HW/SW Co-Design for Embedded Systems

University of California at Berkeley
Magneti Marelli
Politecnico di Torino
Cadence Design Systems

March 1995

Outline

- Reactive Real-Time Embedded Systems
- Hardware-Software Co-Design
- System Specification Models
  - Co-Design Finite State Machine
- Formal Design Methodology
  - Software Synthesis
  - Hardware Synthesis
  - Validation
- Summary
- Case Studies

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:
  - performs a single function
  - 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, ...)

Embedded System Design

- Main design criteria:
  - cost
  - performance
  - reliability
  - features
  - modifiability
- Software used for cost, features and modifiability

  (cheaper; easier debugging and upgrading)
- Hardware used for performance

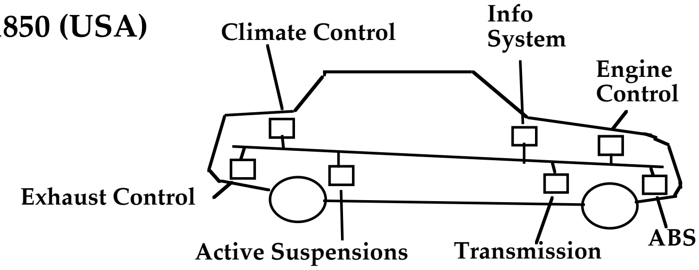
Reactive Real-time Systems

- Reactive Real-Time Systems
  - "React" to external environment
  - Maintain permanent interaction
  - Ideally never terminate
  - Subject to external timing constraints (real-time)

Embedded Controller Example: In-Vehicle Network (IVN)

Several protocols have been proposed for the implementation of in-vehicle networks. Among them:

- CAN, VAN (Europe)
- J1850 (USA)



IVN - Implementation

Automotive networks come in three classes:

class	application	max.latency	bit rate
A	Body electronics	20-50 ms	< 10 kbit/s
B	Information sharing	1-10 ms	10-20 kbit/s
C	Real-time controls	< 1 ms	0.125-1 Mbit/s

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

- class A - CPU can handle comm. + appl.
- class C - communication IC required

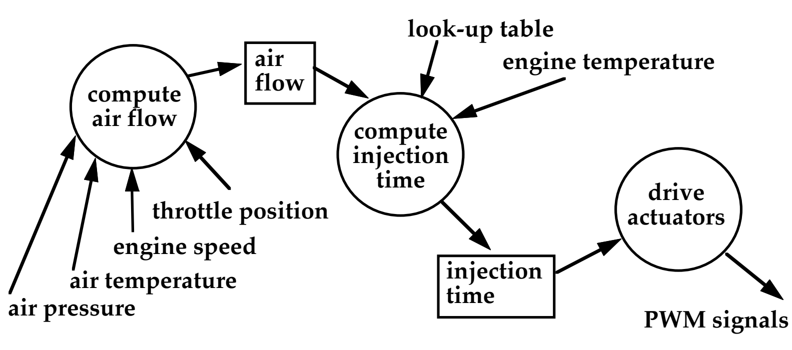
Embedded Controller Example: Engine Control Unit (ECU)

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

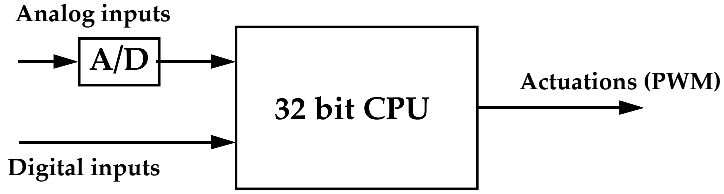
- **Major constraints:**
  Low fuel consumption
  Low exhaust emission

Engine Control Unit (ECU) - 2

Task: control injection time (3 sub-tasks)



