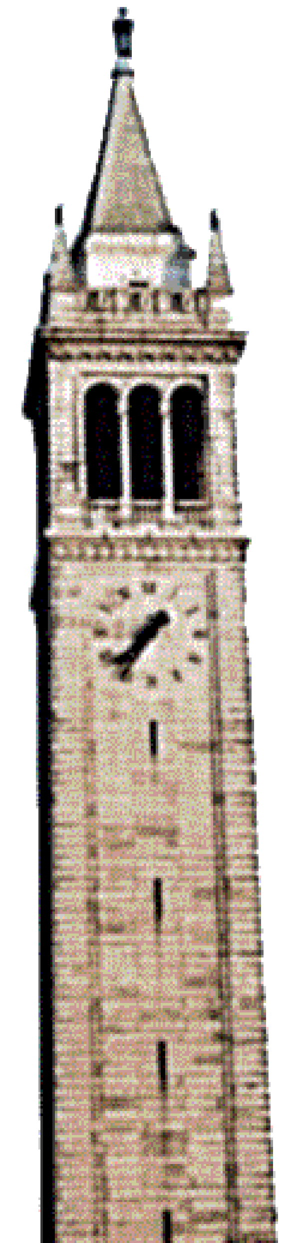
The STATEMATE Semantics of Statecharts

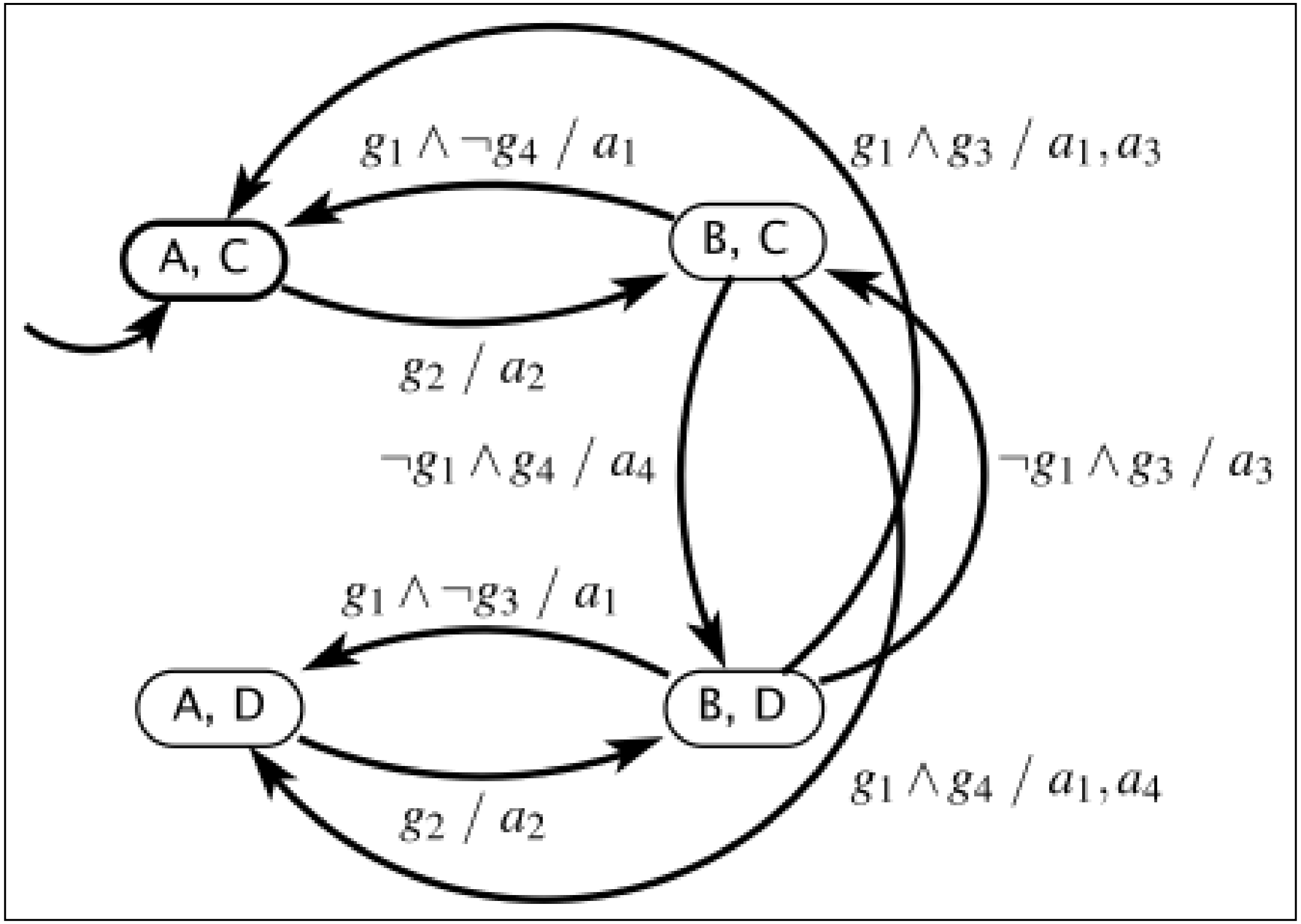
Surveyed and presented by
Hokeun Kim, Ben Zhang
EECS, University of California, Berkeley



Introduction

- Motivation
  - Finite State Machines
    - Advantages as a MoC
      - Good for designing reactive systems
      - Easy to use
      - Powerful algorithms for synthesis & verification
    - However, it is not suitable for designing systems with high complexity!

Introduction (cont'd)



FSM wo/ Any Extension

Introduction (cont'd)

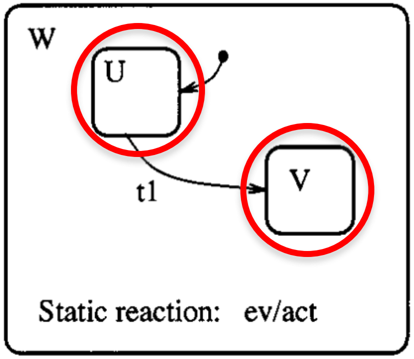
- Statechart
  - Design semantics for extended FSMs
  - Unofficial
  - Free to propose semantics
- STATEMATE
  - An implementation of Statechart
  - First executable semantics of Statechart

Introduction (cont'd)

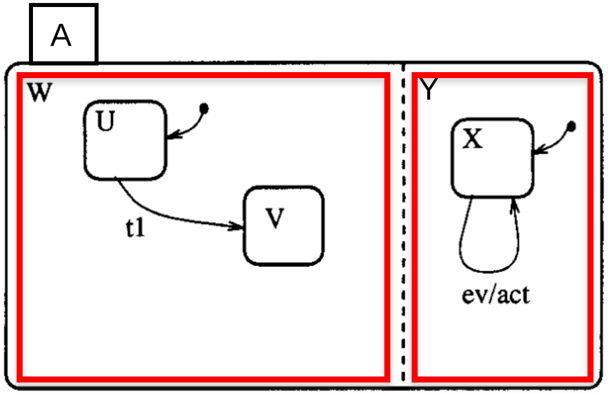
- About STATEMATE
  - Commercial tool
  - Designed for the specification and design of real-life complex systems coming from a variety of disciplines
- Main features of STATEMATE
  - Hierarchy of states
  - Orthogonality (concurrency)
  - History Connectors (memory of status)

The Basics

- States
  - OR-states



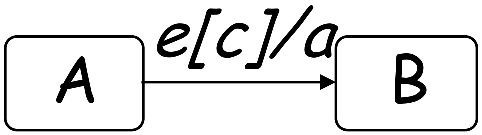
  - AND-states



  - Basic states
  - Root

The Basics (cont'd)

