CSI 2009 Tutorial

Formal Methods & Verification
(A Practical View)

Kavi Arya
CSE, IIT-Bombay
Presenter: Kavi Arya,
CSE Dept., IIT-Bombay,
Powai, India 400076

Tel. : +91 22 2576 7909
Fax. : +91 22 2572 7022
Email: kavi@iitb.ac.in

Material developed with Prof. S. Ramesh (GM Research)
Tutorial Outline

• Motivation & introduction to Formal Methods in Embedded Systems Design
• Introduction to System Verification
• Academic & Research Lab Verification Tools
• Modeling Formalisms: FSM, Statecharts
• Introduction to Esterel
• Esterel in Use
• Summary & Conclusions
Beware of the computer!

- From computers to embedded and networked SoCs
- Complete change in device interaction
- Growing number of critical applications
Embedded Systems in Automobiles
BMW 745i : Prelude To Complexity

Another Life Cycle Example : The Software Error
External view

The problem: software error, a desynchronization of the valvetronic motors

• Rough running engine, possibly stall
• Severity: 6 incidents in 5,470 Cars with 2 rear endings
  – “alleged injury” of BMW passengers
  – Fault of drunk or inattentive following drivers

“Engine malfunction, drive with moderation”
Bosch EMU For Four Wheeler (Multi Cylinder)

Source: Bosch Brochure: Ref 6
Motivation

- **Pentium SRT Division Bug**: $0.5 billion loss to Intel
- **Mercury Space Probe**: Veered off course due to a failure to implement distance measurement in correct units.
- **Ariane-5 Flight 501 failure**: Internal sw exception during data conversion from 64 bit floating point to 16 bit signed integer value led to mission failure.
  - The corresponding exception handling mechanism contributed to the processor being shutdown (This was part of the system specification).
Introduction to Formal Methods and Verification
Formal Methods

– Formal - Mathematical, precise, unambiguous, rigorous
– Static analysis
– No test vectors
– Exhaustive verification
– Prove absence of bugs rather than their presence
– Subtle bugs lying deep inside caught
Three-step process

• Formal specification
  – Precise statement of properties
  – System requirements and environmental constraints
  – Logic - PL, FOL, temporal logic
  – Automata, labeled transition systems

• Models
  – Flexible to model general to specific designs
  – Non-determinism, concurrency, fairness,
  – Transition systems, automata

• Verification
  – Checking that model satisfies specification
  – Static and exhaustive checking
  – Automatic or semi-automatic
Formal verification

- **Major techniques**
  - Equivalence checking
  - Model checking
  - Language containment
  - Theorem proving

<table>
<thead>
<tr>
<th>Model Spec</th>
<th>Tr. Systems/ Automata</th>
<th>Logic</th>
</tr>
</thead>
<tbody>
<tr>
<td>Logic</td>
<td>Model Checking</td>
<td>Th. Proving Eq. Checking</td>
</tr>
</tbody>
</table>
EQUIVALENCE CHECKING

• Checking equivalence of two similar systems (circuits)
• Comparison of two boolean expressions - BDDs
• Highly automatic and efficient
• Useful for validating optimizations, scan chain insertions
• Works well for combinational circuits
• Limited extension to sequential circuits
• Most widely used formal verification technique.
• Many commercial tools:
  – Design VERIFYer (Chrysalis), Formality (Synopsis), FormalPro (Mentor Graphics), Vformal(Compass), Conformal (Verplex), etc.
Model checking/Language Containment

• Another promising automatic technique
• Checking design models against specifications
• Specifications are temporal properties and environment constraints
• Design models are automata or HDL subsets
• Checking is automatic and bug traces
• Very effective for control-intensive designs
• Commercial and Academic tools: FormalCheck (Cadence), BlackTie (Verplex), VIS (UCB), SMV (CMU, Cadence), Spin (Bell labs.), etc.
• In-house tools: IBM (Rulebase), Intel, SUN, Fujitsu (Bingo), etc.
Theorem proving

- Theoretically most powerful technique
- Specification and design are logical formulae
- Checking involves proving a theorem
- Semi-automatic
- High degree of human expertise required
- Mainly confined to academics
- Number of public domain tools
  - ACL2 (Nqthm), PVS, STeP, HOL
    - ACL2 used in proving correctness of floating point algorithms
Formal verification (experiences)

– Very effective for small control-intensive designs-blocks of hundreds of latches
– Many subtle bugs have been caught in designs cleared by simulation
– Strong theoretical foundation
– High degree of confidence
– Hold a lot of promise
– Require a lot more effort and expertise
– Large designs need abstraction
– Many efforts are underway to improve
Systems verified

• Various microprocessors (instruction level verification):
  – DLX pipelined architectures, AAMP5 (avionics applications), FM9001 (32 bit processor), PowerPC
• Floating point units:
  – SRT division (Pentium), recent Intel ex-fpu, ADK IEEE multiplier, AMD division
• Multiprocessor coherence protocols
  – SGI, sun S3.Mp architectures, Gigamax, futurebus+
• Memory subsystems of PowerPC
• Fairisle ATM switch core
State of the art

