Lecture 12. Overview

- TTM semantics
- RTTL
- Description of StateTime and StateClock
- Example specifications; complete verification process

Readings:

1) J. Ostroff, "A Visual Toolset for the Design of Real-Time Discrete Event Systems", to appear in IEEE Transactions on Control Systems Technology, May 1997.

2) J. Ostroff, "Tools for the Systematic Design of Discrete Event Control Systems", submitted to Formal Methods in System Design, Kluwer, January 1997.

TTM Framework

TTMs are timed extensions of fair transition systems of Manna and Pnueli. A TTM $M = \langle V, I, T, J \rangle$ is:

V: finite set of typed system variables (not necessarily finite-domain). 2 distinguished variables: clock time t with type $= Z^{+}$. The event variable indicates what transition has just been taken. Its type is the transition set.

A *state* s is a mapping that assigns to each variable $v \in V$ a value in *type(v)*. A set of all states is denoted by Σ.

Example: Let $V = \{t, e, b, r\}$.
Then $s = \langle t : 10, e : tick, b : true, r : 6.8\rangle$ - denotes a system state after taking a clock tick at time $t = 9$.

TTM Framework (Cont'd)

I: the *initial condition* - satisfiable boolean expression that characterizes the states at which the execution of TTM can begin. State s s.t. $s \models I$ is an *initial state*.

T: a finite set of transitions which includes the distinguished transition *tick*. Each transition $\tau \in T$ is a function $\tau : \Sigma \to 2^{\Sigma}$.

J: a *justice* set where $J \subset T$. The justice constraint for each transition $\tau \in J$ disallows computations in which τ is continually enabled but not taken beyond a certain point. The *tick* transition is always in the justice set.

TTM Framework - Cont'd

A transition τ consists of:

- enabling condition (does not include time)
- simultaneous update function, denoting variables that change when the transition is taken
- lower bound l - transition should be enabled but not taken for at least l ticks
- upper bound u - transition must be taken by u ticks of the clock unless it is disabled, i.e., preempted by some other transition.

Example:

Some More Notation

- Use unprimed and primed system variables to indicate evaluations in pre- and poststates.
Example: Let $V = \{t, e, b, r, y\}$.
Then the tick transition relation is:
$\tau_{tick}(V, V')$: $(t' = t+1) \wedge (e' = tick) \wedge \text{same}(b, r, y)$
Non-tick transition relation is:
$\tau_{\alpha}(V, V')$: $(r' = r + y) \wedge (r \geq y) \wedge (e' = \alpha) \wedge \text{same}(t, y)$

Taxonomy of transitions:
- A *spontaneous transition* $\tau[0, \infty]$ may occur at any point after becoming enabled, or never.
- A *just transition* must eventually occur if it is continually enabled. Qualitative quantity since although a just transition must occur, no finite bound on time of occurrence is given.
- A *timed transition* like $\tau[3, 7]$ must occur within an interval specified by a lower and an upper time bound.

And Some More....

- *A computation* is a sequence of state changes and transitions, starting in an initial state.
- From any state, any enabled transition is taken *in one atomic step*.
- Concurrency is modeled by interleaving.
- The clock must tick infinitely often, and an arbitrary (but finite) number of non-tick transitions can be taken between any two ticks of the clock.

Three sources of non-determinism:
- A transition relation itself may be non-deterministic
- Several transitions can be enabled at the same time, but only one is taken.
- Timing intervals introduce non-determinism.

So, this model is able to specify asynchronous systems (no way to do this in SMV except via processes).

Example - Air Bag System

Computer checks all sensors in an automobile every 30 ms (one tick). When an impact is registered, start a chemical reaction that generates nitrogen gas to start filling the air bags. The air bag is fully inflated at 55 ms.

Figure 3, p. 10 of 1)

TTM Computations and Trajectories

A *computation* $\langle$*initial*, $s_0\rangle$, $\langle\tau_1, s_1\rangle, \langle\tau_2, s_2\rangle, \ldots$ is an infinite sequence of states satisfying the following:
1. Initialization - the first state satisfies the initial condition
2. Succession - states are related by transitions correctly
3. Justice - for each transition τ is in the justice set.

A *trajectory* is a computation further constrained by the lower and upper bounds:
4. Ticking - clock ticks infinitely often in the computation.
5. Lower and upper bounds are respected.

