical”
- The problem is deferred to the partitioning and scheduling phases:
  - use Formal Verification to identify critical events
  - partition or schedule the system so that the resulting constraint is satisfied
- The same technique can be used to assign priorities to events
Network of CFSMs: Depth-1 Buffers

© ASV et al. 1996
CFSMs and FSMs

- The best known automated formal verification methods are based on synchronous FSMs
  - define the behavior of a CFSM network as an equivalent “standard” FSM network
- Event-driven model: self-loop until events are detected
- Depth-1 buffers on input and output events implemented as non-deterministic FSMs
- Additional “verifiability” (atomicity) constraints:
  - events are detected only if a transition occurs
  - all inputs are “cleared” if a transition occurs

© ASV et al. 1996
Buffer FSMs

111 => 11
00-  => 0-
011 => 0-

012 => 0-
112 => 11

011 => 0-
111 => 12

10-  => 11
112 => 12
- 0- => 0-

112 => 12
00-  => 0-
012 => 0-

012 => 0-
011 => 0-

111 => 11

Main FSM

© ASV et al. 1996
CSFM network
   - Specific representation format: SHIFT
   - Unbounded-delay interpretation
SHIFT description
   - List of input variables
   - List of output variables
   - Tabular transition relation
   - Arithmetic expressions represented as (library) function netlists
Hardware - Software Architecture

- Hardware:
  - One or more microcontrollers
  - ASICs, DSPs....
- Software:
  - Set of concurrent *tasks*
  - Scheduler
  - Customized operating system
- Interfaces:
  - Hardware modules
  - Software procedures (polling, interrupt handlers, ...)

© ASV et al. 1996
System Partitioning

© ASV et al. 1996
Interfaces Among Partitions

- Automatically generated
- Standardized strobe/data protocol (corresponding to the event/value primitive)
- Allow to use hand-designed modules (following the interfacing convention)

© ASV et al. 1996
An example of interface: hw to sw

© ASV et al. 1996
System Validation

- Safety-critical real-time systems *must* be validated
- Explicit exhaustive simulation is infeasible
- Formal verification can achieve the same level of safeness
- How to use verification and simulation together?
- Simulation can be used initially for
  - Quick functional debugging
  - Ruling out obvious cases (can be expensive to verify)
- Then formal verification takes over for exhaustive checking, but...
- Simulation is used again as *user interface* to provide the designer with *error traces*
Example of Formal Verification

(*KEY == ON) => *START

(*KEY == ON) and (*BELT == ON) =>

(*KEY == OFF) or (*BELT == ON) =>

(*END == 10) or (*BELT == ON) or (*KEY == OFF) => *ALARM = OFF

(*END == 5) => *ALARM = ON

WAIT

OFF

ALARM

© ASV et al. 1996
Example of Formal Verification

- Untimed property, e.g. using Temporal Logic (CTL, Pnueli ‘77)
  - $\text{AG}(\text{ALARM\_ON} \rightarrow \text{AF}(\text{ALARM\_OFF}))$

- Assumption: non-zero unbounded delays
  - Property doesn’t hold
  - Deduce reason for failure from error trace
    - Need tighter delay range
    - Specification refinement

© ASV et al. 1996
Example of Formal Verification

- Pick any delay $K$
  - Property holds

- Conclusion:
  - Any implementation with *bounded non-zero* delays satisfies the property
Example of Formal Verification

- Timed property, e.g. Timed Temporal Logic (TCTL, Koymans ‘85)
  - $\text{AG(\text{ALARM\_ON} \rightarrow \text{AF}_{\leq 6s}(\text{ALARM\_OFF}))}$

- Property doesn’t hold for all $K$, it only holds for:
  - 0 input delay, and
  - output delay ranging from 0 to 0.5 s
Example of Formal Verification

- A weaker timed property
  - $AG(ALARM\_ON \rightarrow AF_{<11s}(ALARM\_OFF))$
- There are some combinations of input and output delays that satisfy the property

- This delay information can be used to "refine" the specification and restrict "legal" implementations to be consistent with the specification
Example of Formal Verification