- FSM based methods: ~ 500 registers
- STE: ~ 10 - 20k registers
- Equivalence checking: ~ million gates designs
- Simulation: million gates capacity
Challenges of formal verification

• Complexity of verification
  – Automatic for finite state systems (HW, protocols)
  – Semi-automatic in the general case of infinite state systems (software)

• State explosion problem
  – Symbolic model checking
  – Homomorphism reduction
  – Compositional reasoning
  – Partial-order reduction
Formal Modeling
Models

• High level abstractions of real systems
• Contain details of relevance
• Full Systems detailed and complex
• Physical components and external components
  – e.g. buses, schedulers, OS/network support software

• Modeling
  – Modeling is a (pre-) design activity
  – Models relatively easier to build
  – Higher level than behavioral models (C models)
    • early detection of bugs,
    • design space exploration and verification,
    • prototypes and synthesis
Formal Models

- Mathematical description of models
- Precise and unambiguous
- Consistent and complete

Formal Verification

- Applies to mathematical models and not to real objects (hence called Design Verification)
- Faithful models essential
  - False negatives (Spurious Errors)
  - False positives (Models pass but System fails)

- Simulation/Testing cannot be dispensed with!
Formal Modeling Languages

- Enable abstract and high level descriptions
- Real languages often ambiguous
- Variation in HDL semantics
- Real languages require more details and effort

Features
Limited and High Level Data Types
- Nondeterminism (arising out of abstractions)
- Concurrency (to structure large systems)
- Communication (for internal and external interaction)
- Fairness (abstraction of real concurrency and schedulers)
Example Modeling Languages

- Finite State Machines
- CSP, CCS, SDL, Promela (for Asynchronous Systems and Protocols)
- Esterel, CFSM (Embedded Controllers)
- Statecharts, UML (System level models)
Models of Hardware

- Hardware blocks are reactive systems:

  ![Diagram of a reactive system]

  - Reactive systems exhibit infinite behavior
  - Termination is a bad behavior
  - Timing/Causality information important
Finite State Machines

- Well-known model for describing control or sequential circuits
- An example (3-bit counter)

- State labels describe bit status
Another Example

- **A Traffic Light Controller**

- States HG - Highway green, FY – Farm road Yellow
- C - Car in Farm road,
- S,L - Short and long timer signal
- TGR - reset timer, set highway green and farm road red
States and Transitions

- States are abstract description of actual machine states
- decided by the states of latches and registers
- Finite no. of States
- No final state - reactive systems not supposed to terminate
- Edge labels - input/condition and output/action
States and Transitions

• Many Flavors of State Machines
  – edge labeled - Mealy machines
  – state labeled - Kripke structures
  – state and edge labeled - Moore machines
  – Labels
    • Boolean combination of input signals and outputs
    • communication events (CSP, Promela)
Semantics of Finite State Systems

• The above description is syntactic
• Semantics associates behaviors
  – Branching Time semantics
    – the tree of states obtained by unwinding the state machine graph
    – possible choices are explicitly represented
  – Linear Time Semantics
    – the set of all possible `runs' in the system
    – the set of all infinite paths in the state machine
Non-determinism

- 2-master arbiter,
- req_i - request from Master i
- This machine is nondeterministic
- In Idle state when req1 and req2 arrive.
- Non-determinism due to abstraction
- More than one behaviour for a given input
Concurrent
• A concurrent (and hierarchical) description of Counter
Concurrent Descriptions

- Compact and easy to understand
- Natural model for hardware and complex systems
- Clear semantics required
- Interleaved model and synchronous models
- Appropriate communication primitives
- Concurrent machines composed to a single global machine
- Global machine captures all possible executions
- Exponential blow-up
Fairness Constraints

- In general, not every run in a state machine is a valid behavior
- Arbiter example
  - the run in which master 2 is never granted the resource
- But all runs are included in transition systems
- Fairness constraints rule out certain runs
- Modeling abstraction of real-time or schedulers

Example

- Every request eventually considered
- The clock tick arrives infinitely often
Fairness Constraints

• Not required with a more concrete description
• But concrete description too complex to verify
• A given property may not require concrete details
• For verification, abstract designs are preferable.
  – proof is simpler
  – proof is robust under alternate implementations.
Generating Formal Models

• Pre-design activity

• Automatic Translation from circuits/HDL designs
  – States decided by the latches/registers in the ckt.
  – Exponential blow-up in the size (State-explosion problem)
  – Usually abstractions required
Design errors

Deadlock

• Look at state (1,1)

• **Unspecified Receptions**
  • State (1,1)
    – P1 can send message 2
    – P2 cannot receive this

• **Non executable interaction - 'Dead code'**
  • State 3 of P1 cannot be reached at all
Live lock/Divergence

• An example:

- Formal Verification generalizes early approaches to detection of such errors!
Formal Specification
Formal Specifications

