Globally Asynchronous Locally Synchronous Systems (GALS): Modeling and Verification

S. Ramesh
CSE Dept.
IIT Bombay
INDIA

Concurrent Systems

- Multiple threads or processes
- Most applications are concurrent in nature
  - Hardware Designs, Operating Systems
  - Networking Software, Embedded Software
  - Auotomative Electronics, System-on-Chip Solutions
- Concurrent Systems hard to develop and verify
- Deadlocks, Livelocks, Fairness, Mutual Exclusion
- Irreproducibility of errors
- Exponential number of runs
- Very Little training in curriculum

Model-based development

- Model-Verify-debug-refine(code-generate) paradigm
- Models are abstract descriptions of relevant behaviors
- High level, platform independent descriptions
- Much simpler and more general than real implementation
- Precise formal semantics (formal languages)
- Verification and performance analysis possible
- Early bug removal, evaluation and design space exploration
- Models refined to implementation automatically
- Correct models lead to correct implementations
- verification of implementation related to models
- Various modeling languages
- Statecharts, Esterel, UML, State machines, Petri nets, SDL

Classical Models of Concurrent Systems

Asynchronous

- Loosely coupled processes
- Concurrency is physical (run-time tasks)
- Communication takes time and usually between pairs
- Nondeterminitistic behaviour
- Useful for pure distributed applications
- CSP, CCS, ADA, SDL, OCCAM, $\cdots$

Synchronous

- Tightly coupled processes
- Logical concurrency (for SW implementation)
- Instantaneous broadcast communication.
- Deterministic behaviour
- Useful for localized embedded applications
- Esterel, Lustre, Signal, Statecharts, $\cdots$

Inadequacy of Classical Models

- Many recent applications are neither synchronous nor asynchronous
  - Distributed Control Applications
  - Industrial Process Controllers
  - Many common embedded applications
    - Aircraft and automobile controllers, ATM networks
  - Multi-agent Robotics
  - System-on-Chip solutions
- These applications consist of
  - Network of Reactive Nodes
  - Nodes have independent I/O interfaces & clocks
  - They synchronize & exchange messages with each other
  - System-wide global constraints
  - Two kinds of concurrency & interaction patterns
- They are called Globally Asynchronous Locally Synchronous (GALS)

Home Page

Title Page

Go Back

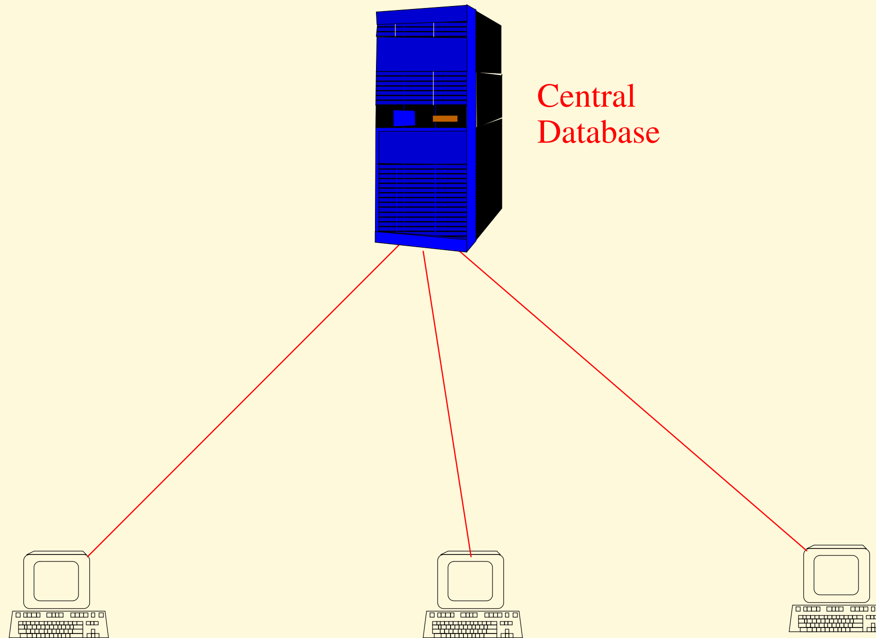
Full Screen

Close

Quit

Example 1:

Automatic Teller Machines Network:



ATM Network is Locally Reactive

- Sensing the card
- reading the user input
- validation, rejection/acceptance

Global Computation

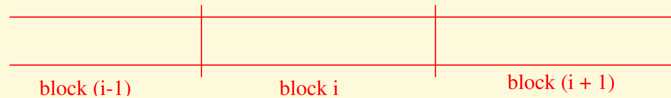
- connects to the central database
- checks the account status
- updates the accounts