- Transitions



  - e: event – Triggers transition
  - c: condition – Guards transition
  - a: action – Carried out when a transition occurs
- Activities
  - Take a nonzero amount of time, like beeping, displaying, or executing lengthy computations
  - Durable whereas actions are instant
  - Defined as either throughout S or within S (e.g. activity A is active throughout/ within state S)

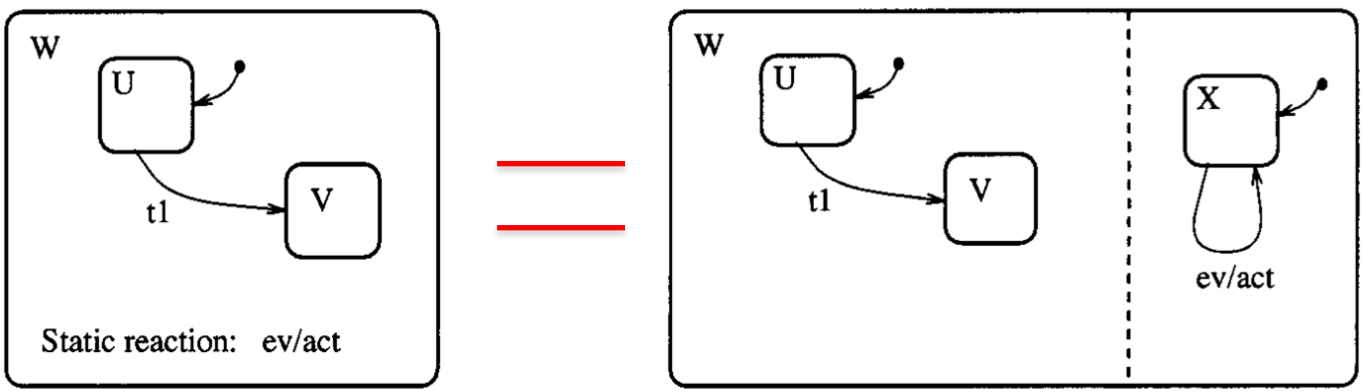
The Basics (cont'd)

- Special events, conditions, actions

	Events	Conditions	Actions
State S	entered(S) exited(S)	in(S)	
Activity A	started(A) stopped(A)	active(A) hanging(A)	start(A) stop(A) suspend(A) resume(A)
Data items D, F	read(D) written (D)	D=F, D<F, etc.	D:=exp
Condition C	true(C), false(C)		make_true(C) make_false(C)
Event e Time units d	timeout(e, d)		
Action a Time units d			schedule(a, d)

The Basics (cont'd)

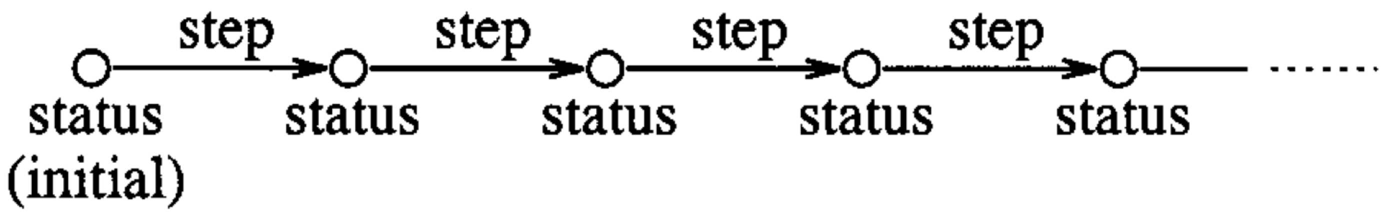
- Static Reactions (SRs)
  - In the same level as transitions
  - Has same format as transitions (e[c]/a)
  - Defined within a state
  - Can be taken whenever within state S where SR is defined



(a) (b)

The Basics (cont'd)

- Behavior of a system in STATEMATE
  - Run
    - A response of the system to a sequence of external stimuli
  - Status
    - Composes a run with a series of other status
  - Step
    - Execution between status
    - By executing a step, subsequent status is obtained

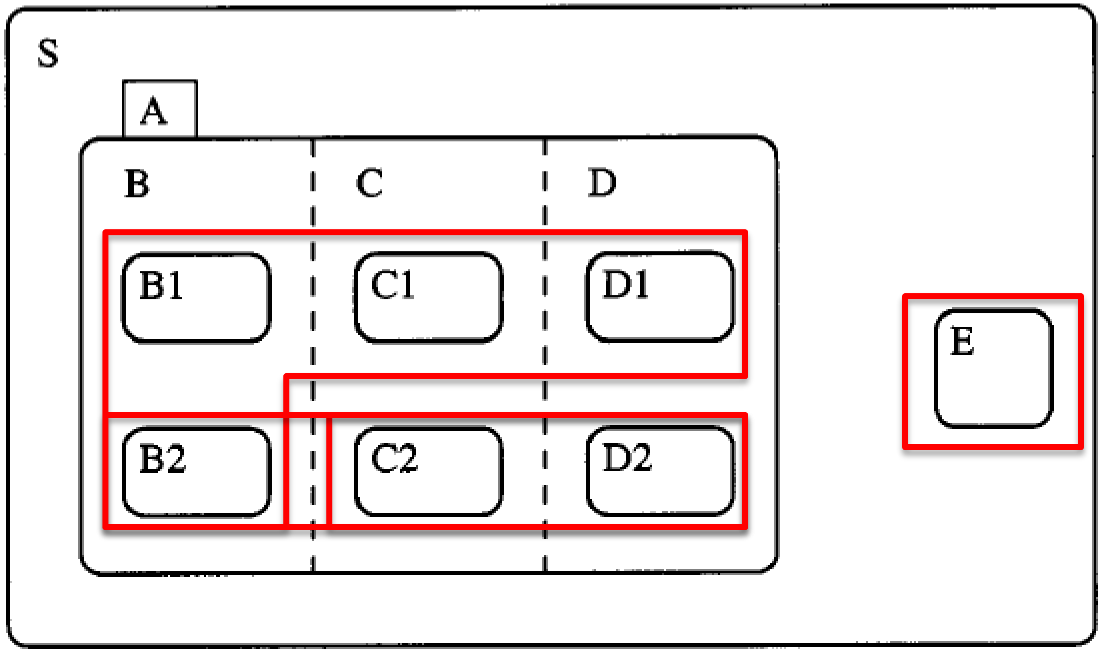


Basic System Reaction

- Configuration
  - A maximal subset of states that the system can be in simultaneously
- Deriving a configuration
  - Conditions
    - R: a root state
    - C: a configuration relative to R
  - Rules
    - $C \ni R$
    - If $C \ni$ OR state A, $C \ni$ exactly one of A's subtates
    - If $C \ni$ AND state A, $C \ni \forall a \in A$

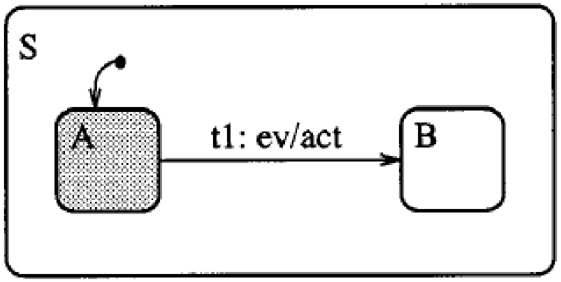
Basic System Reaction (cont'd)

- Basic configuration
  - A set of basic states in a legal configuration



Basic System Reaction (cont'd)

- An example of a response
  - When trigger ev occurs,



    - Transition t1 is enabled
    - exited(A), entered(B) are generated
    - in(A) becomes false, in(B) becomes true
    - Actions for exited(A) and entered(B) are executed
    - All SRs in S are enabled and executed if triggers are true
    - All activities specified for within or throughout A are deactivated and those for throughout B are activated

Compound Transition (cont'd)