- Purely hardware implementation
  - Both “6s” and “11s” properties hold if propagation delay < 0.5s
- Hw-Sw implementation
  - Software implementation has non-zero input delays
    - No Hw-Sw can satisfy the “6s” property: zero input delay is not feasible
  - Some “fast” Hw-Sw implementation can satisfy the “11s” property
Problems of Formal Verification

- Is the error trace “real?”
  - Maybe not, because of our simple model of environment...
  - No driver can turn off and on the key in one tenth of a second!
  - Some behaviors may not be possible
- The cause of failure may be hard to decipher from “automatic” simulation
- Formal verification is hard (state explosion)
  - Longest run takes 6.5 hrs
    - 60 million states (time unit 0.1s)
Our Co-design Environment

Formal Verification

Graphical EFSM

ESTEREL

Compilers

Partitioning

CFSMs

Sw Synthesis

Sw Code + RTOS

Intfc Synthesis

Hw Synthesis

Logic Netlist

Prototype
High-level Co-simulation

- Functional (untimed) simulation allows:
  - functional (partial) correctness, by generating inputs and observing outputs
  - debugging, by easy access to internal states
- High-level (timed) co-simulation allows:
  - feasibility analysis for specification
  - hardware/software partitioning
  - architecture selection (CPU, scheduler, ...)
- Cannot be used to validate the final implementation
  - need a much more detailed model of HW and SW architecture

© ASV et al. 1996
Co-simulation Requirements

- Fast, for rapid testing of
  - different input stimuli
  - different architectures
- Interactive
  - quickly change architectural parameters
  - easily analyze results and debug
    (graphical interface)
- Accurate
  - hardly compatible with speed and interactivity

© ASV et al. 1996
Existing tools and methods

- Hardware-oriented simulation
  - Processor modeled at instruction or Register Transfer level (Verilog, VHDL, ...)
  - Fairly accurate, but fairly slow
- Functional simulation (mostly for DSP)
  - “Block” programming environments
    - data flow: MATLAB, SPW, COSSAP, ...
    - control flow: SDL, StateCharts, ...
  - System modeled as discrete or continuous data flow
  - Computation time is usually ignored
- Prototyping (breadboards...)

© ASV et al. 1996
Our co-simulation approach

- Based on synthesized software timing estimates
- Synthesized C code annotated with clock cycles required on several processors
- Clock cycle accumulation during simulation to synchronize the software
  - with the hardware
  - with the environment
- Uses Ptolemy (Lee et al. 92) as:
  - graphical interface
  - simulation engine
  (heterogeneous models can coexist)
Our Co-simulation Approach

- Resource scheduling problem:
  - hardware CFSMs are concurrent
    (simulated in a cycle-based fashion)
  - only one software CFSM can be active at a time
  - use the same (selectable) scheduling policy as will be used in the real system
Trade-off Evaluation

- Parameters associated with each hierarchy level:
  - can be changed on the fly (no recompilation)
  - define different architectural aspects:
    - implementation of each CFSM
    - CPU type, clock speed, ...
    - constant inputs (scaling factors, priorities, ...)
- Hierarchical inheritance eases structured partitioning
- Automatically transmitted to following synthesis steps
Types of analysis

- Powerful graphical environment to generate inputs and analyze outputs (Ptolemy)
- Functional simulation:
  - no clock cycle accumulation
  - useful for debugging and demonstration to customer
    (“virtual prototype”)
- Timed co-simulation:
  - “Lost” input events (missed deadlines) can be selectively reported
  - CPU utilization graphs (for schedulability analysis)
Future Work

- Interrupt handling
  - nested interrupts
  - maskable interrupts
- Multi-processor systems
  - static allocation
  - dynamic allocation
- Clock accumulation also within the scheduler
- Co-simulation in other environments
  - VHDL, Verilog output for HW and SW
Our Co-design Environment

Formal Verification

Partitioning

Simulation

Graphical EFSM

ESTEREL

Compilers

CFSMs

Sw Synthesis

Intfc Synthesis

Hw Synthesis

Sw Code + RTOS

Logic Netlist

Prototype

© ASV et al. 1996
Software Implementation Problem

- **Input:**
  - set of tasks (specified by CFSMs)
  - set of timing constraints (e.g., input event rates and response constraints)

