Modeling & Analysis of Timed Systems

Wang Yi
Uppsala University
CUGS May 7-8, 2012

Main goal of this course

What’s inside the tools: UPPAAL & TIMES
(and also some recent work on multicore timing analysis if time allows)
UPPAAL: www.uppaal.com

- Developed jointly by
  - Uppsala university, Sweden
  - Aalorg university, Denmark

- UPPsala + AALborg = UPPAAL
  - SWEDEN + DENMARK = SWEDEN
  - SWEDEN + DENMARK = DENMARK
TIMES: www.timestool.com

- A branch of UPPAAL, developed at Uppsala
- TIMES = a Tool for Modeling and Implementation of Embedded Systems

TIMES a tool for resource scheduling and code synthesis

- System Model (Design)
- Question (specification)
- Yes (Debugging Information)
- No! (Debugging Information)
- Schedulability Analysis
- Executable code
- Rapid prototyping
OUTLINE

- A Brief Introduction
  - Motivation ... what are the problems to solve
  - CTL, LTL and basic model-checking algorithms

- Timed Systems
  - Timed automata, TCTL and verification problems
  - UPPAAL tutorial: data structures & algorithms
  - TIMES: schedulability analysis using timed automata

- Recent Work
  - The multicore timing analysis problems
  - Some solutions: WCET analysis and multiprocessor scheduling

Main references (papers)

- Temporal Logics (CTL, LTL)

- Timed Systems (Timed Automata, TCTL)
  - **On-line help of UPPAAL:** www.uppaal.com
Main references (books)

- Edmund M. Clarke, Orna Grumberg and Doron A. Peled, *Model Checking*

Lecture 1
Motivation and some historical remarks
The dream started 40 years ago in 1960’s aiming at “bug-free software”

What does this program do?
[Floyd 1967, Hoare 1969]
It computes the Greatest Common Divisor (gcd) of $x_1$ and $x_2$ [Floyd 67]

Specification (*partial correctness*)

Hoare logic: \{P\} program \{Q\} [Floyd 1967, Hoare 1969]

- Assume, initially (pre-condition)
  - $x_1 > 0$, $x_2 > 0$
- After each iteration of the loop (invariant)
  - $y_1 > 0$, $y_2 > 0$, $\text{gcd}(x_1, x_2) = \text{gcd}(y_1, y_2)$
- When done (post-condition)
  - $y_1 = \text{gcd}(x_1, x_2)$
What does this program do?

```
start
y1, y2 := x1, x2

if y1 == y2
  print(y1)
  stop
else if y1 > y2
  y1 := y1 - y2
  y2 := y2 - y1
else
  y2 := y2 - y1

if x1 > 0 and x2 > 0
  y1 > 0, y2 > 0, gcd(y1, y2) = gcd(x1, x2)
```

Can you check this?

Yes, you may prove it manually by induction on the number of iterations. Question: can you automate the proof?

Software verification (now, a hot topic)
One more example (*Total correctness*)

```
Function foo(n)
begin
  if n==1 then 1 
    else if even(n) then foo(n/2) 
      else foo(3*n+1)
  end
end
```

**Does this program terminate for any n? (WCET?)**

**Reality: 10 years later (1980’s)**

- The majority of programs are never proven correct! what went wrong?
  - Difficult to find and prove invariants: partial correctness
  - Difficult/impossible to prove termination: total correctness
  - Difficult to write complete specifications: what I really want?
- What to do?
  - Start another research program! In 20 years, the problems will be solved, hopefully
History: Model-checking invented in 70’s/80s

[PNueli 77, Clarke et al 83, POPL83, Sifakis et al 82]

- Restrict attention to finite-state programs
  - Control skeleton + boolean (finite-domain) variables
  - Found in hardware design, communication protocols, process control
- Temporal logic specification of e.g., synchronization pattern
  - There are algorithms to check that MODEL of program satisfies: SPEC
  - e.g. Alternating Bit Protocol skeleton, around 140 states, 1984
- BDD-based symbolic technique [Bryant 86]
  - SMV 1990 Clarke, McMillan et al, state-space $10^{20}$
  - Now powerful tools used in processor design
- On-the-fly enumerative technique [Holzman 89]
  - SPIN, COSPAN, CAESAR, KRONOS, IF/BIP, UPPAAL etc
- SAT-based techniques [Clarke et al, McMillan, ...]

History: Model checking for real time systems, started in the 80s/90s

- Extension of model checking to consider time quantities
  - Models, specifications, and algorithms can be extended
- Timed automata, timed process algebras [Alur&Dill 1990]
- Tools
  - KRONOS, Hytech, 1993-1995, IF 2000’s
Model Checking

Model: $M$

Property: $\varphi$

Timed Automaton

Model Checker

Yes!

No!

Error trace

Problems that can be addressed by Model Checking

Checking correctness of
- Communication protocols
- Distributed Algorithms
- Controllers
- Hardware circuits
- Parallel and distributed software
- Embedded and real-time systems and software
  e.g., Absence of race conditions, proper synchronization, ....