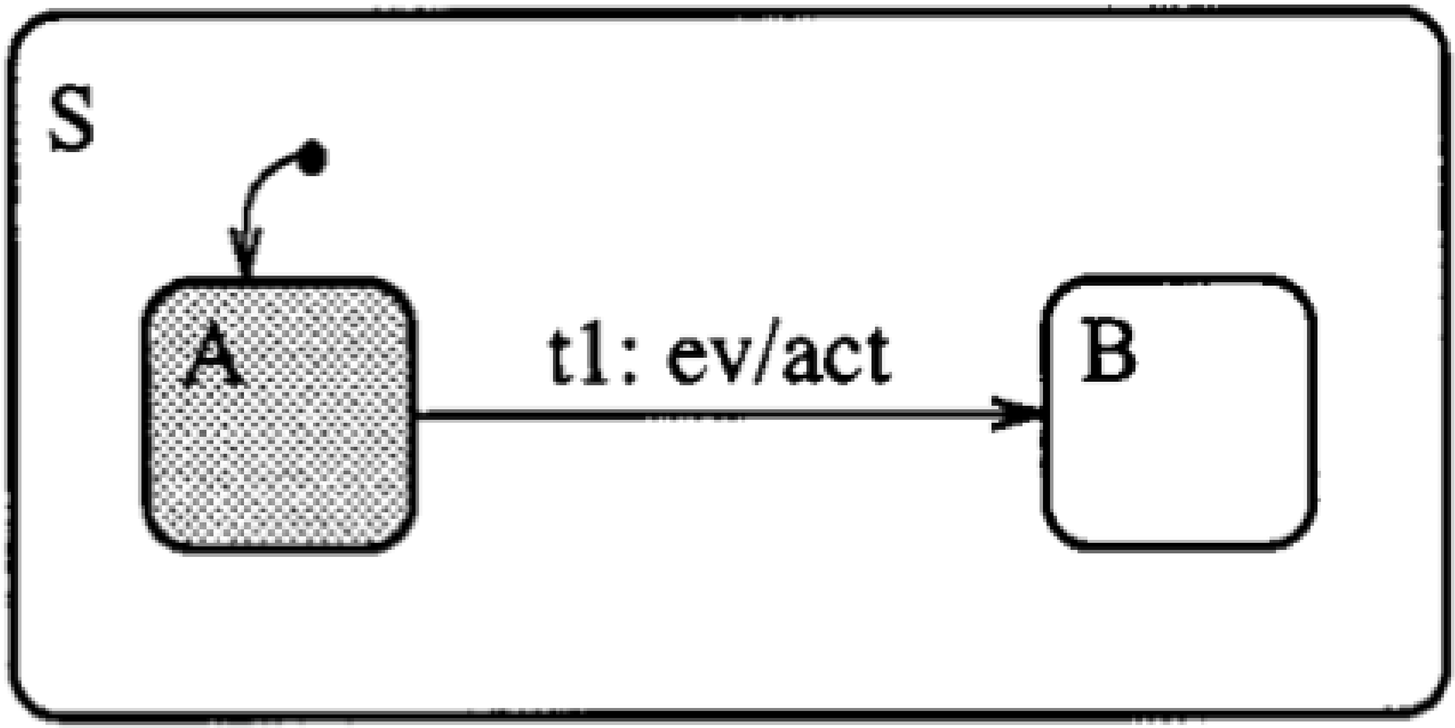


Figure 5.

Compound Transition

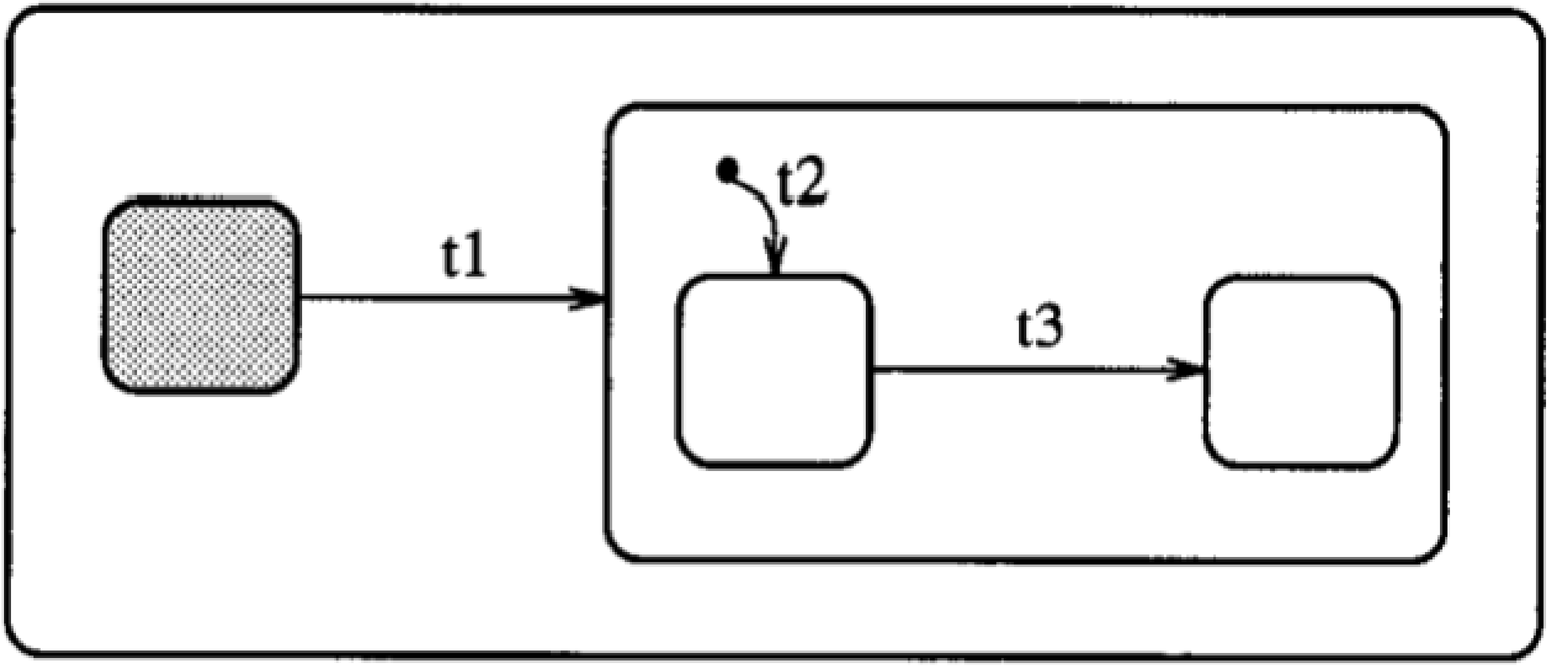


Figure 8.

Compound Transition (cont'd)

- **transition segments**
- **connectors:**
  - **joint/fork** (AND)
  - **condition/selection/junction** (OR)

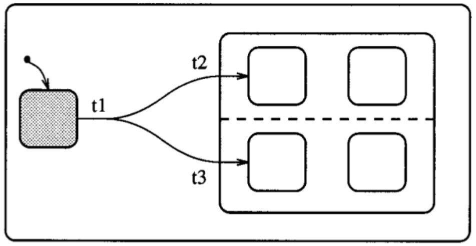


Figure 9.

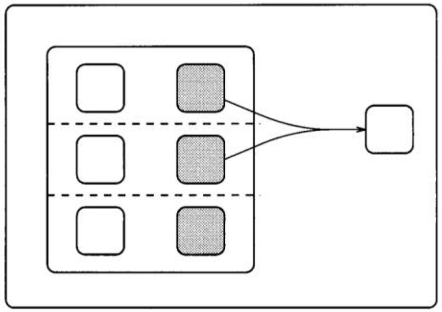


Figure 10.

