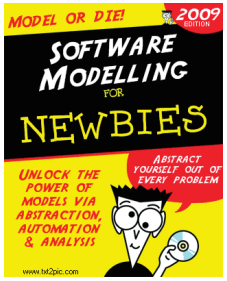
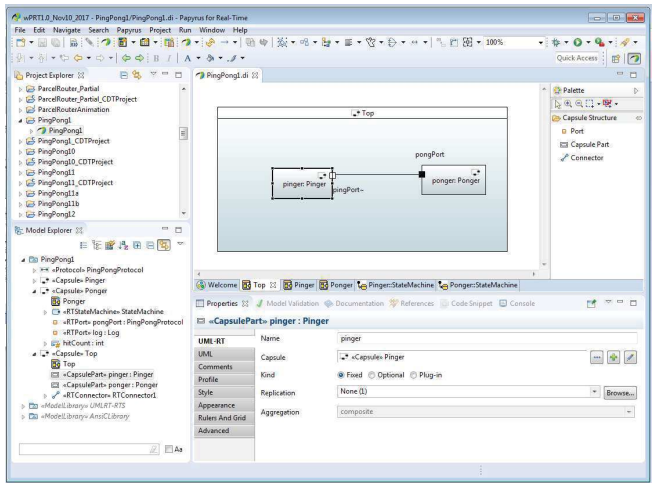
Beyond Code:
An Introduction to Model-Driven Software Development (CISC836)



UML-RT and IBM RSARTE: Part I

Juergen Dingel
January 2021

UML-RT and RSARTE: Sneak Peek



Modeling Languages

Modelica
- Physical systems
- Equation-based

Simulink
- Continuous control, DSP
- time-triggered dataflow

Stateflow
- Reactive systems
- Discrete control
- State-machine-based

Lustre/SCADE
- Embedded real-time
- Synchronous dataflow

UML-RT
- Embedded, real-time
- State-machine-based

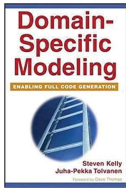
AADL
- Embedded, real-time

UML

UML MARTE
- Embedded, real-time

increasing generality ⟷ increasing domain-specifity

Examples in

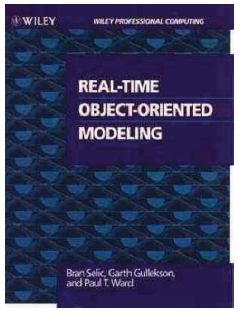


[Kelly, Tolvanen 2008]

[Voelter 2013]

UML-RT: History

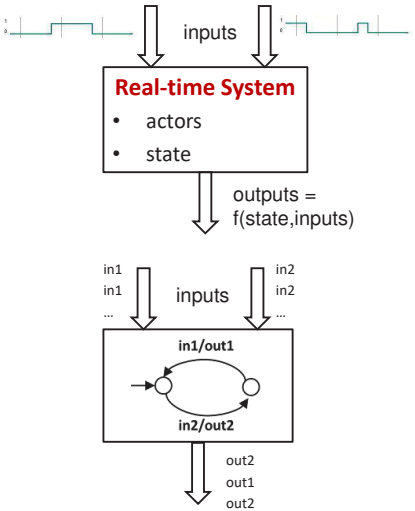
- Real-time OO Modeling (ROOM)
  - ObjecTime, early 1990 ties
- Major influence on UML 2
  - E.g., StructuredClassifier
- "RT subset of UML"
- Tools
  - ObjecTime Developer
  - IBM Rational RoseRT
  - IBM RSA-RTE
  - Eclipse Papyrus-RT
  - Protos eTrice



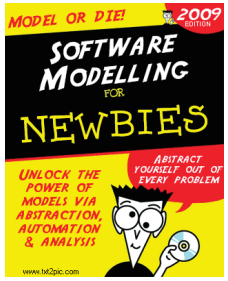
[Selic, Gullekson, Ward. *Real-Time Object-Oriented Modelling*. Wiley. 1994]

UML-RT: Characteristics

- Domain-specific
  - Embedded systems with soft real-time constraints
- Graphical, but textual syntax exists
- Small, cohesive set of concepts
- Strong encapsulation
  - Actors (active objects)
  - Explicit interfaces
  - Message-based communication
- Event-driven execution
  - State machines



Beyond Code: An Introduction to Model-Driven Software Development (CISC836)