Engine Control Unit (ECU) - Option 1

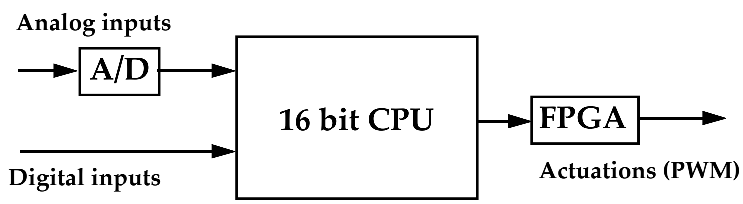


CPU has to:

- process input data
- compute outputs
- control actuators

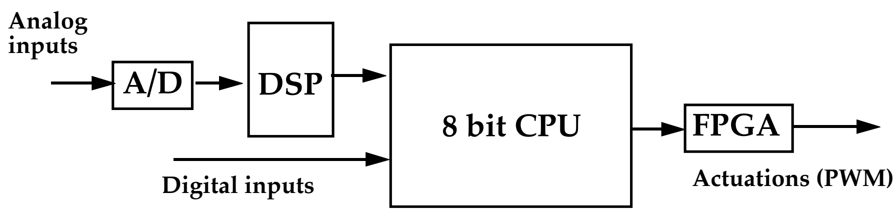
- Relatively easy to design
- May not meet timing requirements

Engine Control Unit (ECU) - Option 2



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

Engine Control Unit (ECU) - Option 3



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

RRTES Implementations

- Possibly contain both hardware and software (ASIC plus embedded software)
- Past Design Methodology
  - Software implementation:
    timing $\longrightarrow$ 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 incompatibility across HW-SW boundary
- Lack of well-defined design flow
  - Time to market
  - Specification revision difficult

➠Need Hardware-Software Co-Design

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

System Specification Models

- Main purpose: provide clear and unambiguous description of system *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 *implementation*

System Specification Models

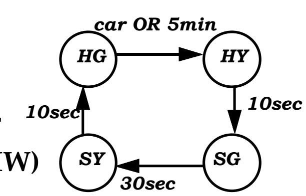
- Distinguish between *models* and *languages*
  (a language implies one or more models)
- Main models for embedded systems:
  - Finite State Machines
  - Data Flow Diagrams
    (Petri Nets)
  - Object-oriented models
  - Hardware simulation models

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, ...
  - Available tools
  - Personal taste and/or company policy

Finite State Machines



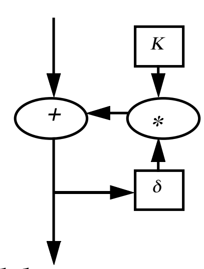
- Advantages:
  - Easy to use (graphical)
  - Powerful algorithms for
    - synthesis (SW and HW)
    - verification
- Disadvantages:
  - Difficult to compose (state explosion...) and decompose
  - Numerical computations cannot be specified
  - Sometimes overspecify implementation

    (sequencing is fully specified)

Finite State Machines

- Typical domains of application:
  - control functions
  - protocols (telecom, computers, ...)
- Different communication mechanisms:
  - synchronous
    (classical FSMs)
  - asynchronous
    (CSPs)

Data Flow Diagrams



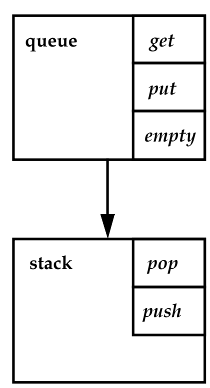
- Advantages:
  - Easy to use (graphical)
  - Powerful algorithms for
    - synthesis (scheduling and allocation)
    - verification
  - Explicit concurrency
- Disadvantages:
  - Efficient synthesis only for restricted models
    (no decision-making)
  - Control specified only in quasi-FSM style

Data Flow Diagrams

- 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)

Object-oriented Models

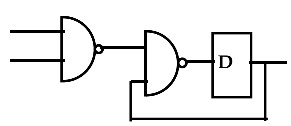
- Advantages:
  - Executable model
  - Natural hierarchical decomposition
  - Inheritance
    (simplifies extension and re-use)
- Disadvantages:
  - Strongly biased towards SW
  - Almost impossible to verify formally
    (too powerful)