- **Output:**
  - set of procedures that implement the tasks
  - scheduler that satisfies the timing constraints

- **Minimizing:**
  - CPU cost
  - memory size
  - power, etc.
Software Implementation

● How to do it?
● Traditional approach:
  ◆ hand-coding of procedures
  ◆ hand-estimation of timing input to scheduling algorithms
● Long and error-prone
● Our approach: three-step *automated* procedure:
  ◆ synthesize each task separately
  ◆ extract (estimated) timing
  ◆ schedule the tasks
● Customized RT-OS (scheduler + drivers)
Software Implementation

- Current strategy:
  - Iterate between synthesis, estimation and scheduling
  - Designer chooses the scheduling algorithm
- Future work:
  - Top-down propagation of timing constraints
  - Software synthesis under constraints
  - Automated scheduling selection
    (based on CPU utilization estimates)
Software Implementation

- Sub-problems:
  - Find appropriate representations for
    - code optimization
    - scheduling
  - Find appropriate code optimization algorithms
    (timing and memory occupation)
  - Find appropriate scheduling algorithm
    (guaranteed performance with acceptable overhead)
Software synthesis procedure

- Specification, partitioning
- S-graph synthesis
- Timing estimation
- Scheduling, validation
- Code generation
- Compilation
- Testing, validation
- Production

© ASV et al. 1996
Task implementation

● Goal: quick response time, within timing and size constraints

● Problem statement:
  ◆ Given a CFSM transition function and constraints
  ◆ Find a procedure implementing the transition function while meeting the constraints

● The procedure code is acyclic:
  ◆ powerful optimization and analysis techniques
  ◆ looping, state storage etc. are implemented outside (in the OS)
Representation Issues

- The software representation should be:
  - Low-level enough to allow detailed optimization and estimation
  - High-level enough to avoid excessive details e.g. register allocation, instruction selection
- Main types of “user-mode” instructions:
  - data movement
  - ALU
  - conditional/unconditional branches
  - subroutine calls
- RTOS handles I/O, interrupts and so on
Multi-valued Decision Diagrams

- Extension of Binary-valued Decision Diagram (Akers ‘69, Bryant ‘86, Kam’92)
  - Appropriate for control-dominated tasks
  - Single-path, single-test evaluation
  - Size strongly depends on variable ordering
  - Well-developed set of optimization techniques
- Must be augmented with arithmetic and Boolean operators, to perform data computations

\[ f = a + b \]
Our Representation: S-graphs

- Acyclic extended decision diagram computing a transition function
- S-graph structure:
  - directed acyclic graph
  - set of finite-valued variables
  - TEST nodes evaluate an expression and branch accordingly
  - ASSIGN nodes evaluate an expression and assign its result to a variable
An example of S-graph

- input event \(*c*
- output event \(*y*
- state int \(a\)
- input int \(b\)
- forever
  - if (detect(*c))
  - if (\(a < b\))
    - \(a := a + 1\)
    - emit(*y*)
  - else
    - \(a := 0\)
    - emit(*y*)

\(\text{BEGIN}\)

\(*c\)
\(\text{T}\)
\(\text{a<b}\)
\(\text{F}\)
\(\text{F}\)
\(\text{F}\)
\(\text{T}\)
\(\text{T}\)
\(\text{END}\)
S-graphs and functions

- Execution of an s-graph computes a function from a set of input and state variables to a set of output and state variables:
  - Output variables are initially undefined
  - Traverse the s-graph from BEGIN to END
- Well-formed s-graph:
  - every time a function depending on a variable is evaluated, that variable has a defined value
- How do we derive an s-graph implementing a given function?
S-graphs and functions

- Problem statement:
  - Given: a finite-valued multi-output function over a set of finite-valued variables
  - Find: an s-graph implementing it
- Procedure based on Shannon expansion
  \[ f = x f_x + x' f_{x'} \]
- Result heavily depends on ordering of variables in expansion
  - inputs before outputs: TESTs dominate over ASSIGNs
  - outputs before inputs: ASSIGNs dominate over TESTs
Example of S-graph construction

$x = a \ b + c$
$y = a \ b + d$

Order: $a, b, c, d, x, y$
(inputs before outputs)