Example 2: InfoPhone

- A standard example for OMAP application
- A High Level view:
- Network of ARM Core, DSP Core, Web server
- Each run at different clocks
- Three different programs run on each of these
- ARM program processes user commands
- DSP does speech processing
- Web server responds to the queries
- Local Computation with Global Constraints

Example 3:

Track Controller:



- Trains enter/exit blocks
- cannot enter without permission
- cannot exit without indication
- Safety property: No two trains in the same block
- Liveness property: Trains able to move from one block to another
- Problem : Design appropriate controllers for each block

GALS Solution :

- A Distributed Controller Network
- One controller per block
- each controller is locally reactive
  - senses entry/exit of trains
  - exchanges signals with the train on the block
- Adjacent controllers talk to each other for controlling entry/exit of trains

CRSM

- Communicating Reactive State Machines
- A language for such GALS systems
- Combines capabilities of asynchronous & synchronous languages
- Derived from our earlier works
  CRP (Berry, Ramesh, Shyam)
- Pictorial Language like UML, Statecharts, Stateflow

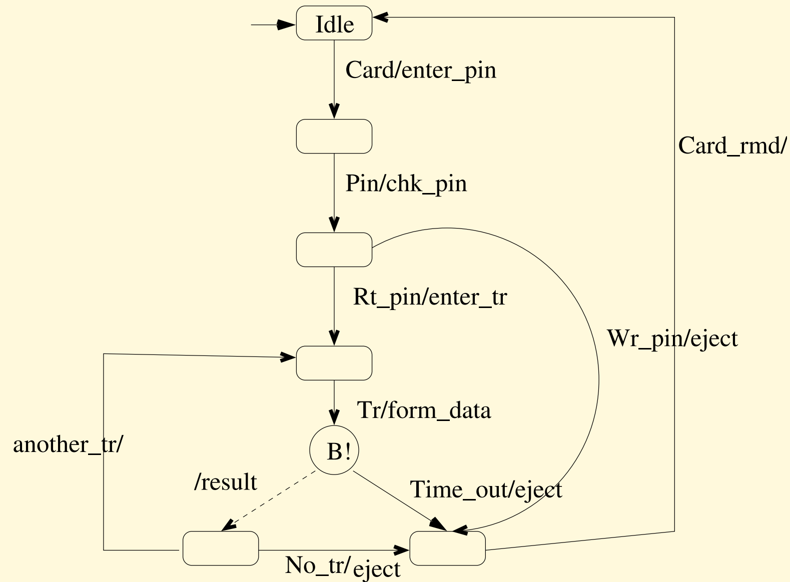
A CRSM

- Network of Reactive nodes
- Reactive nodes have independent I/O interface and clocks
- Synchronize to achieve system-wide global constraints
- Two types of concurrency primitives
- Two types of communication

CRSM Nodes

- Structured Mealy Machines
- Argos/Statecharts

Simple CRSM Node

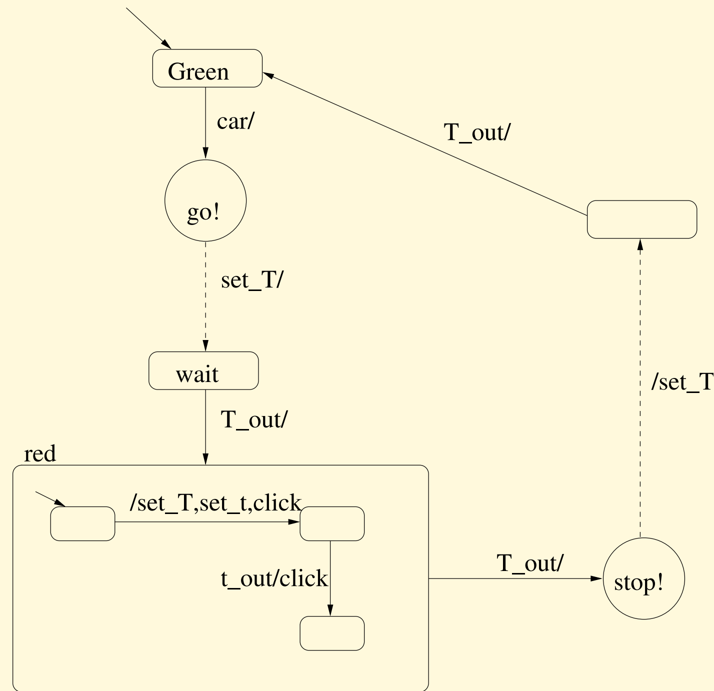


Nodes

- Normal states
- Rendezvous or communication States for interaction between nodes
- These states have channel names as labels
- Communicating nodes have same channel names
- Channels shared between exactly two nodes
- Normal and Special Exit edges
- Exit edges for communication states
- Leaving via exit edges when communication is completed
- Waiting for communication can be preempted
- Edge Labels, in general are:
  - b/o
  - b - booleans expression over input signals
  - o - set of output signals
- Start state, no final state (Reactive Systems)