Object-oriented Models

- Typical domains of application:
  - Rapid prototyping
  - Complex (mainly SW) systems
- Various CASE methodologies use OO

Hardware Simulation Models



- Advantages:
  - Powerful algorithms for
    - synthesis
    - verification
  - Standard
    (VHDL, Verilog, UDL/I)
  - Timing is handled explicitly
- Disadvantages:
  - Strongly biased towards HW
  - Not really standard, nor really formal...

Hardware Simulation Models

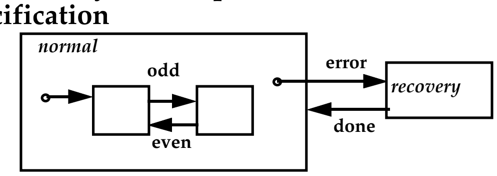
- Typical domain of application:
  - Hardware design
- Can be considered a least common denominator among SW and HW

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, ...
- Easy to use for control-dominated systems
- Simulation (animated), SW and HW synthesis, verification
- Generally extended with arithmetics
  (but no longer verifiable)
- Hierarchical states necessary for complex reactive system specification



Reactive Synchronous Languages

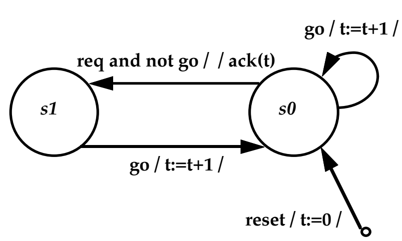
- Assumptions:
  - the system continuously reacts to internal and external *events* by emitting other events
  - events can occur only at discrete instants
  - *zero* (negligible) computation time
- Both control (Esterel) and data flow (Lustre, Signal)
- Very simple syntax and clean semantics
  (based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

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

Example

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;



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 (but memory-less and instantaneous) arithmetic and/or Boolean functions
  - ➠All the state of the system is under form of events
- Need rules that define the CFSM behavior

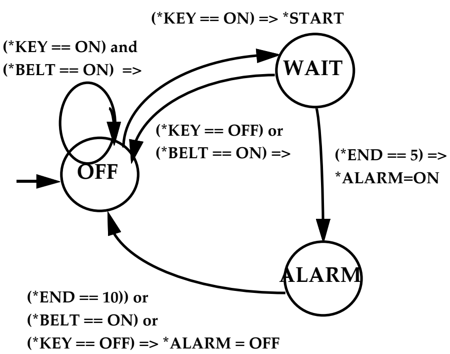
Example: Seat Belt Alarm Control

- "Natural Language" specification

"5 seconds after the key has been turned on, if the seat belt has not been fastened, then an alarm will beep for 10 s or until the key is turned off"

Example: Seat Belt Alarm Control

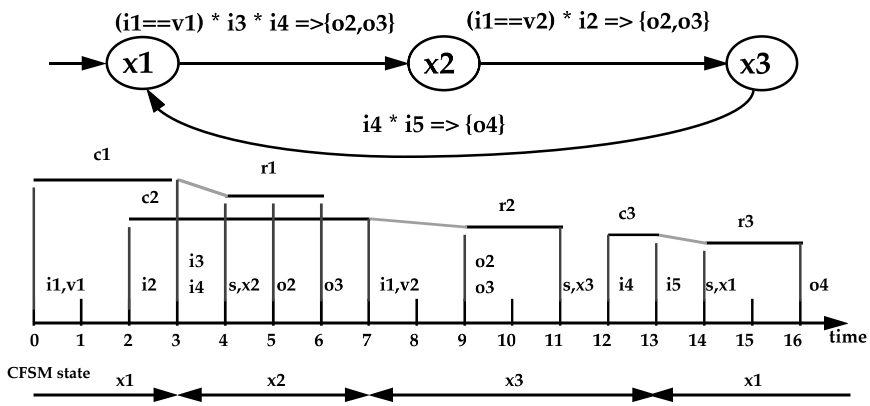
```
#This is an ESTEREL file

input BELT_ON, END_10Sec,...
output START_TIME,ALARM_ON,
relation END_5_SECS #
BELT_ON#..
loop
 await KEY_ON;
  do
   emit START_TIME;
   await END_5_SECS
   emit ALARM_ON
   ..............
 watching [BELT_ON or KEY_OFF]
 time_out
   emit ALARM_OFF
 end
end
```