© ASV et al. 1996
Example of S-graph construction

\[ x = a \ b + c \]
\[ y = a \ b + d \]

Order: \(a, b, x, y, c, d\) (interleaving inputs and outputs)
S-graph optimization

- General trade-off:
  - TEST-based is faster than ASSIGN-based (each variable is visited at most once)
  - ASSIGN-based is smaller than TEST-based (there is more potential for sharing)
- The procedure can be iterated over s-graph fragments:
  - local optimization, depending on fragment criticality (speed versus size)
  - constraint-driven optimization (still to be explored)
From S-graphs to instructions

- TEST nodes ➔ conditional branches
- ASSIGN nodes ➔ ALU ops and data moves
- No loops in a *single* CFSM transition
  (user loops handled at the RTOS level)
- Data flow handling:
  - “don’t touch” them (except common subexpression extraction)
  - map expression DAGs to C expressions
  - C compiler allocates registers and select opcodes
- Need source-level debugging environment (with any of the chosen entry languages)
Software synthesis procedure

- Specification, partitioning
  - S-graph synthesis
  - Timing estimation
- Scheduling, validation

- Code generation
  - Compilation
  - Testing, validation
- Production

© ASV et al. 1996
Performance and cost estimation

- S-graph: low-level enough to allow accurate performance estimation
- Cost parameters assigned to each node, depending on:
  - system type (CPU, memory, bus, ...)
  - node and expression type
- Cost parameters evaluated via simple benchmarks
  - need timing and size measurements for each target system
  - currently implemented for MIPS, 68332 and 68HC11 processors
Example: 68HC11 timing estimation
- Cost assigned to s-graph edges
  - (different costs for taken/not taken branches)
  - Estimated time:
    - min: 26 cycles
    - max: 126 cycles
- Accuracy: within 20% of profiling
Experimental results (68HC11)
Experimental results (68HC11)

- Support: inputs before dependent outputs
- Two-level: switch on state switch on inputs
Future Work

- Better synthesis techniques
  - add state variables to simplify s-graph
  - performance-driven synthesis of critical paths
  - exact memory/speed trade-off
- Estimation of caching and pipelining effects
  - may have little impact on control-dominated systems
    (frequent branches and context switches)
  - relatively easy during co-simulation
Software synthesis procedure

- Specification, partitioning
  - S-graph synthesis
  - Timing estimation
  - Scheduling, validation
    - not feasible
    - feasible
  - fail

- Code generation
  - Compilation
    - Testing, validation
      - fail
      - pass

- Production
The scheduling problem

• Given:
  ◆ estimates on the minimum and maximum execution times for each CFSM transition (from the S-graph)
  ◆ a set of timing constraints
    e.g., input event rates and input-to-output deadlines, “critical” events

• Find an execution ordering for CFSM transitions that satisfies the constraints:
  ◆ either static, pre-computed (off-line)
  ◆ or dynamic, decided at run time (on-line)
Scheduling algorithms

- Off-line scheduling: determine a cyclic execution order that satisfies the constraints
  - weak constraints: round-robin cyclic executive
    (like the synchronous hypothesis in Esterel)
  - tight constraints: call each CFSM only when it is expected to receive an event
    (based on expected I/O rates)
- Advantages: simple, fast, highly predictable
  (essential for mission-critical systems)
- Disadvantage: low utilization of CPU to guarantee constraint satisfaction
Scheduling algorithms

- On-line scheduling: determine a set of priority values that determine the next runnable CFSM
- Priorities can be statically or dynamically determined
- A running CFSM may or may not be interrupted in the middle of a transition (preemptive/non-preemptive algorithms)
- Advantage: higher CPU utilization
- Disadvantage: more complex, higher overhead (dynamic and preemptive most complex)
Scheduling algorithms

- Currently implemented algorithms:
  - round-robin cyclic executive
  - off-line I/O rate-based cyclic executive
  - static pre-emptive: Rate Monotonic Scheduling (Liu ‘73):
    - highest I/O rate has highest priority
  - dynamic pre-emptive: Earliest Deadline First (Liu ‘73):
    - CFSM with nearest deadline has highest priority
Problems with Current Approach