Composition Operators of CRSM

1. Hierarchical Composition



- Red is a super state
- Entry to a super state
  - entry to a sub state through default arrow
- Exiting a super state
  - Preempting the ‘sub’ computation

Home Page

Title Page

Page 15 of 48

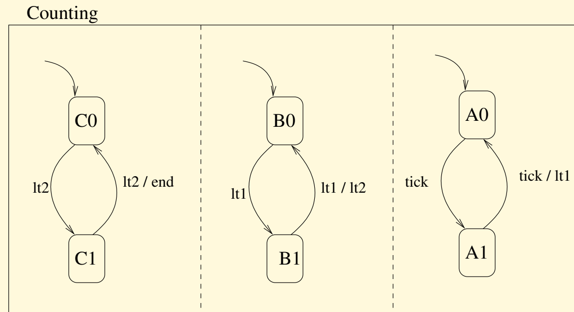
Go Back

Full Screen

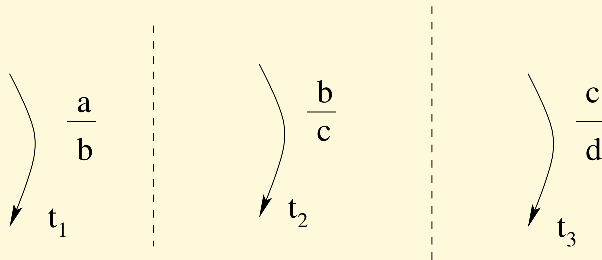
Close

Quit

2. Synchronous Parallel Composition

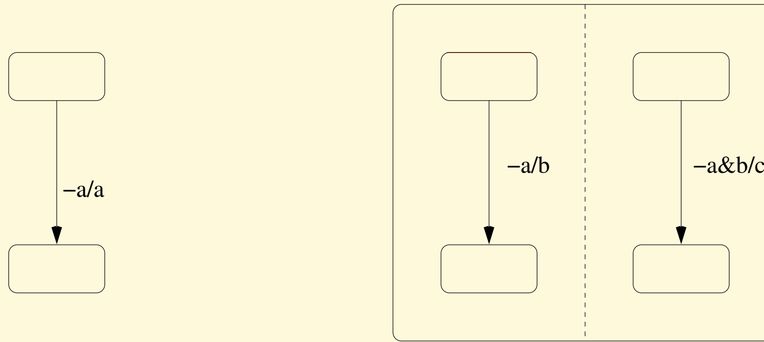


- Multiple Control Points
- Transitions can trigger one another (Broadcasting)
- Simultaneous Transitions (Synchrony Hypothesis)



- If ‘a’ is input, t_1, t_2, t_3 all are taken!

Causality Problems



- Absence of behaviours
- Nondeterminism
- correct programs are reactive, deterministic and causal
- Execution is a series of macro steps
- Each macro step is a consistent, complete and causal series of micro steps