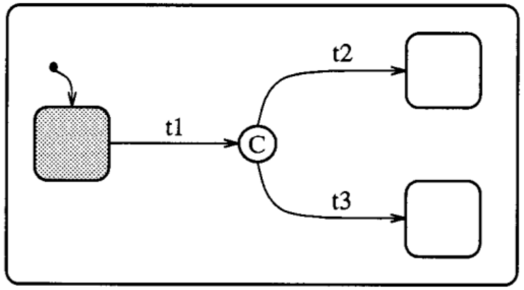


Figure 7.

Compound Transition (cont'd)

- **Two types of CTs**
  - **initial CT**
  - **continuation CT**

- A **full CT** is a combination of
  - one initial CT
  - possibly several continuation CTs
  - when executed, lead the system to a full basic configuration

Compound Transition (cont'd)

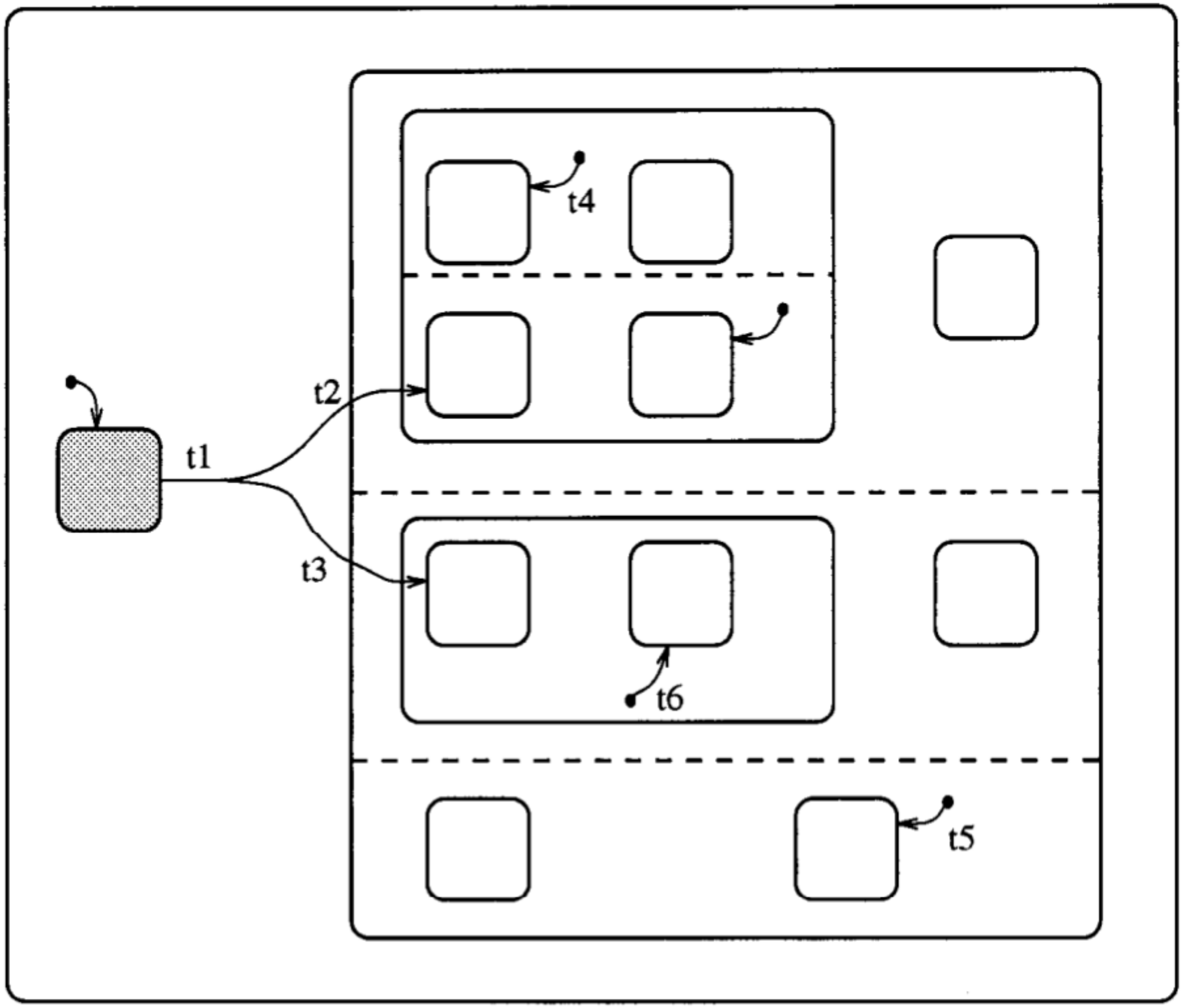


Figure 12.

Examples of CTs

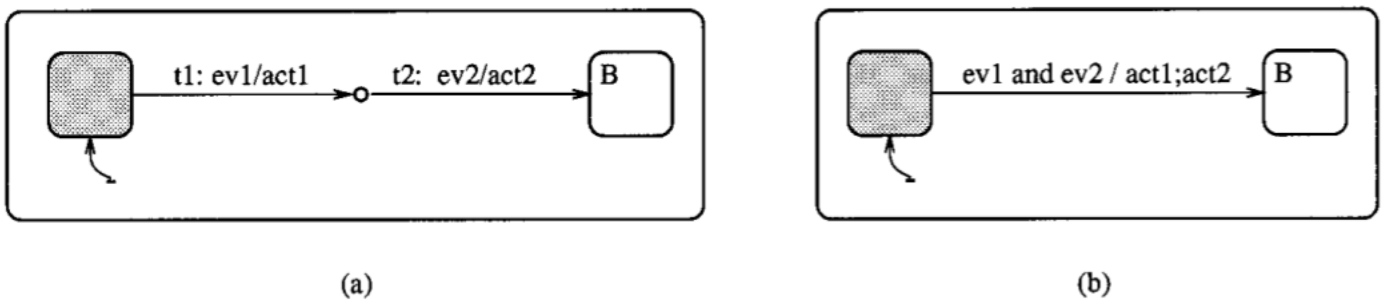


Figure 6.

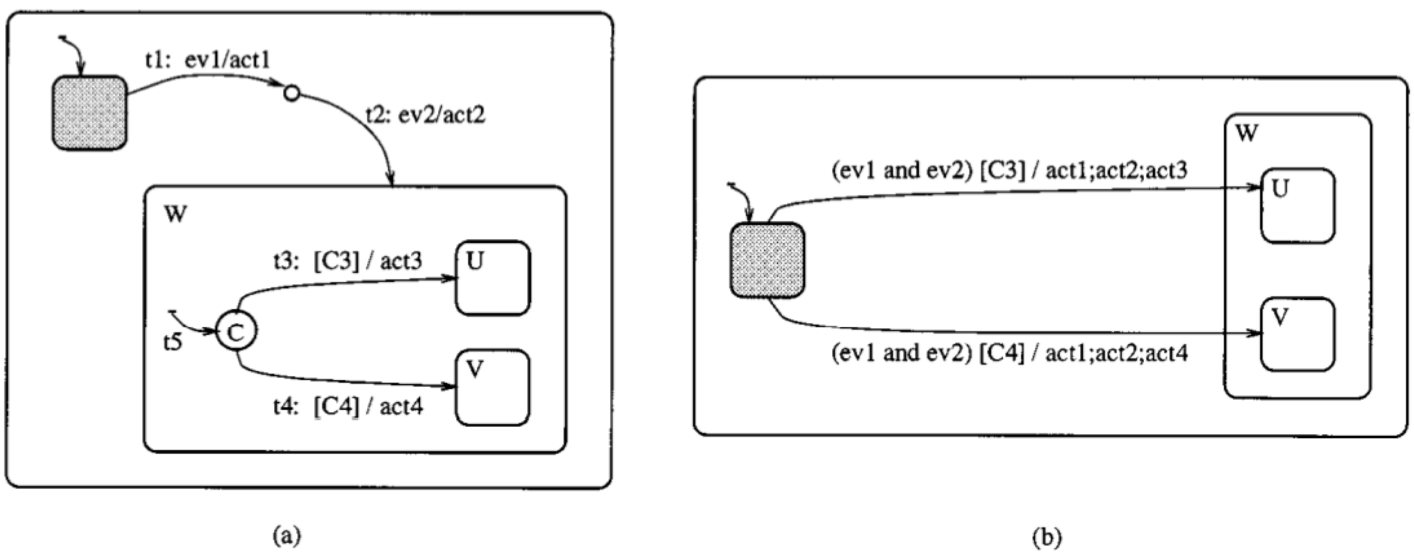


Figure 11.