- Current scheduling algorithms:
  - Lots of manual analysis required
  - Either guaranteed performance with high overhead
  - Or no guarantee but highly efficient
  - Schedulability analysis is usually very pessimistic
    - waste of CPU power at run time
- Scheduling algorithm choice is left to the user
Future Work

- Propagation of constraints from external I/O behavior to each CFSM
  - probabilistic: Markov chains
  - exact: FSM state traversal
- Satisfaction of constraints within a single transition
  (e.g., software-driven bus interface protocol)
- Automatic choice of scheduling algorithm, based on performance estimation and constraints
- Scheduling for verifiability
Other scheduling models

● Problem: computation result may depend on dynamic schedule

● Synchronous systems (Esterel, Signal, Lustre): no scheduler needed
  (as long as the software is fast enough)

● Data-flow systems: result does not depend on scheduling if event detection is blocking (Kahn ’74)

● Can we obtain determinism without losing efficiency?
Our Co-design Environment

Formal Verification

Partitioning

Simulation

Graphical EFSM

ESTEREL

Compilers

CFSMs

Sw Synthesis

Intfc Synthesis

Hw Synthesis

Sw Code + RTOS

Logic Netlist

Prototype

© ASV et al. 1996
Hardware Synthesis

- CFSMs interpreted as synchronous register-transfer specification
- Direct implementation as combinational logic + registers
- Non-zero delay implemented by latching all the outputs
  - Ensures correct composition (Moore-type synchronous FSMs)
  - Improves testability
- Logic synthesis for various target implementations
  - FPGAs and FPICs for rapid prototyping

© ASV et al. 1996
Hardware Synthesis

CFSM 1

comb. logic

L

CFSM 2

comb. logic

L
Micro-controller peripherals

- Custom HW (fully programmable, expensive)
- On-chip or off-chip peripheral (partially programmable, inexpensive)
Previous work

- Chou et al. (DAC 94): synthesis of device drivers (given choice and protocol)
- Mitra et al. (TVLSI 96): mapping of function to complex peripheral devices (syntactic matching only)
Peripheral modeling approach

- Ideally: implement specified function using peripherals (if possible)
- Currently: use three models
  - Behavioral (Ptolemy) model for co-simulation
  - CFSM model for RTL co-simulation and rapid prototyping
  - C model for implementation (programming and interfacing with the peripheral)
- Parameters customize all models simultaneously (plug-in replacement of abstraction levels)
- Synthesizable CFSM model key to limited re-targetability

© ASV et al. 1996
Peripheral modeling approach

- The user must
  - decide in advance which functions *may need to be* implemented on a library peripheral
  - choose the best fitting model from a library
  - co-simulate to decide implementation (SW, custom HW, peripheral, ...)
- The co-design environment takes care of:
  - synthesizing in SW or HW
  - extracting peripheral programming SW from library (may be partially micro-controller independent)
  - interfacing transparently
Current Status

● Modeled various peripherals of the 68hc11 family
  ◆ Timer unit
    ◆ input capture (measuring time of events)
    ◆ output compare (generating delays)
  ◆ A/D converter
  ◆ PWM generators
● Writing and debugging a new model requires 1-2 days (for simple peripherals)
Our Co-design Environment

- Formal Verification
- Partitioning
- Simulation
- Graphical EFSM
- ESTEREL
- Compilers
- CFSMs
- Sw Synthesis
- Intfc Synthesis
- Hw Synthesis
- Sw Code + RTOS
- Logic Netlist
- Prototype

© ASV et al. 1996
Why hardware prototypes?

- High-level co-simulation cannot be used to validate the final implementation
  - need a much more detailed model of HW and SW architecture
- Low-level co-simulation (using HW simulator) is too slow
- Need to validate the design in the real environment
- Example: engine control
  - specification cannot be formalized ("must run well")
  - must be loaded on a vehicle for test drives
Rapid prototyping methodology

- Synthesis from a high-level uncommitted specification
- Hardware, software, RT-OS and interfaces are automatically generated
- Use the APTIX Field-Programmable Interconnect Circuits to quickly modify the board configuration
- System netlist changes reflected on the board in a matter of minutes
- Can be programmed via EPROM for field deployment
The APTIX Board