Home Page

Title Page

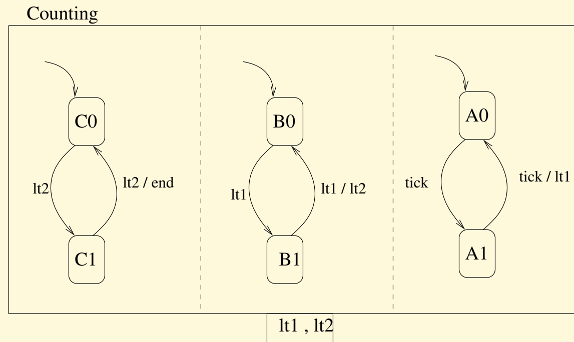
Go Back

Full Screen

Close

Quit

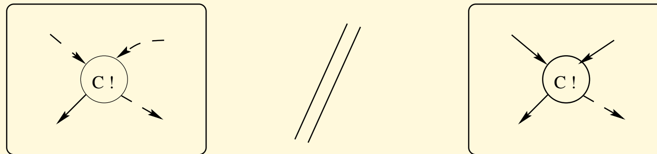
3. Signal Hiding



- internal ‘lt1,lt2’ invisible outside
- external ‘lt1,lt2’ cannot influence

Network of CRSM nodes

$$N_1 \; // \; \dots \; // \; N_k$$

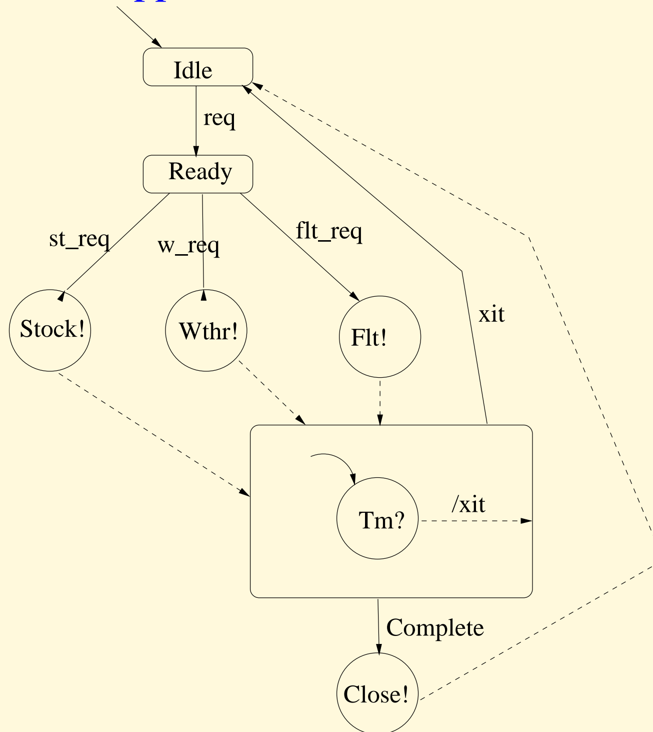


- rendezvous nodes agree to take exit edges ‘simultaneously’
- Synchronised Transitions
- clock ticks at communication points are synchronized
- Waiting for communication is preemptible.