CFSM Rules: phases

- Four-phase cycle:
  - ❶ Idle
  - ❷ Detect input events
  - ❸ Execute one transition
  - ❹ Emit output events

CFSM Trace Semantics



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: priorities

- Problem: how do we make sure all urgent events are processed immediately?
- Need to test if an event has not occurred yet
- Need the *weakest possible constraint* on the implementation

  - ➠Force the *detected* input events of the CFSM to be processed in order

Model of time

- Discrete:
  - Sufficiently accurate for synchronous systems
  - Feasible formal verification
- Model semantics: *Timed Traces* i.e. sequences of events labeled by time of occurrence

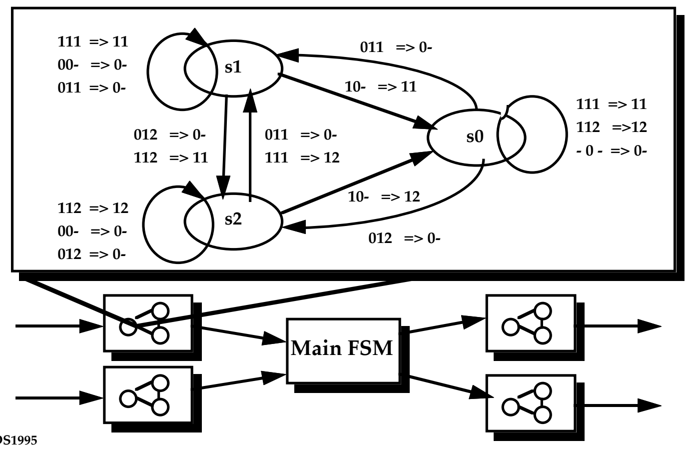
CFSMs and FSMs

- Best known automated formal verification methods are based on standard* FSMs
  - ➠Define an equivalent behavior to the CFSMs' one by using FSMs
- Event-driven model: self-loop until events are detected
- Depth-1 buffers on input and output events implemented as *non-deterministic* FSMs