Dealing With History

- S has history
  - H -> transition's target is substate of S
  - H* -> transition's target is basic configuration
- S doesn't have history (never-in or cleared)
  - transition's target is S

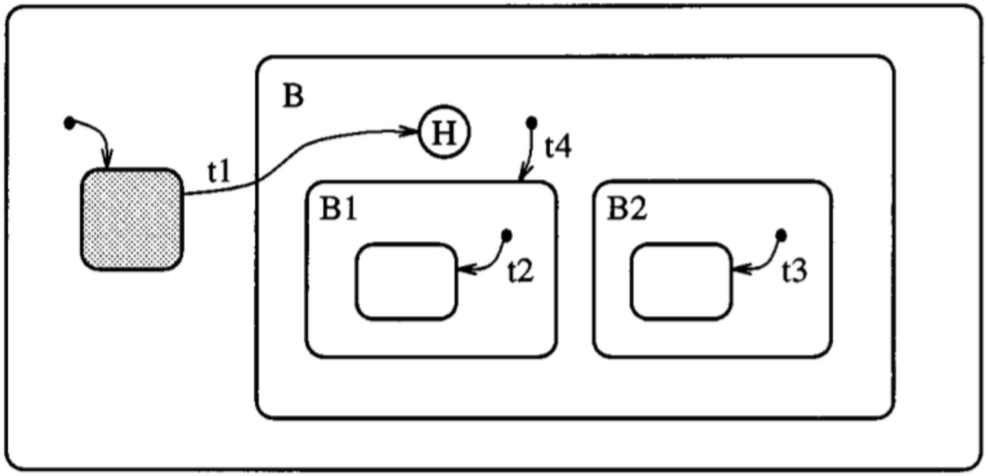


Figure 13.

Dealing With History (cont'd)

- history-clear(S)
  - only applies to S
- deep-clear(S)
  - applies to S and all its descendant states

- a new entry
  - new history information

Dealing With History (cont'd)

- Two subtle issues

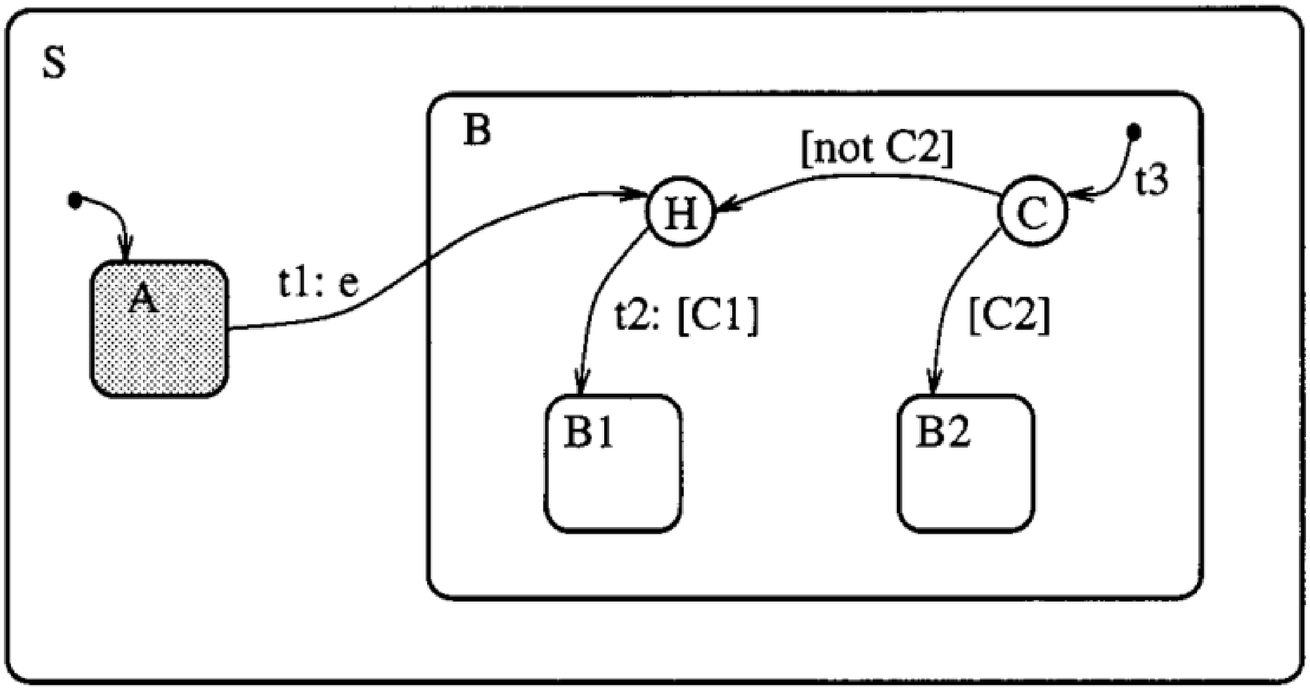


Figure 14.

Dealing With History (cont'd)

- Two subtle issues

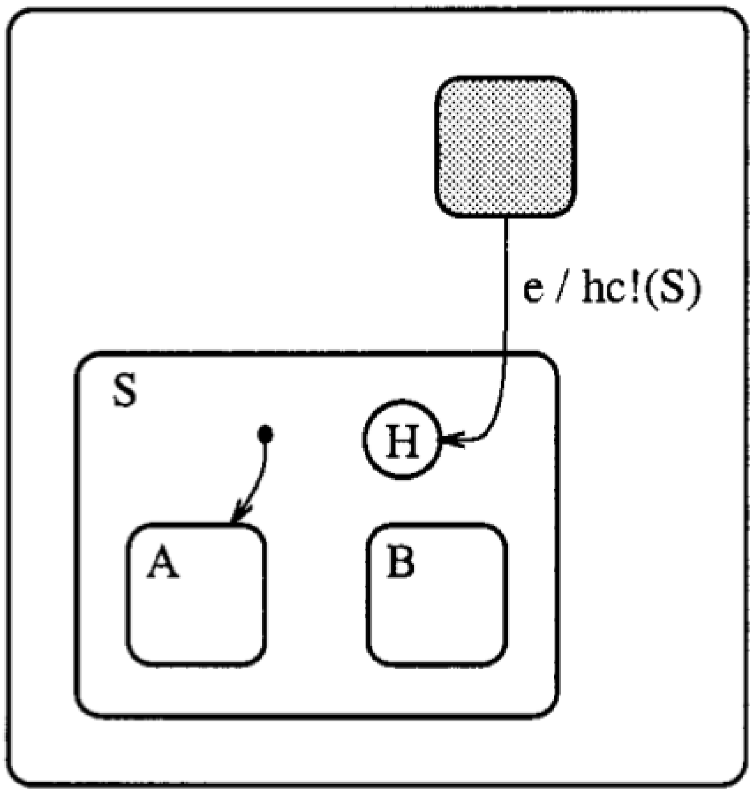


Figure 15.