Example: InfoPhone

- Consists of three nodes
- ARM, DSP and Web

ARM application



Home Page

Title Page

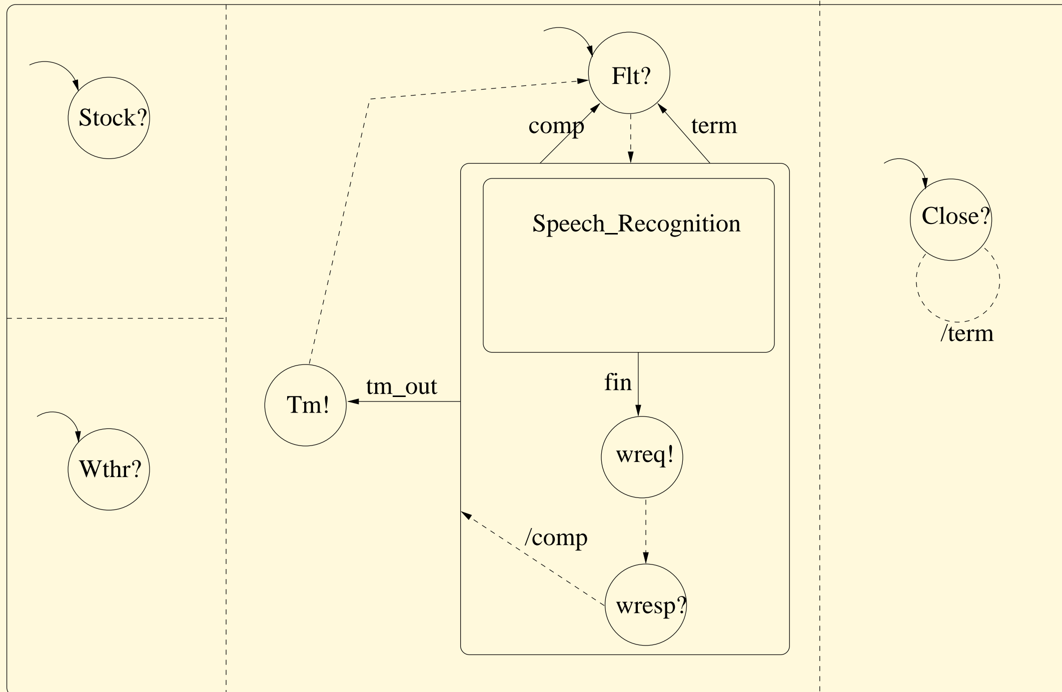
Go Back

Full Screen

Close

Quit

DSP application



Home Page

Title Page

Page 21 of 48

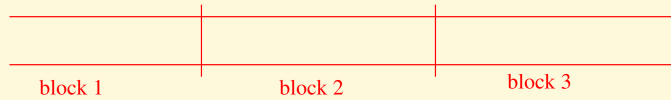
Go Back

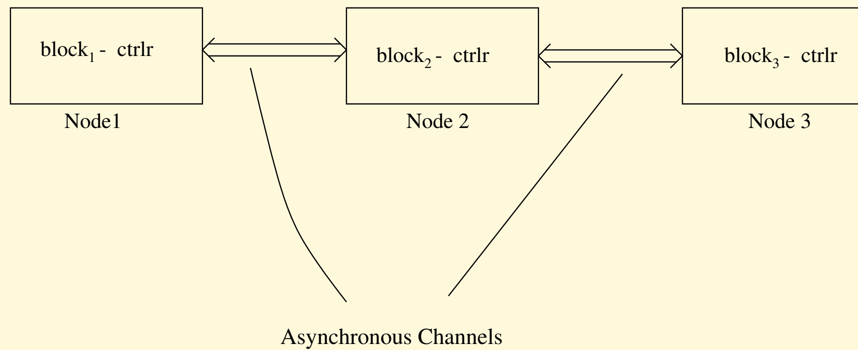
Full Screen

Close

Quit

EXAMPLE : Track Controller (Fischer et al ’92)





Home Page

Title Page

Page 22 of 48

Go Back

Full Screen

Close

Quit

Train

I

ok1

enter&ok2

ok

A

resv_next

resv_prev

bye

cn!

cp!

go_ahead

go_ahead

D

Interface

E

cn?,cp?

(cp,cn)

time_out

F

ok2

bye

enter &ok1

ok

G

Block Controller

General CRSM

- Various extensions to pure CRSM
- Valued signals
- Variables and assignments
- Entry and exit functions
- Buffered communications
- value-passing communications

Formal Semantics of CRSM

Node Semantics (similar to Esterel)

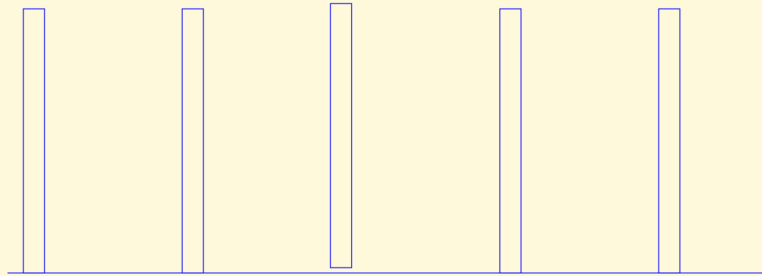
- An execution is a series of reaction instants
- In each reaction instant, a set of input signals is consumed and a set of output signals is generated.
- input signals include rendezvous requests
- set of infinite traces
- Each entry at the ticks of clock of the node

Network semantics (similar to CSP, CCS)

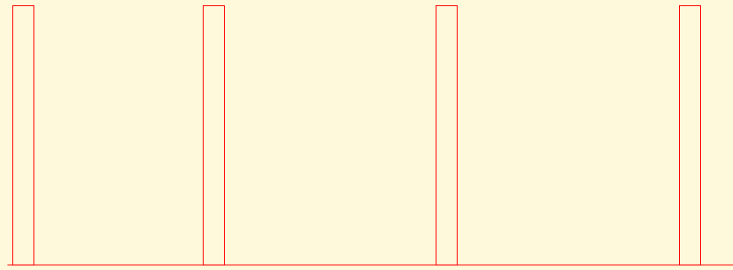
- interleaving of traces of nodes
- synchronization of rendezvous points
- clocks have common ticks at communication instants