Analog components

RAM
FPIC
EPROM
FPIC
FPIC
CPU
FPGA
FPIC
Connectors
The Development Environment

- Microcontroller emulator for software debugging
- XCHECKER to program and observe XILINX FPGAs
- APTIX FPIC-D can bring any interconnection to a logic analyzer pod
- APTIX software can program the HP logic analyzer
- Currently the bottleneck is the FPGA programming environment (slow CAD, slow programming, slow debugging)
Current status and future work

- Virtual prototyping environment is complete
- Physical prototyping environment:
  - debugged “toy” system
  - currently developing a dashboard controller
- Future work:
  - enhance software debugging capabilities
    (ESTEREL source debugging via emulator)
  - automate system netlist generation
  - explore new types of FPGAs
Current Design Flow

- System specification:
  - ESTEREL
  - FSM editor
  - graphical CFSM net editor
- SW synthesis and estimation
- High-level co-simulation
  - functional debugging
  - trade-off evaluation
- Formal verification
- SW, HW, RTOS synthesis
- Low-level co-simulation and prototyping
Car dashboard example

- Implemented a dashboard controller:
  - Speedometer and odometer
  - Safety functions (seat-belt alarm)
  - Fuel gauge control
- Timing functions implemented by the user (custom scheduler)
- 23 interacting CFSMs, of 13 different sorts:
  - Speed, Odo, RPM (speedometer, odometer)
  - Belt (safety)
  - Crossdisp (fuel)
  - FRC, Timer (scheduler)
Car dashboard example

- Control functions specified with Esterel
  - automated translation from Esterel to CFSMs
- CFSM interconnection specified graphically

fuel
key, belt

clock

FRC
Timer

wheel
Odo

engine

Belt
Fuel

Crossdisp

speed_disp

RPM

odo_disp

RPM_crossdisp

FRC

Fuel_crossdisp

odo_crossdisp
Car Dashboard Implementation (I)

- Defined by hand, based on fast co-simulation output
- Custom HW blocks:
  - Timer counter (originally part of 68HC11 timer unit)
  - PWM generators (jitter is critical)
- SW blocks: everything else
- Estimated code size: 7Kbytes (excluding OS)
- Estimated CLBs on XILINX: >170
- HW/SW communication: 31 bits (events) + 82 bytes (values) of memory-mapped I/O
Car Dashboard Implementation (I)

- Actual code size:
  - user code: 9130 bytes ROM + 370 bytes RAM
  - custom RTOS: 1543 bytes ROM + 1475 bytes RAM
  - size estimation accuracy: within 20%
    (error due to ignoring different cost of SW MULT)
- Actual HW size: 2 XILINX 3000
  - 130 CLB + 50 I/O
  - 144 CLB + 48 I/O
  (automated partitioning)
Car dashboard Implementation (II)

- Re-defined using 68HC11 peripherals:
  - Timer unit for input event processing and scheduling
  - PWM generators for outputs (on 68HC11GAUSS)
- SW blocks: everything else
- Estimated code size: 10Kbytes (including RTOS)
- PWM generators can also be implemented on FPGA (if not available on micro-controller)
Comparison with Manual Design

● Exact comparison is difficult
● Similar code size
  (8K bytes vs. 10Kbytes)
● Synthesized code is much faster
  (3 times for the speedometer computation chain)
● RTOS overhead is being reduced
  ◆ careful usage of RAM
  ◆ support for micro-controller resource usage
    (timers, PWM generators, serial I/O)
Conclusions

- Introduced a new model for hardware and software
  - Simple: finite state, no complex synchronization,...
  - Uncommitted to a specific implementation
  - Event-driven causal model with implicit unbounded delays
Conclusions

- The proposed methodology is based on a implementation-independent specification format
- Partitioning and synthesis are made easier by the common model used
- Partitioned hardware and software implementations are automatically generated
- The FSM model derived from a CFSM is compatible with many formal verification algorithms
Conclusions

- Rapid prototyping is required for embedded system design

- Three levels of prototypes:
  - functional prototype, with animation
  - approximate timing prototype, for performance evaluation and partitioning
  - physical prototype for testing on the field

- All three depend on
  - uncommitted formal specification
  - automated synthesis