• Verification involves checking that a design model meets its specification.
• Specification states what the system is supposed to do
• Design describes how this is done

Specification
• Describes unambiguously and precisely the expected behavior of a design.
• In general, a list of properties.
• Includes environment constraints.
• Symbolic logic or automata formalisms
• Consistency and Completeness
Specification of System (hw) blocks

• Properties and Constraints specify possible states and transitions
• They state set of possible valid `runs'
• Valid runs are infinite sequences (or trees) of states and transitions
• Formal specifications are finitistic and precise descriptions
• Classification of Properties:
  – Safety properties
    • "undesirable states are never reached",
    • "desirable things always happen".
  • Progress or Liveness Properties
    • "desirable state repeatedly reached"
    • "desirable state eventually reached"
Examples

Safety Properties
• A bus arbiter never grants the requests to two masters
• Message received is the message sent
• Elevator does not reach a floor unless it is requested
• At any time traffic is let either in the farm road or on the highway
• every received message was sent

Liveness Properties
• car on the farm road is eventually allowed to pass
• Elevator attends to every request eventually
• every bus request is eventually granted
• every sent message was received
Specification Formalisms

• Properties and Constraints specify permissible behaviours
• Behaviours are infinite runs (reactive systems)
• They are infinite objects, in general.
• We need finitistic representation of such infinite objects for precision
• Two Major formalisms:
  – Symbolic Logics: Linear and Branching Temporal Logics,
  – Automata
Temporal Logics

• Logics well-known for precise specification, amenable to symbolic manipulations.
• used in a variety of contexts:
  – Propositional Logic/Boolean algebra for combinational HW
  – Predicate logics for software
  – Higher order logics for language semantics.
  – Temporal logic for hardware and protocols.

• Qualitative temporal statements
• Examples:
  – If it is cloudy, eventually it will rain
  – It never rains here
Properties of Hardware blocks

• Temporal in nature
  – At any time only one unit is accessing the bus
  – Every request to access the bus is granted ultimately.

• Two Kinds of TL
  – Linear Temporal Logic (LTL):
    • Time is a linear sequence of events
  – Branching time temporal logic (CTL, CTL*):
    • Time is a tree of events
Model Checking
Automatic Verification

- Model Checking and Language Containment
- For finite state systems like Hardware blocks, protocols and controllers.
- Systems modeled as transition systems or automata
- Specifications temporal formulae (LTL, CTL) or automata
- Verification:
  - Model Checking: A finite state system or automaton satisfies a temporal logic specification iff it is a model of the formula.
  - Language Containment: An automaton model ($M$) of the system satisfies an automaton specification ($S$) if the language of $M$ is contained in that of $S$. 
Complexity of CTL model checking

- Algorithm involves backward traversal
- Linear on the sizes of both formulae and model
- Size of the model exponential in size of latches
- Reduction Techniques:
  - Symbolic Model checking Techniques
  - Compositional Verification
  - Symmetry based reduction
Verification
by
Theorem Proving
Theorem Proving

• Classical technique
• Most general and powerful
• non-automatic (in general)

Idea
• Properties specified in a Logical Language (SPEC)
• System behavior also in the same language (DES)
• Establish (DES $\rightarrow$ SPEC) as a theorem.
A Logical System

- A language defining constants, functions and predicates
- A no. of axioms expressing properties of the constants, function, types, etc.
- Inference Rules

A Theorem
- `follows' from axioms by application of inference rules has a proof
Proof

• Syntactic object
  \( A_1, A_2, \ldots, A_n \)
• \( A_1 \): axiom instance
• \( A_n \): theorem
• \( A_{i+1} \) - Syntactically obtainable from
• \( A_1, \ldots, A_i \) using inference rules.
Examples

• Propositional logic and its natural deduction system

• Prove $\sum_{i=1}^{N} i = N(N + 1)/2$, using Peano's axioms and mathematical induction
Full Adder

- sum := (x ⊕ y) ⊕ cin
- cout := (x ∧ y) ∨ ((x ⊕ y) ∨ cin)

Theorem: sum = x + y + cin - 2 * cout
Proof: Use properties of boolean and arithmetic operators.
Problems with the approach

- Verification is a laborious process
- Manual proofs could contain error
- If proof exists, system is correct otherwise, no conclusion.