Semantics

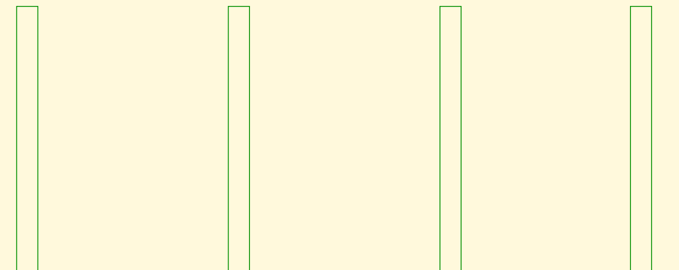
Node 1



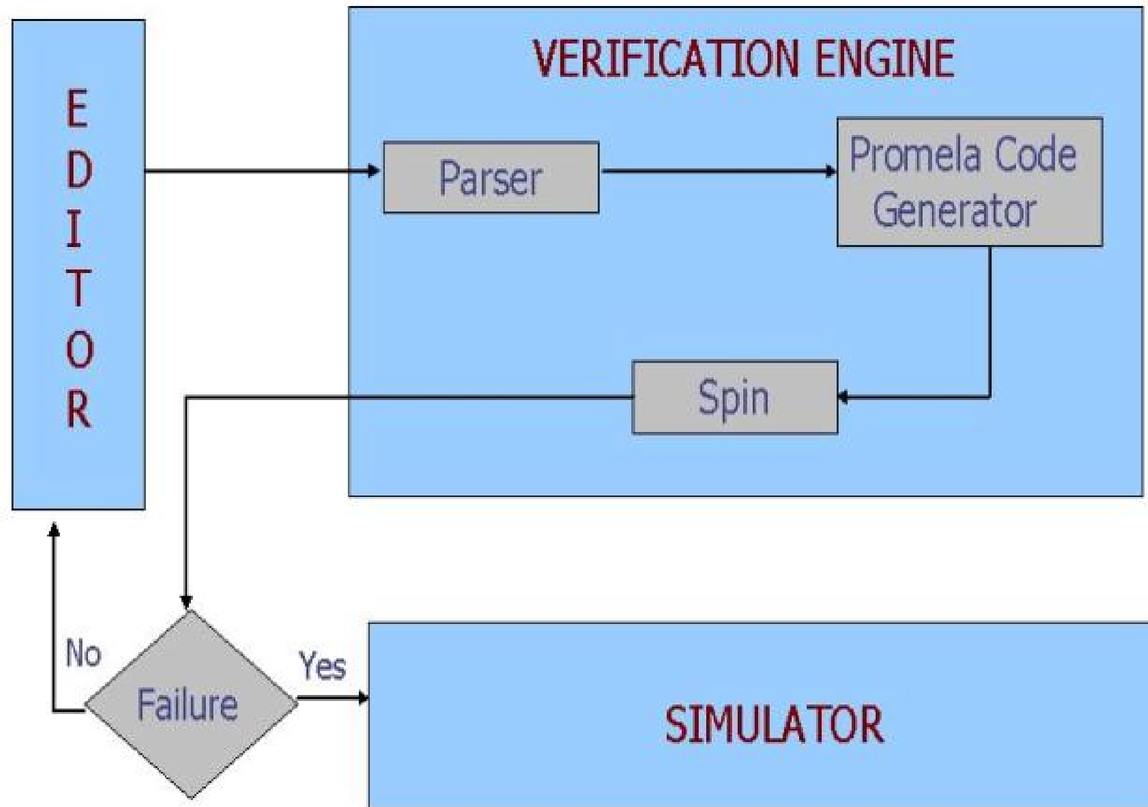
Node 2



Node 3



Tool Support for CRSM



Editor

CRSM Editor - ATM1.crsm

File Help

States Transitions Orthogonality

NState CState Init Hierarchy

Label Del Move Undo Clear Signals ZoomOut

/
Teller
Automata
A1
Idle
GetPin
PIN!
Authenticate
A1
Correct?
S2
S1
A2
Incorrect?
S3
S4
GetAmount
Amount!
AmtChk
A1
Ok?
S5
S6
A2
NotOk?
S7
S8
Bank
Automata
A1
PIN?
B1
Correct!
Incorrect!
Amount?
B2
Ok!
NotOk!

/ / Teller

cardValid/pin
Idle
GetPin
pinCode/
no/keepCard
PIN!
exit/ejectCard
Authenticate
yes/enterAmt
NO/ejectCard
GetAmount
amt/
Amount!
YES/delMoney,ejectCard
AmtChk

Home Page

Title Page

Page 28 of 48

Go Back

Full Screen

Close

Quit

Simulator

Display Window – ATM-ex

Files Options

- Teller
  - A1
    - Idle
    - GetPin
    - PIN
    - Authenticate
      - A1
        - Correct
        - S2
        - S1
      - A2
        - Incorrect
        - S3
        - S4
    - GetAmount
    - Amount
    - AmtChk
      - A1
        - Ok
        - S5
        - S6
      - A2
        - NotOk
        - S7
        - S8
- Bank
  - A1
    - PIN
    - B1
    - Correct
    - Incorrect
    - Amount
    - B2
    - Ok
    - NotOk

Idle, GetPin, PIN, Authenticate, GetAmount, Amount, AmtChk

cardValid/pin, pinCode/, no/keepCard, exit/ejectCard, yes/enterAmt, NO/ejectCard, amt/, YES/delMoney.ejectCard

Environment Window

Command Info Help

Pure Inputs	Pure Outputs	Local Signals
cardValid	getAmt	yes
pinCode	keepCard	no
amt	ejectCard	YES
exit	delMoney	NO
	pin	

Context Switch Tick Reset Recorder

Formal Verification

What?