Scope of Transition

- in State A, e and f are generated

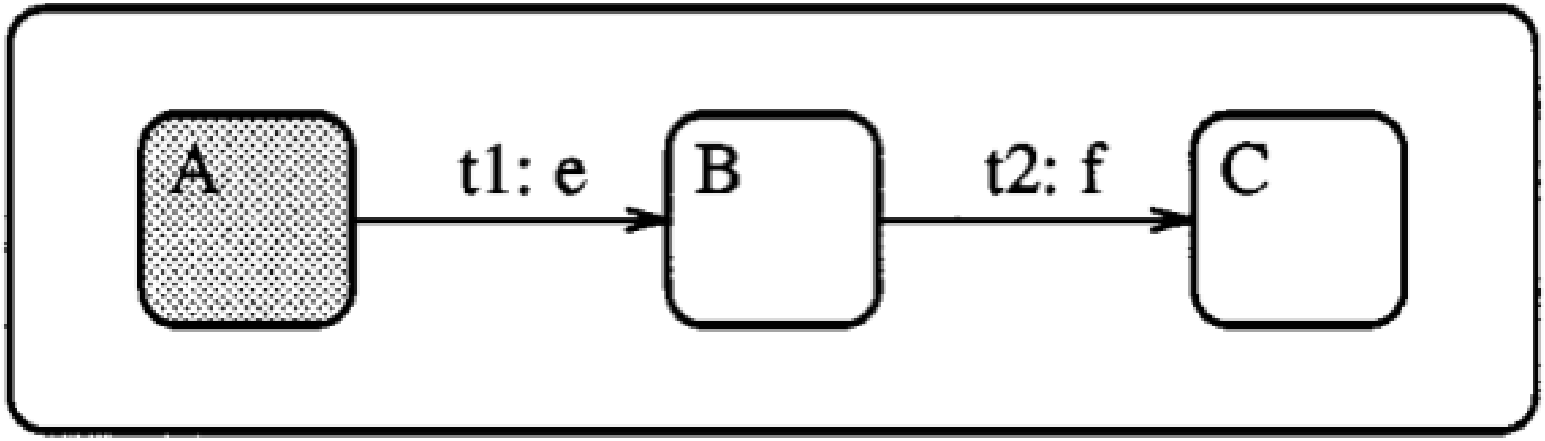


Figure 16.

Scope of Transition (cont'd)

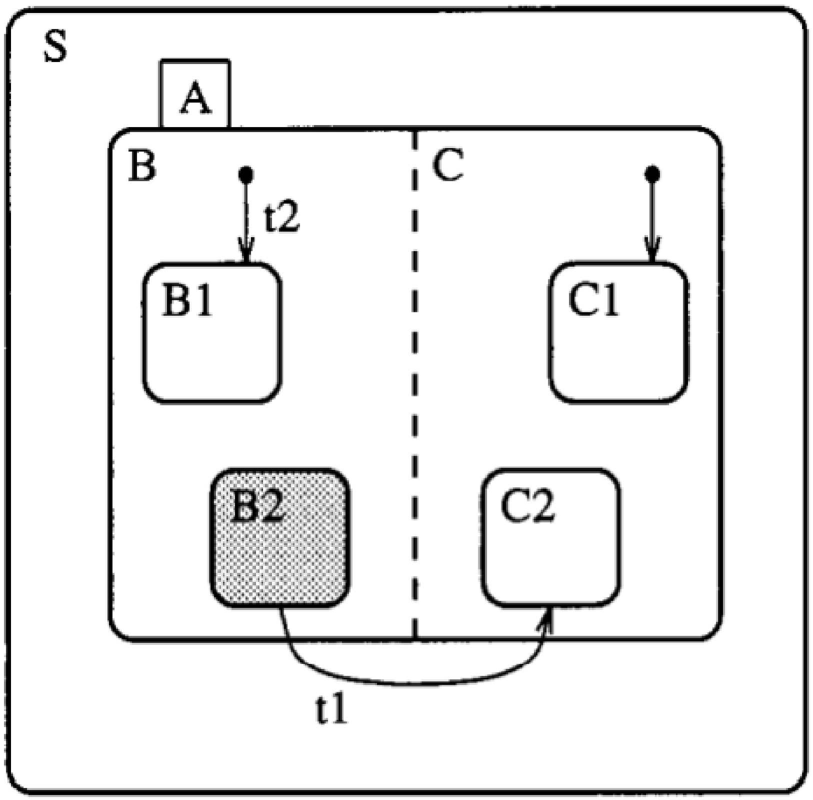


Figure 17.

- The **scope** of a CT
- the lowest OR-state in the hierarchy of states which is a proper common ancestor of all the <u>sources</u> and <u>targets</u>

Scope of Transition (cont'd)

- An interesting chart:

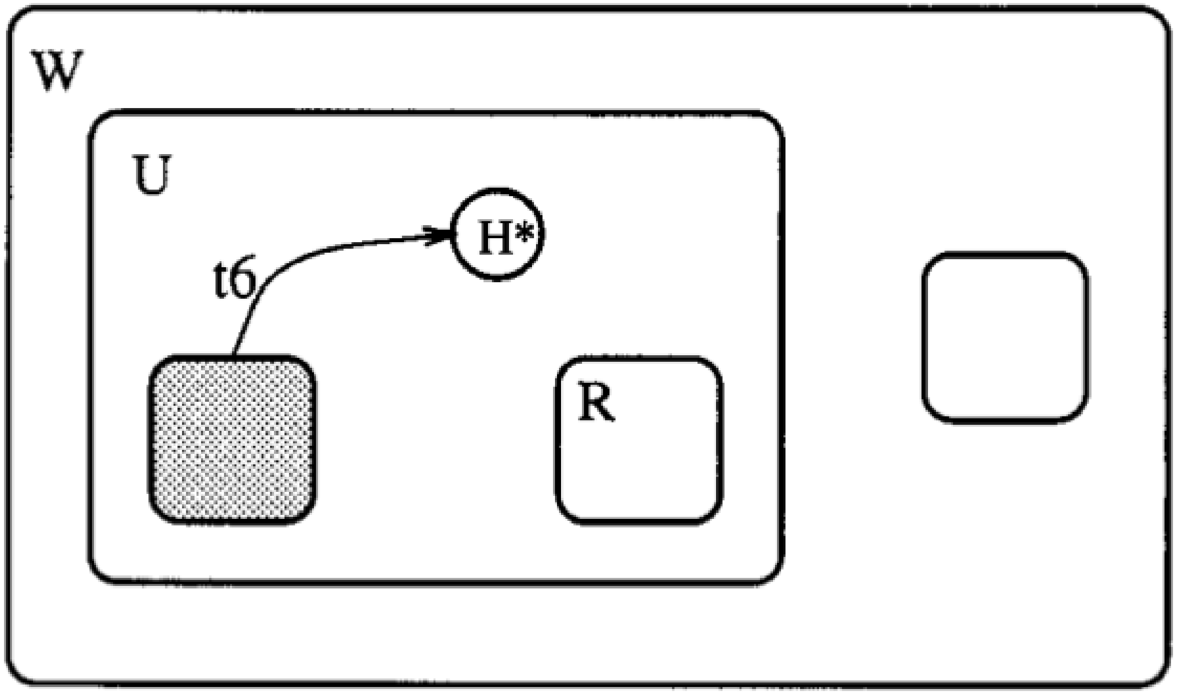


Figure 19.