Interactive Theorem Provers
- Ease the process of theorem proving
- Proof-Checking
- Decision Procedures
- Proof Strategies
- Theory building
- Many systems are available: Nqthm, PVS, HOL, Isabelle, etc.
Academic & Research Lab
Verification Tools
Verification tools

• A large number of tools exist
• Exhaustive review impossible
• For a comprehensive list and details:
  – Formal Methods Home Page:
    http://www.comlab.ox.ac.uk/archive/formal-methods.html
  – Centre for Formal Design and Verification of Software
    (CFDVS) at IIT Bombay
    http://www.cfdvs.iitb.ac.in
• BRIEF review of a VERY FEW model-checking tools
  in order now!
FC2toolset

Home page:

http://www-sop.inria.fr/meije/verification/

• Automatic Verification of Finite State Communicating systems
• Form the basis for Esterel verification
• Developed at INRIA and Ecole de Mines, Sophia Antipolis
• Based upon process algebra notation
• Modeling Language: CSP/CCS, Esterel (Xeve tool)
• Specification language: abstract state machines
Verification Approach

• Comparison of specification machine with the model after abstraction: use of Observational Equivalence, Symbolic Bisimulation

• Compositional Minimization and Abstraction

• A very powerful notion of abstraction (re-labeling complex sequence into a single label)

• Explicit and BDD representation of state machines
Example

DLX Controller

• An Esterel model
• 100s of states
Summary & Conclusions
Summary

• Formal verification of embedded systems (SOCs, etc.) has brought hard-core engineers and theoreticians on a common platform
• Model checking most successful so far
• Equivalence checking successful in restricted cases, catching up fast
• The gap between embedded system (VLSI) advancements and advancements in FV techniques can be filled to some extent by semi-formal methods
  – Hard to measure “success” for such techniques?
  – I might not have detected one out of 100000 bugs, but this might be the killer bug
Summary

• System verification will be an important area in future
• Simulation and emulation still predominant techniques used in industry and justifiably so
• However, formal methods ARE NEEDED when applicable and when one can’t afford to miss a bug
  – Success stories: Protocols, FSMs, specific processors, floating point arithmetic etc.
CSI 2009 Tutorial
Formal Methods & Verification

Esterel: Introduction
Esterel: Motivation
Embedded Software

Typical structure of a simple embedded Software

\textit{loop}

\textit{read} inputs/sensors;
\textit{compute} response;
\textit{generate} actuator outputs

\textit{forever}
Embedded Software (contd.)

• **Design Decisions**
  - How to read inputs?
  - How often to read inputs?
  - Which order to read the inputs?
  - How to compute responses?
  - How to generate the responses?
  - How often to generate?
The Simplest Approach

Round Robin Scheme

```
loop
    await tick;
    read S1; take_action(S1);
    read S2; take_action(S2);
    read S3; take_action(S3);
    forever
```

Tick is a time interrupt
Problems

• Processing speed decides the input rate!
• Fine for interactive systems not for reactive systems
• But it should be the other way around:
  – Characters coming at an network interface card
  – Video frame processing
  – Signals from pacemaker’s environment
• All sensors are treated identically
  – Some require urgent processing
Problems

• System response function of respective inputs
  – In general, it depends upon all inputs and
  – On the history - state dependent

• Fragile scheme
  – More sensors - more processing delay
The Most General Scheme

- Task1 || Task2 || ... || Task8

- Tasks
  - Sequential threads
  - Concurrently executed
  - can be scheduled and suspended
  - wait for specific time period or events
  - communicate with each other
The Most General Scheme

• Real-time OS (RTOS kernel)
  – Manages the tasks
  – Task communications
  – Timer services
  – Schedules the tasks for execution using various
  – Scheduling strategies
Challenge with RTOS

• Too much time and space overhead
• More complex design
• Writing and understanding concurrent tasks very difficult
• Race conditions, deadlocks, livelocks
• Concurrency model is asynchronous
• No timing guarantees
Challenge with RTOS (contd.)

- Priorities, scheduling
- System behavior highly unpredictable
- Or building predictable system is very challenging
- Analysis very difficult
- Thorough simulation required
Synchronous Approach

A novel Methodology

- Originated from three French groups
  - Through Esterel, Lustre, Signal
- Basis for Statecharts, stateflow
- Very successful in application domain
  - Lustre in SCADE (Telelogic)
    - Aerospatiale - Airbus 340 and 385
      (entirely designed using SCADE tool)
    - Schneider Electric for nuclear plants
Synchronous Approach (contd.)

– Esterel Technologies
  • Rafale bomber (successor of Mirage) by Dassault Aviation
  • TI (DSP Chips), ST Microelectronics (DVD Chips)
  • Intel and Motorola use Esterel tools
  • Cadence Lab. (for HW design)
  • POLIS HW-SW Co-design

– SIGNAL
  • Snecma - airplane engines

– Statecharts in I-logix Tool
  • Developed for avionics application
Main Features of Synchronous Approach

- A 3-level architecture
  1. Interactive (I/O) interface
  2. Reactive Kernel
  3. Data Management

- Each level requires different kinds of processing
- Separation of concerns
- Kernel is the most complex
Synchronous Execution

• Interface acquires inputs and forwards outputs,
  – Interrupt processing, reading sensor values,
  – Conversion of logical and physical I/O

• Reactive kernel periodically executed
  – Computes logical outputs given logical inputs.
  – Execution is atomic
  – No change of inputs during execution
  – Reaction

• Data Management by conventional sequential functions
  – called by reactive kernel
Synchrony Hypothesis

• Reaction is instantaneous
• Abstraction: reaction time is insignificant compared to the period
• Reaction is atomic
• This abstraction is realistic
• Can be checked - loop free computation
Synchrony Hypothesis

Other features:

• provides rich set of constructs for programming the kernel
  – Kernel is the most complex part
  – Interface, Data management programmed in host language
Synchrony Hypothesis (contd.)