- Rigorous checking of programs against specifications
- Specifications are properties
- Properties expressed in logic (CTL, LTL, etc.)

Why?

- Problems with traditional verification
- Need for rigorous Verification
- Safety-critical and high quality applications

How?

- Verification using Model-checking or theorem proving
- Many tools exist: SPIN, SMV, VIS, PVS, sTeP, etc.

Problems with Formal Verification

- Problem of specifying
- State explosion problem (model-checking)
- High human expertise (theorem proving)
- Huge extra effort
- Verification of the models rather than the real implementation
- Confusion over choice of methods and tools

Rest of the talk

- Our attempts to solve some of the problems in the context of CRSM

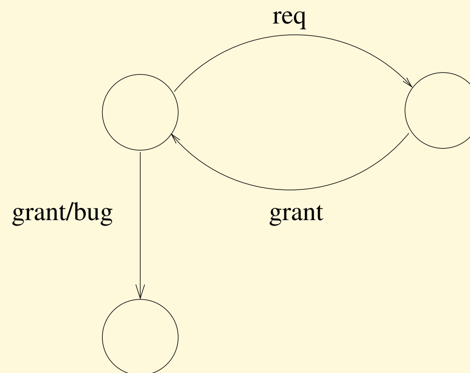
Formal Specification Problem

- One of our major concerns
- Independence from code/design
- Consistency
- Completeness
- Complex specification languages (LTL, CTL, CTL*, FOL, etc.)
- Two languages (Specification and modelling)
- different skills for mastering them
- lack of training/experience in specification
- quality of specification influences that of verification
- additional steps in development

Observer based verification

- an attempt to solve some of these problems
- Idea:
  - properties often can be viewed and modelled as observers
  - observers can *monitor* the system states and
  - complain when bad states are reached (properties violated)

Example: Absence of unsolicited response



Observers

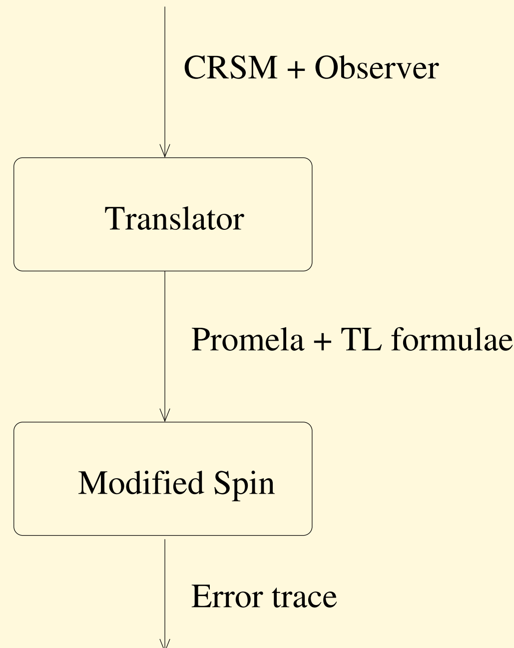
- Observers can be written as another program in the same language and developed hand-in-hand
- Observation can be modelled as synchronous parallel composition
- Observer and program run together synchronously and run-time checks made
- Or the combined state space of observer plus the program can be statically computed and analysed for reachability of bug states
- Synchronous parallel operator comes in handy here
- One language approach
- Verification limited to safety properties

Distributed Observers

- CRSM program consists of multiple synchronous nodes
- For verification, we employ one observer per node
- these observers monitor the local nodes
- also communicate with each other
- Program + observer is a CRSM program
- which can be analysed statically or at runtime

Verification Tool

- We have built a tool based on this idea
- Program + distributed observers translated to PROMELA
- Using SPIN, reachability of bad states is checked
- Necessary property for verification is automatically generated
- SPIN tool modified to generate a counter example which can run in our simulator



State Explosion Problem

- Size of the state space analysed is exponential on the number of concurrent components
- Various Solutions exists
  - Efficient Representations using Symbolic Techniques (BDDs)
  - Compositional Reductions: Reduce Components and Combine
  - Abstraction: Collapse states ignoring data values, or irrelevant details
  - Modular Verification: Verify components
- Many of the tools use one or more of these techniques
- We are exploring some of these reductions at the level of CRSM