Once the transition τ becomes enabled at position i, it begins to "mature" but cannot be taken until its lower time bound number of ticks has been taken, and then becomes "ripe" for execution. "Ripe" transition must be taken or become disabled before the upper time bound number of ticks has expired.
Transitions "mature" together as time advances but execute separately in an interleaving manner.

Clock Variables in TTM

- If the only infinite-domain variable is t, can sometimes get a finite-state system by introducing *clock variables*.
- Separate variable for each timing constraint.
- Set to 0 when enabling condition becomes true.
- Incremented with every *tick* transition.
- Can recover timing information from computation traces.

Example:

Review of Linear-time Temporal Logic

Operators:
Standard boolean connectives and quantifiers of predicate logic plus

$\bigcirc$	-	next
$\Box$	-	henceforth
$\Diamond$	-	eventually
$\mathcal{W}$	-	waiting for
$\mathcal{U}$	-	until
$\bigcirc$	-	previous
$\Box$	-	has always been
$\Diamond$	-	once
$\mathcal{B}$	-	back
$\mathcal{T}$	-	has always been

Real-Time Temporal Logic (RTTL)

- Linear time temporal logic with addition of timing operations.
- All formulas are interpreted w.r.t. a computation and then quantified for all computations.

Examples:

1) $\Diamond((e = \textit{inflated}) \wedge (t = 8))$.

2) $(e = \textit{impact} \wedge t = 7) \rightarrow \Diamond(e = \textit{inflated} \wedge t = 7)$.

3) $\Box[(e = \textit{impact} \wedge t = 7) \rightarrow \Diamond(e = \textit{inflated}) \wedge t = 8)]$

Use *entails* operator $\Rightarrow$ for expressing response properties: $[p \Rightarrow \Diamond q] \equiv \Box[p \rightarrow \Diamond q]$

4) $\Box\Diamond(e = \textit{tick})$ - clock ticks infinitely often. This property holds on infinite computations.

5) $p\mathcal{U}q$: p is true in all positions up but not including q.

6) $p\mathcal{W}q$ (p waiting-for q) is defined as $(p\mathcal{U}q) \vee \Box p$.

RTTL - Cont'd

Unclocked properties - not involving clock variable t.
RTTL includes standard temporal logic and also allows *clocked* properties.
Sometimes need to introduce new temporal logic operators:

$\Diamond_{\leq 5} p = \Diamond_{[0,5]}$ - before the 6th tick of the clock p holds

$\Diamond_{[4,6]} p$ - p holds between 4th and 6th tick

$\Diamond_4 p = \Diamond_{[4,4]} p$

$\Box_{<5} p$ - p is true from the initial position until just before the 5th tick of the clock.

But we can express these using clock variable t:

$$\Diamond_{[4,6]} q : [\forall t_0 \mid ((t = t_0) \rightarrow \Diamond(q \wedge (t_0 + 4 \leq t < t_0 + 6)))]$$

Zeno Computations in TTMs

Not every TTM is guaranteed to satisfy the ticking and time bound requirements, since can have 0-time cycles, called *Zeno computation*. E.g., immediate transition $\tau[0,0]$ that is a self-loop.

- Can disallow self-looping immediate transitions or at least check the validity of $M \models \Box\Diamond(e = \tau)$
- If we do want Zeno behaviors, then check temporal requirement r for every non-Zeno path:
$M \models (\Box\Diamond(e = \tau) \rightarrow r)$

StateTime Toolset

1) to describe devices and systems diagrammatically
2) execute the description so as to validate that the description is a reasonable model of the actual system
3) check that description conforms to its requirements using model-checking and theorem proving.

Figure 4, p. 13 of 1)

StateTime Toolset (Cont'd)

- *Build* - visual tool for constructing TTMcharts (like StateCharts but with simpler semantics and time bounded transitions). Automatically translates TTMcharts into TTMs.

- *Verify* - first model-checker for a subset of real-time logic.
  - Explicitly enumerates the global state reachability graph of any finite state TTM
  - Written in Prolog - slow
  - Generates a number of states proportional to the product of upper time bounds of all the transitions!
  - Is able to check response properties such as $p \Rightarrow \Diamond_{[l,u]}$ leaving time bounds $[l,u]$ unspecified - finds values for which the property is satisfied or gives a counter-example.