- **Multiform notion of time**
  - Time like any other external event
  - Example:
    - Train must stop within 50 meters
    - Alarm raised when gas overflow limit reached

- **Kernel compiled into sequential automaton**
  - No tasks and no scheduling overhead
  - No priorities, no race conditions

- **Predictability is very high**
  - delay 5 sec; delay 5 sec = delay 10 sec
State Machines

• State machines are flat with no structure
• Esterel provides rich structure
  – Large machines difficult to understand

Consider the specification:
  – Emit O as soon as inputs A and B arrive
  – Reset each time if input R occurs
FSM Implementation:
Esterel Implementation

module ABRO:
  input A,B,R;
  output O;
  loop
    do
      [ await A || await B ];
      emit O
      watching R
    end
  end module

• The code more compact than FSM
• Each signal appears exactly once!
• Statechart descriptions also more compact
Esterel

- An imperative language for programming reactive kernels
- It is a textual language!
- An Example: Seat-Belt Controller
- Here is a requirement:

"Five seconds after the key is turned on, if the belt has not been fastened, an alarm will beep for five seconds or until the key is turned off"
module belt_control:
  input reset, key_on, key_off, belt_on, end_5, end_10;
  output alarm(boolean), start_timer;
loop
  do
do
  emit alarm(false);
every key_on do
do
  emit start_timer;
  await end_5;
  emit alarm(true);
  await end_10;
  watching [key_off or belt_on];
  emit alarm(false);
end
  watching reset
end
Esterel Solution

- Structure reflects closely the requirements
- Constructs are high level
  - loop, every, watching, emit, await
- Sounds similar to the informal language phrases
- But having a precise semantics
- Easy to see the correctness of solution
- Nice syntactic structure
- Compare it with the state machine solution
Behavior of program:

**Belt - Controller**
Esterel Implementation

```
module ABRO:
  input A,B,R;
  output O;
  loop
    do
      [ await A || await B ];
      emit O
    watching R
  end
end module
```

- The code more compact than FSM
- Each signal appears exactly once!
- Statechart descriptions also more compact
Layered Organization: Conventional View

- Application Tasks
- Scheduling, IP Communication
- I/O Handlers
- Hardware

OS
Layered Organization: Esterel View

- Esterel Application
- I/O Handlers
- Bare Machine

Esterel Program + Data Handler
Layer Interaction

Interface

Signals & sensors

Esterel (Reactive kernel)

Data Handling part

Function calls.
Execution Model

- execution is a series of reactions
- invoked from an external 'main' program repeatedly at discrete points of time
- one reaction per invocation
- control returns after each reaction
Reaction

• Considered instantaneous!
• Control flows from one statement to its next
• Concurrent control flows
• Input signals do not change in status nor in their values within a reaction.
• Output and local signals may change
• Signal presence tested and variables updated
• Reaction proceeds until *pause* is encountered
Reaction

• Reaction stops when *pause* is encountered in all active threads
• Next reaction starts from the next statement
• Status and values of input signals are reset at the end of reaction
• New values are set by the environment
Advantages of Esterel

- Model of time gives programmer precise control
- Concurrency convenient for specifying control systems
- Completely deterministic
- Finite-state language
- Execution time predictable
- Much easier to verify formally
- Can be implemented in hardware as well as in software
State Machines

• State machines are flat with no structure
• Esterel provides rich structure
  – Large machines difficult to understand

Consider the specification:
  – Emit O as soon as inputs A and B arrive
  – Reset each time if input R occurs
FSM Implementation:

Esterel Implementation

module ABRO:
    input A,B,R;
    output O;
    loop 
do
        [ await A || await B ];
        emit O 
        watching R
    end
end module

• The code more compact than FSM
• Each signal appears exactly once!
• Statechart descriptions also more compact
Esterel: Basic Features and Constructs
An Esterel program

• Describes the behavior of the reactive kernel
  – Has rich set of constructs for programming the kernel
  – Kernel is typically finite state

• Interacts with its environment through an abstract interface
  – Signals and Sensors are the means of communication
  – Input, Output and Local signals
  – Sensors are inputs only
An Esterel program (contd.)