Modular Verification

- The idea: Given a property, choose an appropriate subprogram and verify the subprogram
- The property chosen is such that it holds for the original program iff it holds for the subprogram identified.
- The subprogram has fewer concurrent components and hence state explosion problem is contained
- This result is somewhat old (Grumberg and Long '91)
- shown the result for an asynchronous model of concurrency
- We have extended this to our model

Our Result (Ramesh '01):

- We have defined a notion of refinement ref s.t.
  - $(A||B)$ ref A (and symmetrically B)
  - $(A_{(q||B)})$ ref A
  - If A ref B then
    - $(A||C)$ ref $(B||C)$ for any C.
    - $C_{(q||A)}$ ref $C_{(q||B)}$ for any C and q in C.
    - A^a ref B^a, under some assumptions
- Then we have the result that
  - If A ref B and B satisfies ϕ then A also satisfies ϕ provided ϕ is a negation-free LTL formula

Home Page

Title Page

Page 39 of 48

Go Back

Full Screen

Close

Quit

Efficient Verification of Programs

Two Strategies for Verification

1. Break the property into local properties and verify against components separately
   - Local verification is simpler and more efficient
   - Not automatic but general.
2. Identify appropriate subcomponent where the property holds.
   - Some kind of signal flow analysis
   - Automatic but not general

Abstracting irrelevant details

- Main problem here is in identifying which parts are irrelevant
- In observer based verification, this is somewhat easier
- Any state that does not lead to BUG states is irrelevant
- How to identify which states do not lead to BUG states
- Backward flow analysis
- Idea of program slicing
- Similar to Cone of Influence reduction

Program Slicing

- Well-known analysis technique in program analysis
- Ease of debugging and testing
- Formal verification would benefit from slicing
- Sequential program slicing (Weiser '84)
- Notion of slicing criterion: $< pc, Var >$
- Definition: $slice(P)$ w.r.t. $< pc, x >$ is P' where,
  - P' is obtained from P by removing some statements
  - If P reaches pc then P' also reaches pc and
  - x has the same value in both P, P' at pc.

Example: Slicing Criterion: $< write(sum), sum >$

```
Entry
read(n)
i:=1
sum:=0
pro:=1
while    i<=n
        sum:=sum+i
        pro:=pro*i
        i:=i+1
end while
write(sum)
write(pro)
Exit
```

```
Entry
read(n)
i:=1
sum:=0

while    i<=n
        sum:=sum+i

        i:=i+1
end while
write(sum)

Exit
```

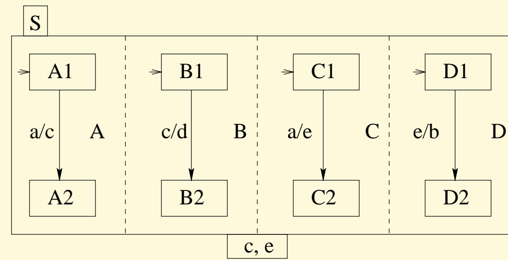
Slicing Reactive Programs

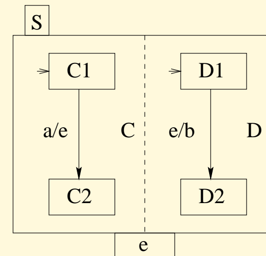
- Slicing criterion not very natural
- Reactive programs are event-oriented
- Non terminating ongoing behaviour
- Time or event ordering need to be preserved
- Proposal for a new definition suitable for reactive programs

Definition of Slice

- Slicing Criterion: just a signal or a set of signals
- Slice of P w.r.t to signal b has the same ongoing behaviour as P as far as b is concerned
- That is, b is present in a computation of P iff it is present in a computation of $Slice(P)$
- $Slice(P)$ obtained from P, by removing edges or states
- Slice of P w.r.t b preserves behaviour w.r.t. b in all computations of P

Example:





- Our Work (Vinod and Ramesh ’02):
  - We have defined the notion of slicing
  - We have developed the slicing algorithm for CRSM
  - Slicing preserves the structure of the program so that other reductions can be applied

Application to Verification

- In observer based verification, we are interested in computations that result in the emission of bug
- Slice w.r.t bug gives rise to a (hopefully) smaller state machine
- which is easier to analyse
- For general verification also this will be useful

Current Work

- Basic Verification engine is ready
- Implementation of modular verification and slicing-based verification in progress
- Some industrial case studies are being considered

Future Work

- Specification Language based upon Message Sequence Charts (MSCs)
- Testing based upon model-checking
- Case studies in robotics and SoC designs