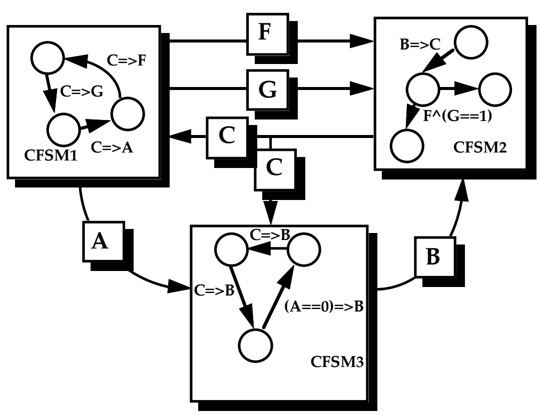
---

∗ Synchronous semantics

Buffer FSMs



Network of CFSMs: Depth-1 Buffers



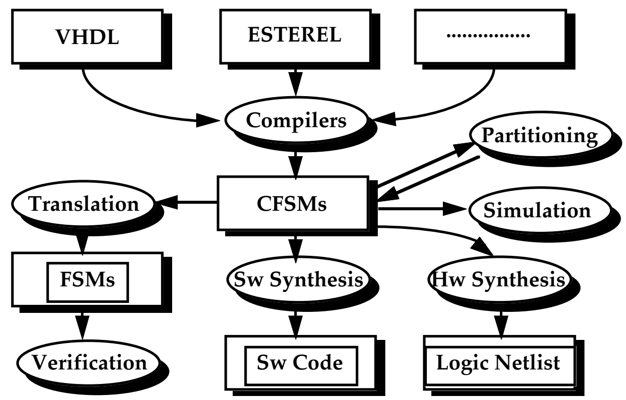
Software Hardware Intermediate FormaT

- CFSM network
  - Specific representation format: SHIFT
  - Unbounded-delay interpretation
- SHIFT description
  - List of input variables
  - List of output variables
  - Transition relation

Software Hardware Intermediate FormaT

- SHIFT transition relation: BLIF-MV extension
- SHIFT semantics vs. BLIF-MV semantics: reactive (unbounded delay) vs. synchronous (1 clock delay)
- Partition of a system: one or more CFSMs
  - ➠ one or more SHIFT *modules*
- Partition is either implemented in sw or hw

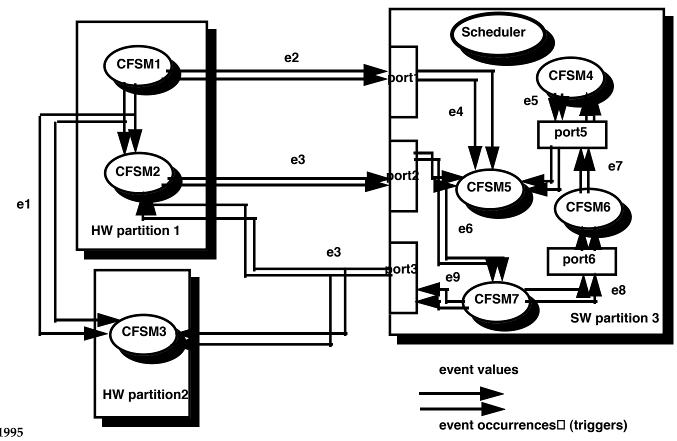
Our Proposal: Codesign Methodology



Hardware - Software Architecture

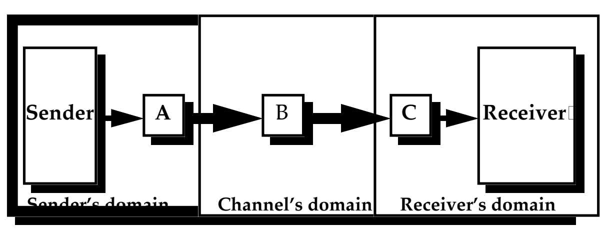
- Hardware:
  - One or more microcontrollers
  - ASICs, DSPs....
- Software:
  - Set of concurrent *tasks*
  - Scheduler
  - ➠Customized operating system

System Partitioning

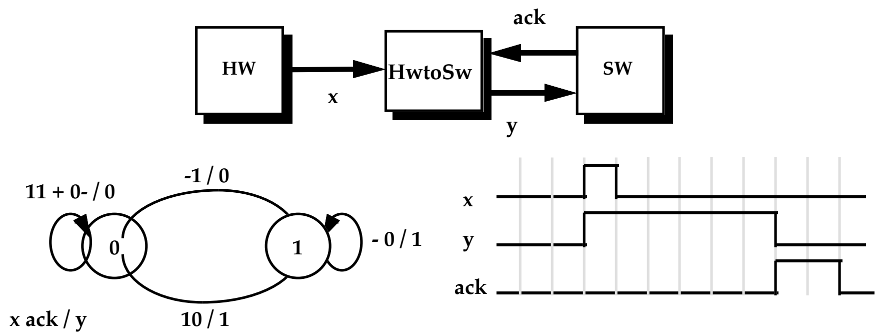


Interfaces Among Partitions

- Automatically generated
- Seven types of interfaces



An example of interface: hw to sw



Software Implementation

- Input:
  - set of tasks (specified by CFSMs)
  - set of timing constraints (e.g., input event rates and response constraints)
- Output:
  - Set of C procedures that implement the tasks
  - Scheduler that satisfies the timing constraints
- Minimizing:
  - the CPU cost
  - the 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: two-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
  - 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)