• Has minimal data processing functions
• Uses the data handling part for major data processing
  – Functions and Tasks are the means of communication.
  – Global and Local variables are used for communication
  – **Host language** support - C, C++, Ada
Signals and sensors

- Signals are the novel means of communication
- Idea from hardware systems
- Software abstractions of the interface
- Signals can be pure or valued
  - pure signals have two status 'presence' or 'absence'
  - valued signals when present carry values
  - values are typed, like integer, boolean, string, float
- Signals are transient! - reset at the end of a reaction
Signals and sensors

- Environment communicates by *setting* input signals
- Program communicates back via output signals
- Local signals are used for communication between concurrent modules
- Has a no. of constructs for handling signals
  - *emit S, await S, present S then* ...
- *Tick* is a special signal always present
- *Sensors* are special signals used as input only
Variables and Expressions

• Esterel is an imperative language and hence uses variables

• Variables can store different types of values
  – integer, boolean, string, float

• Variables retain values until updated (across reactions)

• Variables can be local to a block of statements, a procedure or function or global

• No sharing of variables with the environment
Variables and Expressions

• No sharing of variables between concurrent threads
• Variables are means of communication along a single sequential thread
• The 'race problem' is absent!
• Expressions can be formed out of variables
Esterel Tools & Illustrative Examples
Esterel Tools

• Compiler
  – C-code for simulation
  – C-code for target implementation in SW
  – `bliff` code for implementation in HW and verification

• Simulator - Xes

• Verifier
  – Xeve
  – fctools
  – atg
Make file - An example

all: blif genc obj es
blif: esterel –Lblif : -soft -causal -main simple
    auto.strl
genc: esterel -simul -main simple auto.strl
obj: gcc -w -c auto.c
es: xes auto.o
The simulation tool - Xes

Figure 1: Simulation Panel
Verification Tools

- **Xeve**
  - Converts bliff code into fc2 format
  - Hiding of signals
- **FcTools**
  - Automata minimization
  - Automata abstractions
- **Atg**
  - Display of automaton
  - Manual inspection
Code generator

• Compiler (without special options) produces
  – reactive engine that implements the automaton
  – one function for each input signal
• The reactive engine is just a procedure
• User has to write a main program that calls the engine
• User has to write a function for each output signal and sensor
• When to call the engine, which input signals are simultaneous decided by the user
• The engine calls the output functions
Example

Main::

- every milli second
- for every input that has occurred
  - call f_input(par);
  - call f_kernel;
- end

• Assume interface functions run concurrently with the main program
• Input parameter gets the value from specific registers or memory
• f_kernel calls output functions corresponding to emitted outputs
A Simple Example

An Auto Controller:

• Initially waits for the engine to be on
• When car is running, acceleration/deceleration controls the throttle valve
• Car stops when the ignition is off
The code

module simple:
input ignition_on, ignition_off, accel;
output control_throttle;
loop
  abort
    await ignition_on;
  every accel do
    emit control_throttle
  end every
when ignition_off
end
end module
Use of Esterel Tools

• Compile the program to simulate (-simul option)
• Compile the program to generate bliff code
• Use xeve to generate automaton
• Use atg for visual display
Automaton for Simple controller

#ignition_off.#ignition_on+!'ignition_off

#ignition_off!'ignition_on

!ignition_off

#accel.#ignition_off+!'accel.#ignition_off!.control_throttle
A Complex Controller

Additional safety feature:

• Monitors status of the door (closed or open) while in motion
• Sends out alarm when door opens
• Ensures that doors are closed while starting
module complex:
  input ignition_on, ignition_off, accel;
  input door_opened, door_locked;
  output alarm, control_throttle, door_lock;
  loop
    abort
    await ignition_on;
    emit door_lock;
  loop
    await door_locked;
    abort
    every accel do
      emit control_throttle
    end every
    when door_opened;
    emit alarm;
    emit door_lock;
  end;
  when ignition_off
  end
end module
Automaton for Complex controller
Esterel & Robotics
Basic components of the robot

- Sensing
- Locomotion
- Control
- Intelligence
- Power
- Communication
Spark 4 Configuration

Microcontroller
ATMEGA128 / ATMEGA32 / ATMEGA16

Sensors
– 3 Linear infrared range finders
– 3 White line sensors
– 2 Shaft encoders
– 1 Directional light sensor
– Battery voltage
– 3 Infrared proximity sensors
– Servo mounted sensor pod
– GPS Receiver
– Gyroscope and Accelerometer
– Magnetometer
– Wireless colure camera
– Ultrasound scanner

ERTS LAB
CSE IIT BOMBAY
Spark 4 Configuration

Indicators
– 16 x 2 LCD
– Buzzer
– LED indicators

Communication
– ZigBee (IEEE 802.15.4)
– WiFi (IEEE 802.11)
– GSM-GPRS
– Infrared
– Wired RS232 serial
– Control Area Network (CAN)
Spark 4 Configuration

Power
- Onboard Lithium ion battery
- Auxiliary power

Motors
- 2 Ultra low power geared DC motors

Other
- Programming through Boot Loader on USB port
- In system programmer
- 6 User selectable modes
- Fast 1 hour Lithium ion charger
Synchronous Programming

• Spark 4 can be programmed using synchronous languages, like Esterel.

• Done by ERTS lab (CSE/IITB)
  [http://www.cse.iitb.ac.in/~erts](http://www.cse.iitb.ac.in/~erts)

• Along with Nex Robotics Pvt.Ltd.
  [http://www.nex-robotics.com](http://www.nex-robotics.com).