StateTime Toolset (Cont'd)

- Translator from TTMcharts to fair transition systems of STeP (Stanford Theorem Prover). STep:
  - The only available tool for checking linear temporal logic with past operators
  - Has a good theorem prover.

- Any unclocked property can be checked automatically
- Clocked can also be checked using the distinguished tick transition as a counter:
$[p \Rightarrow \Diamond_{[1,1]} q] \equiv [p \Rightarrow (e \neq \textit{tick})\mathcal{U}(e = \textit{tick} \wedge (e \neq \textit{tick}\mathcal{U}q))]$
But model-checker is exponential in the size of formula, so this is not practical for large time bounds.

StateTime Toolset (Cont'd)

- Clocked properties can be proven using Verify or rewritten in terms of additional clock variables. Also can construct an *observer* which detects violations of the timing properties. This increases the size of reachability graph but decreases the size of the temporal logic formula, and model-checking algorithm is

$$O(\mid M \mid \times e^{|r|})$$

- $\mid M \mid$ can be of order $m \times u$, where m is the number of states in the untimed reachability graph (all lower bounds are zero and all upper time bounds are infinity), and u is the product of all the upper time bounds.
- However, this theoretical limit is usually not reached, especially when we deal with clock variables.

Delayed Reactor Trip Example

Problem description:

"if power exceeds power threshold PT and the pressure exceeds delayed set point DSP, then wait for 3 seconds. If after 3 seconds the power is still greater than PT, then open the relay for 2 seconds."

Hardware Implementation:

Figure 1, p. 5 of 1)

Informal requirements:

(R1) Relay must remain open until Timer2 times out which happens after 2 sec.

(R2) If power reduces to an acceptable level then the relay should be closed as soon as possible (thus allowing the reactor to operate once more).

(R3) The controller never deadlocks.

DRT Example - Cont'd

Need to implement this using microprocessors with cycle time of 100ms.
First try (faulty!):

Figure 2, p. 6 of 1)

Analysis and Design of DRT

sud = *plant* || *control*
plant (environment) - relay, pressure, power sensors
controller (computer system - see pseudocode)

Plan:
(a) Use the Build tool to describe and simulate the *plant* as a TTM object
(b) Write requirements in RTTL of how the *plant* should behave
(c) Reverse engineer the proposed pseudocode *controller* into a TTM object
(d) Use the analysis tools to prove that the chart of *sud* conforms to the requirements.
Might also want an observer *obs* to watch the system for deficient behavior.

TTMchart of *sud*

Figure 5, p. 17 of 1)

TTMchart of DRT *plant*

Figure 7, p. 18 of 1)

- Can simulate the plant. Will be trajectories where the relay is not opened when it should be. For this reason, need a controller (software system).
- Sometimes, need user input, when nondeterminism is possible.

DRT Requirements

(R1): $(\mathit{bothHi} \wedge \Diamond_{30} \mathit{powerHi} \Rightarrow \Diamond_{[30,32]} \Box_{<20} (R = \mathit{open})$
"whenever a critical high is sensed, and 30 clock ticks later the power is still high, then the relay is opened within 32 ticks from the critical state and remains open until the 20th tick".
Controllers cannot respond to instantaneous changes in the sensor values (need to wait for one clock tick); controller cannot respond to critical high when it is in the middle of counting its timeouts; also need output object *delay*.

$\mathit{bothHi} \equiv \mathit{init}(\mathit{control}) \wedge R = \mathit{closed} \wedge D = \mathit{delay} \wedge (P = 1 \wedge W = 1)$
$\mathit{powerHi} \equiv (D = \mathit{delay} \wedge W = 1)$
$\mathit{init}(\mathit{control}) \equiv (F_1 = \mathit{normal\_full} \wedge T_a = 0 \wedge T_b = 0)$

(R2): $\mathit{powerLo} \Rightarrow \Diamond_{\leq 2} (R = \mathit{closed})$, where
$\mathit{powerLo} \equiv (D = 0 \wedge W = 0) \wedge \mathit{init}(\mathit{control})$

(R3) sanity check for controller, that controller must always return to its initial state after both times have finished their count:

(R3): $\neg \mathit{init}(\mathit{control}) \Rightarrow \Diamond_{\leq 52} \mathit{init}(\mathit{control})$

DRT Controller

- Verified initial pseudocode, so that loop is executed every one tick of the clock.
- Verify found an error - the transition that opens the relay after counter T_a has timed out has an unnecessary dependence on the pressure.