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

- Should be:
  - Low-level to allow detailed optimization and estimation
  - High-level to avoid excessive details (e.g. register allocation issues)
- Choice: *Decision Diagram*
  - Appropriate for control-dominated tasks
  - Well-developed set of optimization techniques
  - Augmented with arithmetic and Boolean operators, to perform computations

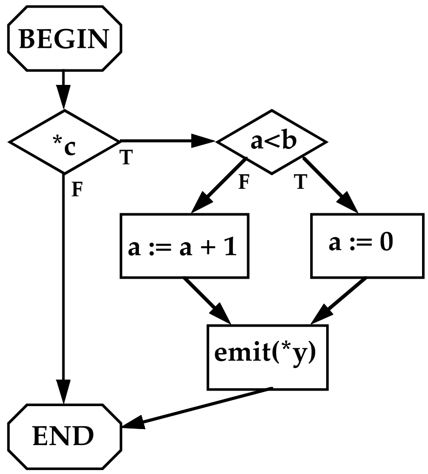
Our Representation: S-graphs

- 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)
```



S-graphs and Functions

- Execution of an s-graph can compute 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

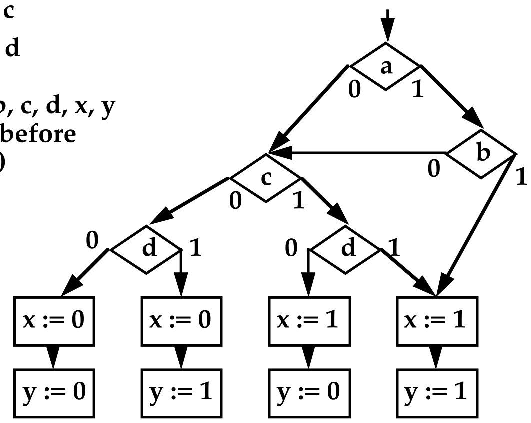
  $f = x\, f_x + x'\, f_{x'}$
- Result heavily depends on ordering of variables in expansion
  - inputs before outputs: TESTs dominate over ASSIGNs
  - outputs before inputs: ASSIGNs dominate over TESTs

Example of S-graph Construction

$x = a\, b + c$

$y = a\, b + d$

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

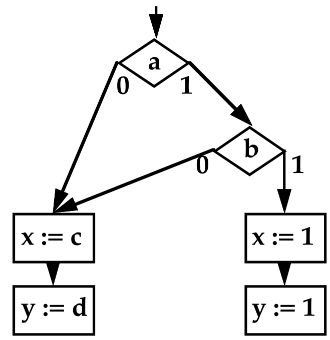


Example of S-graph Construction

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)

S-graph Implementation

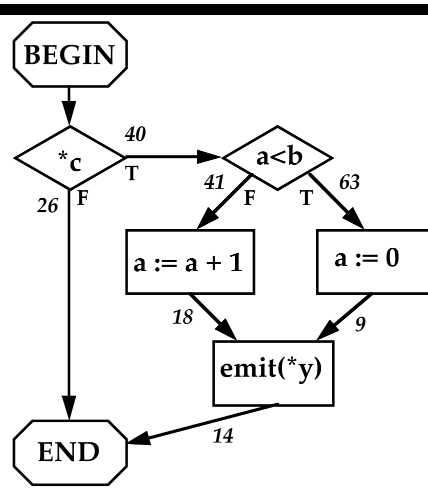
- Direct mapping into programming language or assembly code
  - Good prediction of final code characteristics
  - C is currently produced, leaving register allocation and instruction selection to general-purpose compiler
- Highly optimized but highly unstructured code
  - Need source-level debugging environment (with any of the chosen entry languages)

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 and 68HC11 processors

Performance and Cost Estimation

- 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 10% of profiling



Performance and Cost Estimation

- Timing information can also be used for cross-profiling
- Software generated from s-graphs accumulates elapsed clock cycles
- Perform simulation and gather profiling data:
  - event response times
  - dynamic analysis of bottlenecks
- Use estimated clock cycles for fast HW/SW co-simulation
- Processor selection at early stages of design

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)
- 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)

The Scheduling Problem

- 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)
- Scheduling algorithm choice is left to the user
- More work needed to do:
  - Verifiable scheduling (less CPU waste)
  - Automated constraint propagation

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 (Leyland, Liu '76):
    - highest I/O rate has highest priority
  - dynamic pre-emptive: Earliest Deadline First
    - CFSM with nearest deadline has highest priority

Open Problems with Scheduling

- 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

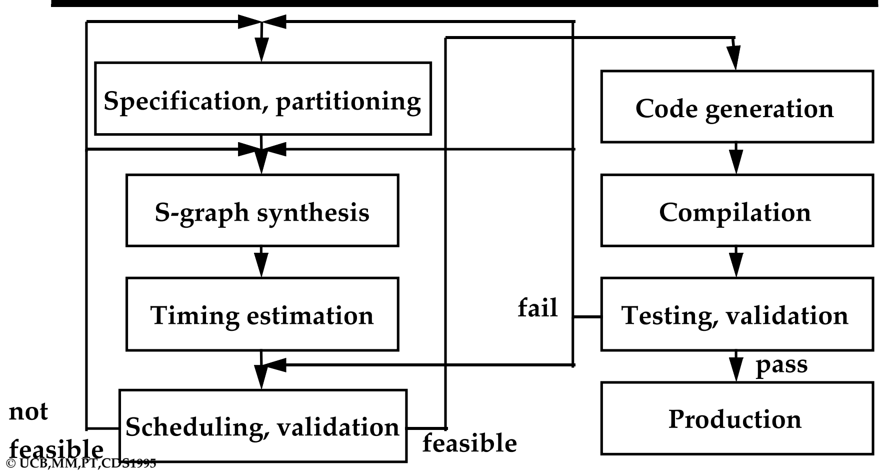
  (e.g., software-driven bus interface protocol)
- Automatic choice of scheduling algorithm, based on performance estimation and constraints

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 ?

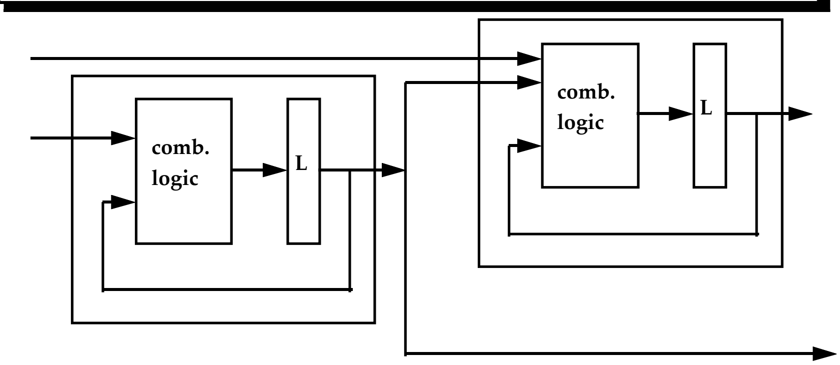
Software synthesis procedure



Hardware Synthesis

- SHIFT is interpreted as a 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 for rapid prototyping

Hardware Synthesis



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 them together ?
- Simulation can be used initially
  - Quick functional debugging
  - Rule out obvious cases (which 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*

Formal Verification

- HSIS
  - Language containment
  - Model checking (CTL)

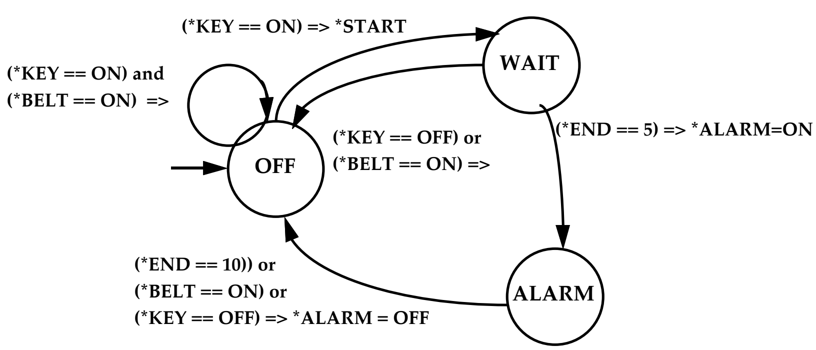
- System described by interacting NDFSMs

$$\text{CFSMs} \xrightarrow{\text{delay automata}} \text{NDFSMs}$$

Formal Verification

- Model described as a network of CFSMs (mapped into FSMs)
- *Time-independent and time-dependent* properties
- Bounded delay modelled by using non-determinism