• Makes use of Spark 4 API, tools and APIs
Esterel Programming

• Applications can be written in Esterel
• An extension of Esterel compiler is available
• Need to use specific names for inputs and outputs
  – touch_1, touch_3, motor_a_speed, motor_b_speed, cputs etc.
• Here is an example
A simple program

module lego1 :
  input TOUCH_1, TOUCH_3;
  output MOTOR_A_SPEED(integer),
       MOTOR_C_SPEED(integer),
       MOTOR_A_DIR(integer),
       MOTOR_C_DIR(integer),
       CPUTS(string);
  relation TOUCH_1 # TOUCH_3;
  constant MOTOR_FWD, 
       MOTOR_REV, 
       MAX_SPEED : integer;
var t : integer in
    emit MOTOR_A_SPEED(MAX_SPEED/2);
    emit MOTOR_C_SPEED(MAX_SPEED/2);

loop
    emit MOTOR_A_DIR(MOTOR_FWD);
    emit MOTOR_C_DIR(MOTOR_FWD);
    emit CPUTS("fwd");
    await [TOUCH_1 or TOUCH_3];
    present TOUCH_1 then
        t:=1;
    else
        t:=3;
    end present;
emit MOTOR_A_DIR(MOTOR_REV);
emit MOTOR_C_DIR(MOTOR_REV);
emit CPU_TS("rev");
await 100 tick; % 1 tick = 1 ms
if t=1 then
    emit MOTOR_A_DIR(MOTOR_FWD);
    emit CPU_TS("right");
else
    emit MOTOR_C_DIR(MOTOR_FWD);
    emit CPU_TS("left");
end if;
await 100 tick; % 1 tick = 1 ms
end loop
end var
Compiling

• This program can be simulated
• Compiled using the extended compiler to generate C code
• Use AVR cross compiler to generate code and download onto the Spark 4
• Look at the web-site for more examples
Summary

• Various models reviewed
  – Flat Finite State machines
  – Hierarchical and Concurrent State Machines (Statecharts)
  – Textual analog – Esterel
  – Not reviewed Sequential models, Data Flow Models, Discrete Event Models

• Each model suitable for particular application
  – State Machines for event-oriented control systems
  – So with Esterel

• Real systems often require mixture of models

• Modeling tools/ lang. should have combination of all the features

• Esterel models realtime systems
  – At a high level of abstraction and
  – Lends itself well to formal verification (open source tools such as Xeve)
Conclusion

• Functional verification of Embedded Systems critically dependent on verification (+re-use of Cores/IPs/VCs)

• Semi-Formal approach based on application of different combinations of verification technologies seem important. No single approach will suffice.

• Formal & executable specifications have to be used at all levels of design hierarchy.

• High degree of automation needed to over-come complexities inherent in Embedded Systems of future.

• Lots of new formal modeling languages and formal specification and verification tools

• As systems become more complex, formal specification and verification is no more a luxury and more the rule esp. for safety-critical systems

• Exciting Area for Research
Bibliography
Papers

Papers

Books

Important web-sites:

- http://www.comlab.ox.ac.uk/archive/formal-methods.html
- http://www.csl.sri.com
- http://dimacs.rutgers.edu/Workshops/SYLA-Tutorials/program.html
- http://godel.ece.utexas.edu/texas97-benchmarks/
- http://citeseer.nj.nec.com/
- http://www.rational.com/uml (Universal Modelling Language HOME-PAGE)
Conference Proceedings

- Computer Aided Verification (CAV)
- Formal Methods in Computer Aided Design (FMCAD)
- International Conference on Computer-Aided Design (ICCAD)
- International Conference on Computer Design (ICCD)
- Design Automation Conference (DAC)
- Asia South Pacific Design Automation Conference (ASPDAC)
- International Conference on VLSI Design (VLSI)
- Advanced Research Working Conference on Correct Hardware Design and Verification Methods (CHARME)
Journals/Magazines

• IEEE Design and Test of Computers

• IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems

• IEEE Transactions on Computers

• IEEE Transactions on VLSI Systems

• ACM Transactions on Design Automation of Electronic Systems

• Formal Methods in System Design

• Formal Aspects of Computing
Other References

• Edward A. Lee, *Overview of the Ptolemy Project*, Technical Memorandum No. UCB/ERL M03/25, University of California, Berkeley, CA, 94720, USA, July 2, 2003
Appendix:
Academic & Research Lab
Verification Tools (contd.)
VIS

- VIS Home page:
  http://www-cad.eecs.berkeley.edu/Respep/Research/vis/

- Verification Interacting with Synthesis.
- A research prototype tool (Proof of Concept)
- U. California, Berkeley (Brayton et. al.)
- U.Texas, Austin, U.Colorado, Boulder
- VIS integrates verification, simulation and synthesis of finite state HW system.
Modeling Language

- A subset of Verilog
- VHDL and Esterel planned
- The back-end language is BLIFF-MV

Specification Language

- CTL, CTL* and Automata
Verification

- Symbolic Model Checking
- Language Emptiness Check
- Equivalence Checking of Combinational designs
- Speciality:
  - Simulation and Synthesis
  - Various representations of Boolean functions (BDD, MDD, etc.)
  - variable ordering heuristics
Specification Examples

• $AG\ ((\text{bit}[0] = 1) \rightarrow AX\ (\text{bit}[0] = 0))$
  Flipping of the zeroth bit in a counter.