Model checking is the appropriate technique when there are many different scenarios of interaction between components in a system
Why testing not good enough

- **Testing/simulation**: coverage problems, difficult to deal with non-determinism and concurrent computation
- **Formal verification/Model-Checking** (= exhaustive testing of software and hardware design) provides 100% coverage

Model-Checking may complement testing to find (design) Bugs as early as possible
Introducing, Detecting and Correcting errors

- 30-50% of development time/money for testing
- Errors detected: the late the more expensive

Motivation: Model Verification

Requirements

- High level design
- Detailed design
- Coding
- Testing
- Deployment

Build model of the design. Analyze it thoroughly

Testing concentrates more on low-level issues. And conformance to model
An ‘abstract’ version of a fielded bus protocol

Reachable? (bug?)

Model-Checking
in a Nutshell
EXAMPLE: Petersson’s algorithm

**turn, flag1, flag2: shared variable**

- Process 1
  - loop
  - flag1:=1; turn:=2
  - while (flag2 & turn=2) wait
  - **CS1**
  - flag1:=0
  - end loop

- Process 2
  - loop
  - flag2:=1; turn:=1
  - while (flag1 & turn=1) wait
  - **CS2**
  - flag2:=0
  - end loop

**Question:** can both run in CS simultaneously?

---

Example: Fischer’s Protocol

![Diagram of Fischer’s Protocol]

**Init**

- $V=1$
- $X<100$ $V_1=1$ $X:=0$
- $X>100$ $V=1$
- $Y<100$ $V_2=2$ $Y:=0$
- $Y>100$ $V=2$

**Critical Section**
Example: the Vikings Problem

*Real time scheduling*

---

**UNSAFE**

> At most 2 crossing at a time
> Need torch

**SAFE**

Mines

What is the fastest time for getting all vikings on the safe side?

---

**UPPAAL** *A model checker for real-time systems*

---

System Model (Design)

Questions (specification)

UPPAAL

No! (Debugging Information)

Yes (Debugging Information)
MODELING

How to construct Model?

Program as State Machine!
A Light Controller

**WANT:** if press is issued twice quickly then the light will get brighter; otherwise the light is turned off.

A Light Controller (with timer)

**Solution:** Add real-valued clock $x$
Modeling Real Time Systems

- Events
  - synchronization
  - interrupts

- Timing constraints
  - specifying event arrivals
  - e.g. Periodic and sporadic

- Data variables & C-subset
  - Guards
  - assignments

### Timed Automata

- $x > 10$
- $a$
- $x := 0$

- $x > 10 \land v == 100$
- $a$
- $x := 0 \land v++$
Construction of Models: Concurrency

Plant
Continuous

Controller Program
Discrete

Model of environment (user-supplied)

UPPAAL Model

Model of tasks (automatic)
SPECIFICATION

How to ask questions: Specs?

Specification = Requirement, Lamport 1977

- **Safety**
  - Something (bad) will not happen
- **Liveness**
  - Something (good) must happen
Specification=Requirement  [Lamport 1977]

- **Safety**
  - Something (bad) will not happen
- **Liveness**
  - Something (good) must happen

- **Realizability (for systems with limited resources)**
  - Schedulability, enough resources?

### Specification: Examples

- **Safety**
  - **AG** $\neg (P1.CS1 \& P2.CS2)$ *Always Globally*
  - **AG** $(m<100)$
  - **EF** $(5<6)$ *Possibly in Future*
    - construct the whole state space
    - Report deadlocks etc.
  - **EF** $(\text{viking1.safe} \& \text{viking2.safe} \& \text{viking3.safe} \& \text{viking4.safe})$
  - **AG** $(\text{time}>60 \text{ imply viking4.safe})$

- **Liveness**
  - **AF** $(m>100)$ *Eventually*
  - **AG** $(P1.\text{try imply AF P1.CS1})$ *Leads to*
(Formal) Verification

- Semantics of a system
  = all states + state transitions
    (all possible executions)

- Verification
  = state space exploration + examination
Verificatioin = Searching

State-Space of a system

(1) SAFETY:
-- Is it possible to fire the bombs?
-- Is it possible to go from A to B within 10 sec?

(2) LIVENESS:
-- Will B be executed eventually (no time bound given)?

Approaches to Verification

- Manual: Proof systems, paper and pen
  - Find invariants (difficult !)
  - Induction: Assume nth-state OK, check (n+1)th OK
  - Boring 😞 (more fun with programming)

- Semi-automatic: Theorem proving
  - Use theorem provers to prove the induction step
  - e.g. PVS, HOL, Coq
  - Require too much expertise 😡

- Automatic: Model-Checking 😊
  - State-Space Exploration and Examination
  - e.g. SPIN, SMV, UPPAAL
Two basic verification algorithms

- Reachability analysis
  - Checking safety properties

- Loop detection
  - Checking liveness properties

Modelling in UPPAAL: example

### Mutual Exclusion Program

Is it possible that P1 and P2 run C1 and C2 simultaneously?
Verification: example

(C1,C2) is not reachable!

UPPAAL Demo
Example: the Vikings Problem

Real time scheduling

At most 2 crossing at a time
Need torch

What is the fastest time for getting all vikings to the safe side?

This sounds too good!
What’s the problem?
Problem with verification: ‘State Explosion’

All combinations = exponential in no. of components

EXAMPLE

13 components and each with 1 clock & 10 states

# of states = 10,000,000,000,000 = 10,000 G
Each needs (10 * 10) * 4Bytes = 400 Bytes

Worst case memory usage >> 4,000,000GB
The dream goes on ... ...

- Model Checking, a useful and applicable technique as compiler theory