Example of Formal Verification



Example of Formal Verification

- Untimed Property
  - AG(ALARM_ON --> AF(ALARM_OFF))
- Assumption: Non-zero unbounded delays
  - Property doesn't hold
  - Deduce reason for failure from error trace
    - Need tighter delay range
    - Specification refinement

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 (TCTL)
  - AG(ALARM_ON --> $AF_{<6\ s}$(ALARM_OFF))
- Property doesn't hold for all K, it only holds for:
  - 0 input delay, and
  - output delay ranges from 0 to 0.5 s

Example of Formal Verification

- A "Looser" timed property
  - AG(ALARM_ON-->$AF_{<11s}$(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 the 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)

What Did We Learn?

- Time unit must be chosen by trading off precision and speed
- Main problem: widely varying time scales (hw, sw and user domains)
  - ➠State explosion!
- Possible solutions:
  - Iterative addition of timing constraints (Balarin et al. CAV '93)
  - Conservative approximations (Burch TAU '92)

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

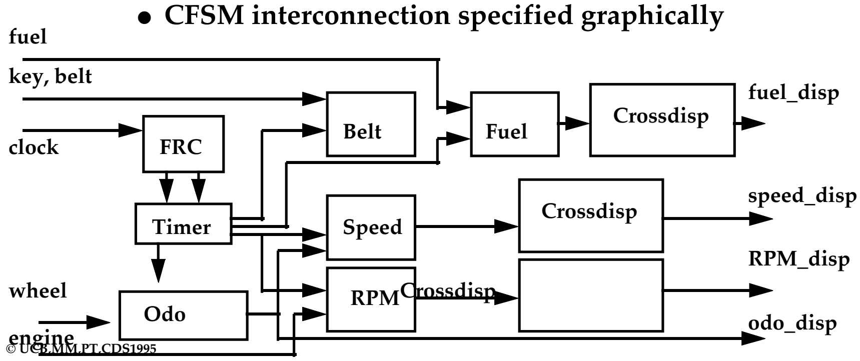
- Proposed methodology is based on a implementation - independent specification format
- Proposed hardware and software implementations are automatically generated
- Partitioning and synthesis are made easier by the common model used
- The FSM model derived from a CFSM is compatible with many formal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

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 in Esterel
  - automated translation from Esterel to CFSMs
- CFSM interconnection specified graphically



Software Synthesis Results

function	estimated timing		estimated size	measured timing		measured size
FRC	147	447	649	130	485	687
Timer	212	775	1341	180	901	1433
Odo	134	339	446	120	352	465
Speed	134	369	462	120	382	465
RPM	134	369	462	120	382	465
Belt	108	311	508	100	312	517
Crossdisp	251	16883	37196	210	23160	39284