Correct Controller Code:

Figure 11, p. 23 of 1)

DRT Controller (Cont'd)

- Easy to translate into TTMs - enabling condition - conjunction of relevant "If" and "Else" branches.

Example:

$$\mathit{enb}(\mu_2) \equiv (W = 1 \wedge T_a = 0 \wedge 1 \leq T_b \leq 19) \vee$$
$$(W = 0 \wedge T_a = 0 \wedge 1 \leq T_b \leq 19)$$
$$\equiv (T_a = 0 \wedge 1 \leq T_b \leq 19)$$

Controller's TTM:

Figure 12, p. 24 of 1)

Verification

- Checked (R2) and (R3) using Verify. It helped discovering correct bounds for (R3)
- For (R1) - need an observer, for which verify a property

$$\Box(V \neq \mathit{bad}),$$

using a theorem prover. (Idea - make the formula shorter!)

Observer:

Figure 14, p. 26 of 1)

Verification (Cont'd)

Q: How do we know we've verified all required properties?

A: We don't! An additional property that must be satisfied is "the relay should not open unnecessarily", formalized as

(R4): $(R = \mathit{closed} \wedge W = 0)$
$\Rightarrow (R = \mathit{closed})\mathcal{W}(W = 1 \wedge P = 1)$

This property was found to be valid.

Can submit other putative theorems ("challenges" to the system), like

$$\Box(e \neq \mathit{failure}) \rightarrow [\neg\mathrm{init(control)} \Rightarrow \Diamond\mathrm{init(control)}]$$

Evaluation of StateTime

Good:
- Allows for systematic analysis of control systems
- Has visual, StateCharts-like input language
- Has native timing model-checker which suggests upper and lower bounds
- Can simulate system, provides useful counter-examples
- Can be translated into input of STeP and model-checked or theorem-proved.

Bad:
- Model-checking is slow. Sometimes can use STeP but in other cases have to produce observers i.e., to verify property r on machine M, we produce observer $M\prime$ and verify property $r\prime$ on it!
- How do we know we did translation correctly?

Dealing with Module Decomposition

- Want to decelop a module m with requireem-nts specified by s.
- Decompose m into two modules m_1 and m_2, with specifications s_1 and s_2.
- Prove validity of $s_1 \wedge s_2 \rightarrow s$ using theorem-prover.
- Hand over pairs (m_i, s_i) to implementors.
- If modules are finite-state, can use model-checker to prove $m_i \models s_i$.
- Otherwise, may try model-checking on finite-state simplification, like 5 philosophers instead of n for a dining philosophers problem. Then use theorem-proving.

Reference 2) describes application of this process to modules of Therac-25.

StateClock

- Augments TTMs (and StateTime) with clocks that can be started and stopped to measure the number of *tick* transitions between the start and stop time of events of interest.
- Thus, can refer to formulas like *(red* $\wedge$ *c = 0)* $\Rightarrow \Diamond$*(green* $\wedge$ *c = 30)*
- Is able to deal with modules and module interfaces. Splits variables into `out own v`, `out share v`, `in v` and clocks.
- Proof obligations are thus derived from module decomposition rule:

$$\left.\begin{array}{r} m_1 \models s_1 \\ m_2 \models s_2 \\ s_1 \wedge s_2 \rightarrow s \end{array}\right\} m_1 \| m_2 \models s$$

Summary of the Course

Looked at specifying and verifying systems using
- Branching-time model-checking (SMV)
- Process algebras via preorder and equivalence (Concurrency Workbench)
- Theorem-proving (Larch)

Also looked at systems to check language containment (COSPAN) and executable specifications (PAISley and StateCharts).
Finally, looked at attempts to specify and verify real-time systems.

Still to come:
- Presentations on PVS and HOL.

Next semester:
- Linear-time model-checking (SPIN?)
- Temporal Logic of Actions (TLA)
- Closer look at process algebras
- More theorem-proving
- More real-time approaches

StateClock GUI - Describes the Magnets in the Therac25 Plant

Figure 7, p. 23 of 2)

food for slide eater

food for slide eater

food for slide eater

food for slide eater