• $AG\ ((\text{Req} = 1) \rightarrow AF\ (\text{Ack} = 1))$
  Every Req is greeted with an Ack eventually

• $AG\ (EX:5\ (\text{State} = TRST))$
  Always state TRST is reached in 5 steps.
SMV

• Home page:
  – http://www-cad.eecs.berkeley.edu/kenmcmil/smv/
• Experimental Research tool from Cadence Berkeley Labs. (McMillan)
• Originally from CMU (McMillan's Ph.D. thesis)
• Modelling Language: Interacting State Machines, Synchronous Verilog
• Specification Language: CTL, LTL
• Verification Approach: Symbolic Model Checking
• Compositional and Symmetry-based Verification Strategy
Modeling language

- Interacting state machines
- Synchronous and asynchronous concurrency
- Allow modular and hierarchical description of finite state systems
- Finite data types: Booleans, scalars, fixed arrays
Examples:

1. MODULE MAIN
   VAR
       request : boolean;
       state : {ready, busy}
   ASSIGN
       init(state):=ready;
       next(state):=case
           state=ready&request: busy
           1 : {ready,busy } ;
       esac;
   SPEC
       AG (request ® AF state = busy)
Examples:

1. Old Syntax
   • `request' unconstrained (input)
   • non-deterministic assignment
   • OBDD representation of tr. reln. constructed and SMC performed.
Example - 3bit counter

MODULE main
VAR
    bit0 : counter cell(1);
    bit1 : counter cell(bit0, carry-out);
    bit2 : counter cell(bit0, carry-out);
SPEC
    AG AG bit2. carry-out
MODULE counter cell(carry-in)
VAR value : boolean;
ASSIGN
    init(value) :=0;
    next(value) := value+carry-in mod 2;
DEFINE
    carry-out : = value & carry-in
Example - 3bit counter

- Assign sections of bit0, bit1 and bit2 execute in parallel
- Data dependence respected
STeP

Homepage:
http://rodin.stanford.edu/

- Stanford Temporal Prover
- Interactive theorem proving tool for verification of concurrent and reactive systems
- Combines model-checking and deductive approaches and verifies
  - parameterized(N-component) circuit designs
  - parameterized(N-process) programs and
  - programs with infinite data-domains
STeP

- Modeling Language: SPL and fair Transition Systems
- Specification Language: LTL and First Order Logic
- Verification Approach: Theorem Proving and Model-checking
STeP overview

- Temporal Logic Formula
- Reactive System (SPL) Program
- Hardware Description
- Fair Transition System
- Model Checker
- Automatic Prover
  - Verification Rules
  - Strengthening of Invariants
  - Propagation
- Bottom-up Invariant Generator
  - First-order Prover
  - Simplification
  - Decision procedures
- Interactive Prover
  - P-valid
  - Counter example
- User
  - P-valid
  - Debugging Guidance

158
Verification in STeP

• Very powerful theorem prover
• Automatic Invariant Generation,
• Collection of Simplification rules
• Decision procedures for linear arithmetic, arrays, bit vectors, etc.
• Generation of Verification Conditions
• BDD simplification
• Rich set of tactics and tacticals
An example SPL Program

main ::
[
  in a : int where ((a  127) ^ (a > - 128))
  in b : int where ((b<127) ^ (b > - 128))
  out c, d : int
  if (a > 0 ^ b > 0) then [
    if (a > b) then [ 12 : skip;c:= a-b]
    else [ 13 : skip;c:=b-a]
  ]
  else if (a < 0 ^ b<0) then [ 14:skip;c:= -(b + 1)]
    else [15: skip; c:=a + b ]
]
Specification file

SPEC
PROPERTY
\( P1 : l2 \rightarrow ((a - b) \leq 127) \land ((a - b) \geq -128) \)

PROPERTY
\( P2 : l3 \rightarrow ((b - a) \leq 127) \land ((b - a) \geq -128) \)

PROPERTY
\( P3 : l4 \rightarrow (-b + 1 \leq 127) \land (-b + 1 \geq -128) \)

PROPERTY
\( P4 : l5 \rightarrow ((a + b) \leq 127) \land ((a + b) \geq -128) \)
Examples

• Leader-Election algorithms

• Needham-Schroeder Security protocol

• Ricart and Agrawala's mutual exclusion algorithm

• Bus scheduler verification
  http://rodin.stanford.edu/case-studies/index.html