Conflicting Transitions

- some common state that would be exited if any one of them were to be taken

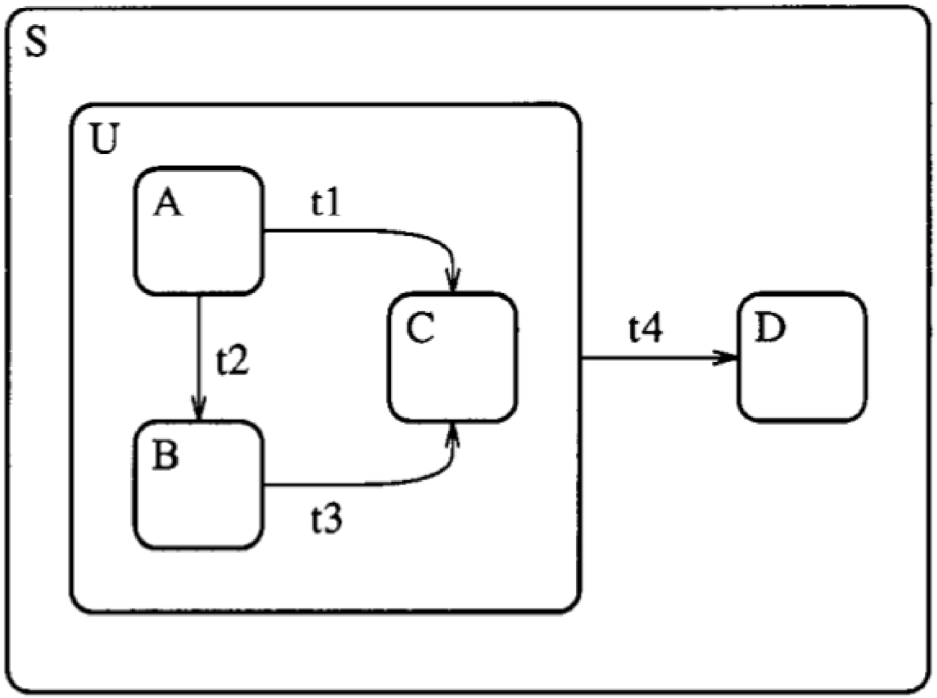


Figure 20.

Conflicting Transitions (cont'd)

- CT vs SR

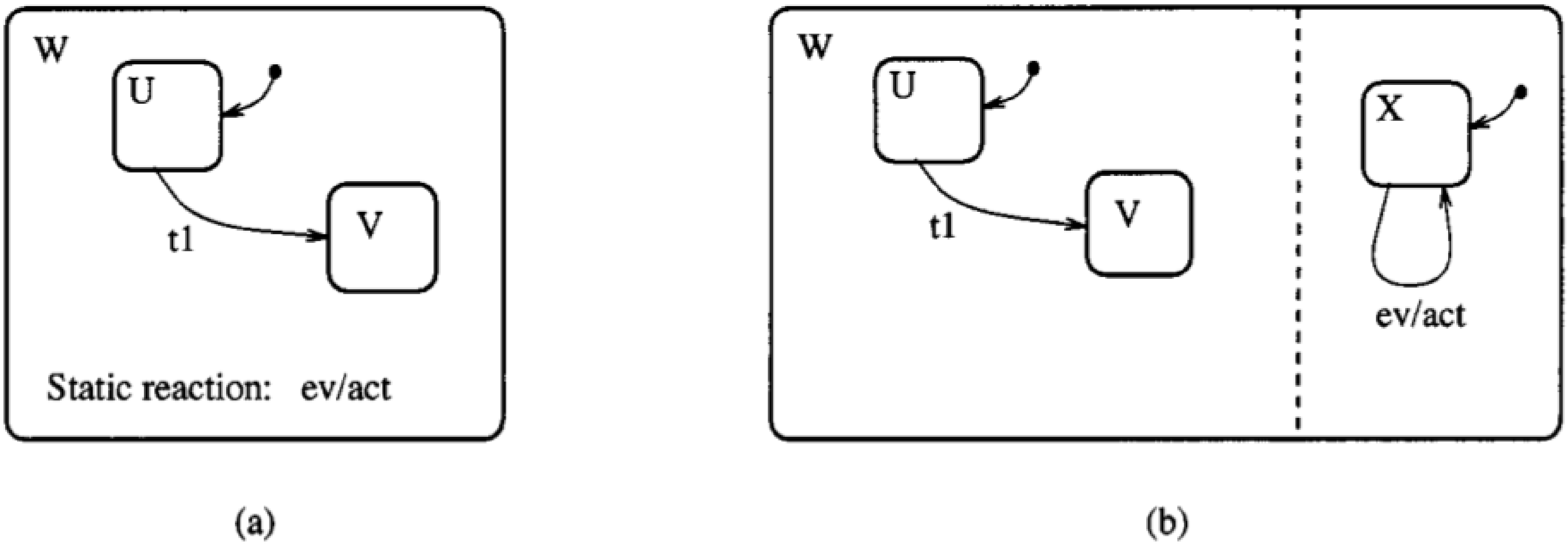


Figure 2.

The Basic Step Algorithm

- Inputs
  - Status of system
    - States, activities
    - Current values of conditions and data-items
    - Events generated internally
    - Scheduled actions and their time for execution
    - Timeout events and their time for occurence
  - Current time
  - External changes
    - Events, change in the values of conditions and data-items
- Output
  - A new system status

The Basic Step Algorithm (cont'd)

Stage 1

- Step Preparation
  - Execute actions caused by any changes in internal status and external events except for states

Stage 2

- Computes the Contents of the Step
  - Figure out enabled CTs and SRs by finding maximal nonconflicting sets

Stage 3

- Execute the CTs and SRs
  - Execute the SRs and CTs
  - Update history and also execute related actions with exiting end entering states

The Basic Step Algorithm (cont'd)

Stage 1

- Step Preparation
  - Execute actions caused by any changes in internal status and external events except for states

- Add external events to the internal event list
- Execute all actions due to changes except for state changes
- Carry out scheduled events
- Generate timeout events

The Basic Step Algorithm (cont'd)

Stage 2

- Computes the Contents of the Step
  - Figure out enabled CTs and SRs by finding maximal nonconflicting sets

- Compute the set of enabled CTs
- Remove CTs in conflict and have lower priority
- Split enabled CTs into maximal nonconflicting sets
- Repeat splitting until there is no more enabled CTs and SRs

The Basic Step Algorithm (cont'd)

Stage 3

- Execute the CTs and SRs
  - Execute the SRs and CTs
  - Update history and also execute related actions with exiting end entering states

- Let EN be a set of enabled CTs and SRs
- For each SR X in EN, execute action associated with X
- For each CT X in EN, let Sx, Sn be exited and entered state,
  - Update history related to Sx, and delete related states from system status
  - Execute actions related to exited(Sx), X, entered(Sn)
  - Add list of stated entered in Sn

Two Models of Time

- Synchronous time model
  - Assumes the system executes a singe step every time unit
- Asynchronous time model
  - Assumes the system reacts whenever an external change occurs
  - Superstep
    - Collection of steps taking zero time

Two Models of Time (cont'd)

- Go commands used for simulation
  - GO-REPEAT
    - Repeatedly executes one step until the system is in a stable state (no more transitions) wo/ advancing clock
  - GO-ADVANCE
    - Advances clock and execute all timeout events and scheduled actions and call GO-REPEAT
  - GO-STEP
    - Executes one step without advancing time
  - GO-NEXT
    - Advances time and execute one step
  - GO-EXTENDED
    - Combination of GO-NEXT and GO-REPEAT

Racing Conditions

- An element is modified more than once
- In greedy model, this may happen in different steps
- So consider superstep
- Also take into account of causality
- A race situation is one in which, had we executed the enabled transitions in a different order (yet a legal one according to the above criteria), we might have obtained different results in one or more of the data-items or condi- tions.

Racing Conditions

- Modified more than once?
- Asynchronous model: superstep
  - e / *f; X* = 5
  - f / *X* = 6.

- A race situation
  - had we executed the enabled transitions in a different order
  - we might have obtained different results in one or more of the data-items or conditions

Racing Conditions (cont'd)

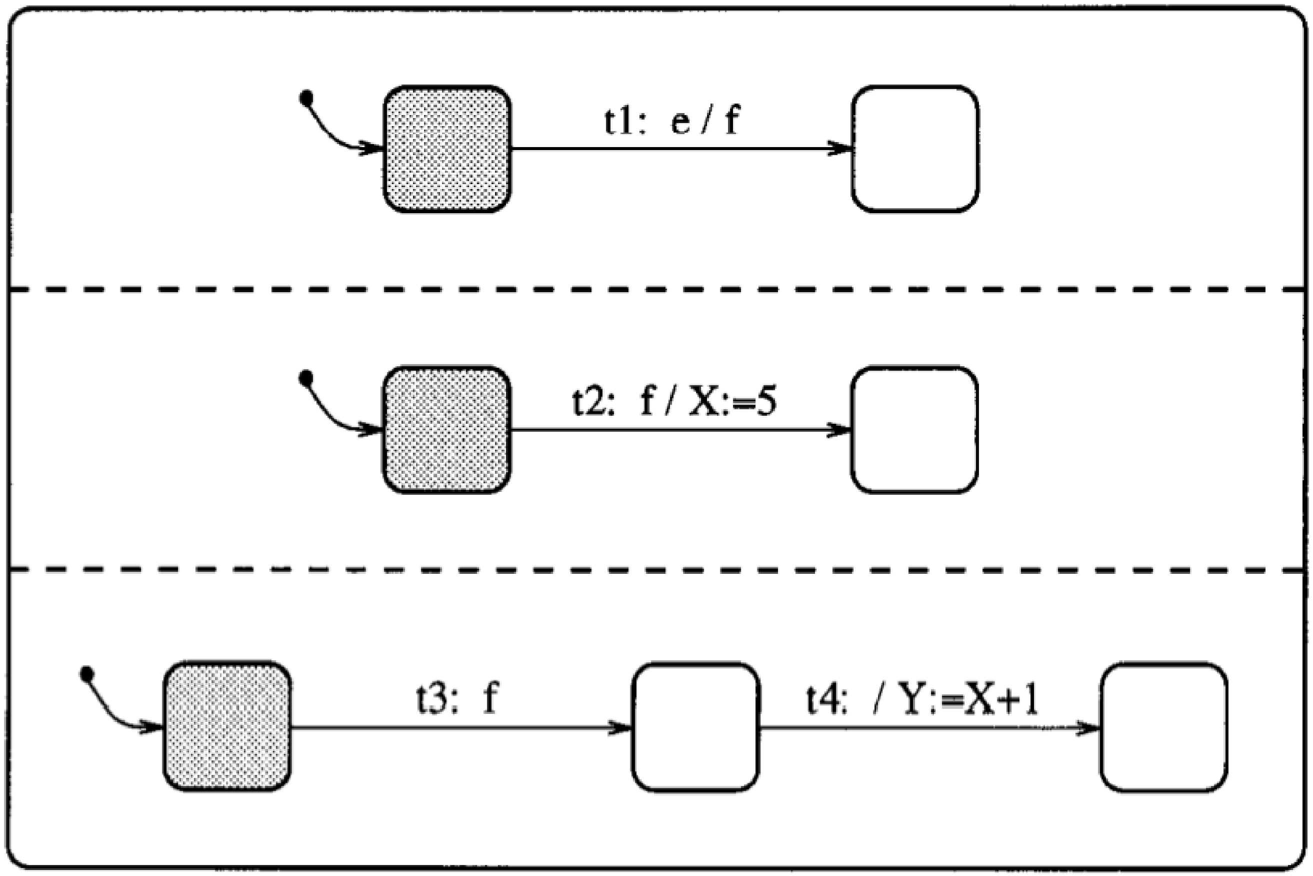


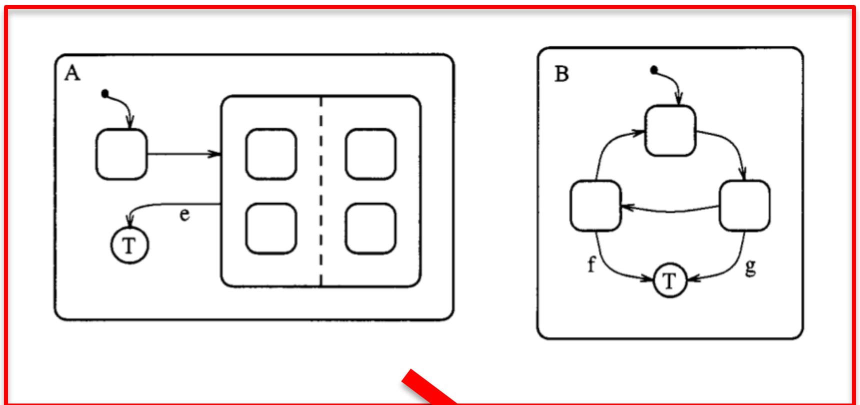
Figure 25.

Multiple Statecharts

- View as a single statechart (concurrent)
  - termination connector? -> special idle state
  - see example later

- Asynchronous vs. Synchronous
  - Asynchronous: all transition simultaneously
  - Synchronous: its own clock (affect timed event)

Multiple Statecharts (cont'd)



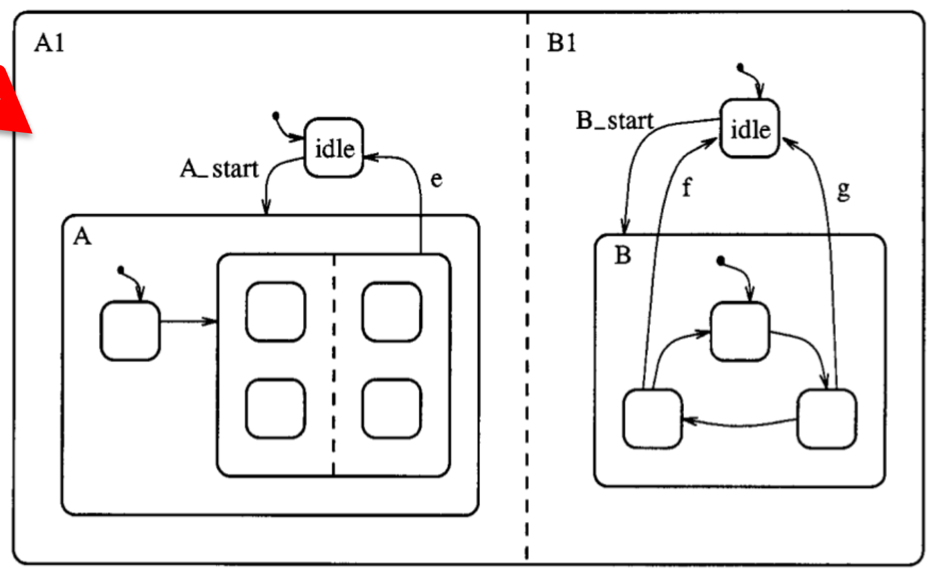


Figure 27.

Appendix: Comparison of Semantics

- One major issue: when changes take effect
  - in this step or next step?
- Comparison with RSML (mostly syntactical)
  - no support for history connector
  - no broadcast communication
  - no support for disjunctions of trigger events
  - steps <-> microstep, superstep <-> step
- More related works cont'd

Appendix: Comparison of Semantics

- Perfect-Synchrony Hypothesis
  - asynchronous and the synchronous time model
- Self-Triggering, Causality
  - events are sensed only in the following
- Negated Trigger Event
- Effect of Transition Executing is Contradictory to Its Cause
- Interlevel Transition
- State Reference
- Compositional Semantics, Self-Termination
- Operational versus Denotational Semantics
- Instantaneous State