UML-RT and RSARTE: Part II

Juergen Dingel
January 2021

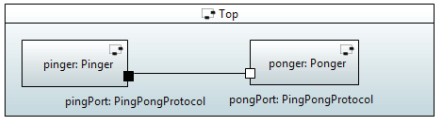
UML-RT/Papyrus-RT: Part II

- Core concepts
  - Structural modeling
  - Behavioural modeling

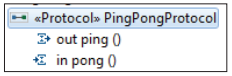
UML-RT: Core Concepts (1)

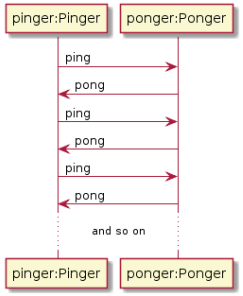
- Types
  - Capsules (active classes)
    - Capsule instances (parts)
  - Passive classes (data classes)
    - Objects
  - Protocols
  - Enumerations
- Structure
  - Attributes
  - Ports
  - Connectors
- Behaviour
  - Messages (events)
  - State machines
- Grouping
  - Package
- Relationship
  - Generalization
  - Associations

UML-RT: Core Concepts (2)



- Model
  - Collection of capsule definitions
  - 'Top' capsule containing collection of capsule instances (parts)
- Capsules
  - May contain
    - Attributes, ports, or other capsule instances (parts)
  - Behaviour defined by state machine
- Ports
  - Typed over protocol defining input and output messages
- State machine
  - Transition triggered by incoming messages
  - Action code can contain send statements that send messages over certain ports

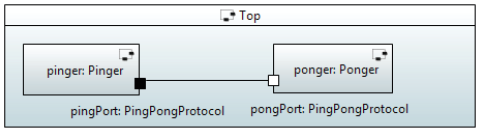


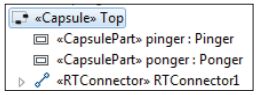


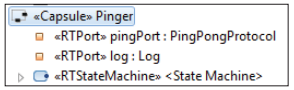
Capsules (1)

- Kind of active class
  - Attributes, operations
  - Own, independent flow of control (logical thread)
- May also contain
  - Ports over which messages can be sent and received
  - Parts (instances of other capsules) and connectors
- Creation, use of instances tightly controlled
  - Created by runtime system (RTS)
  - Cannot be passed around
  - Stored in attribute of another capsule (part)
  - Information flow only via messages sent to ports

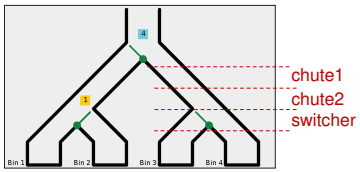
  ⇒ better concurrency control and encapsulation
- Behaviour defined by state machine

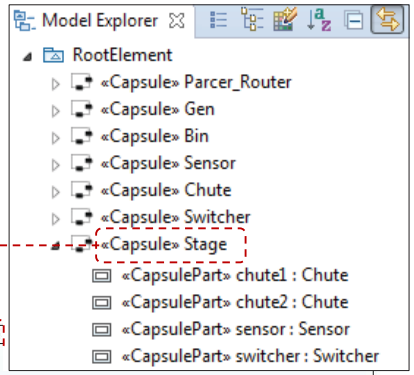


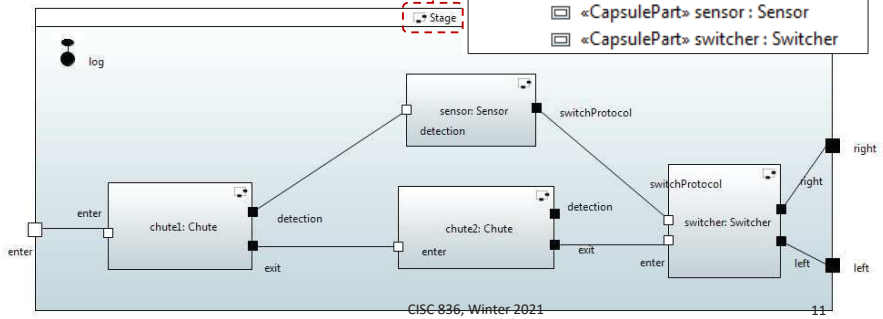




Example: Capsules and Capsule Parts

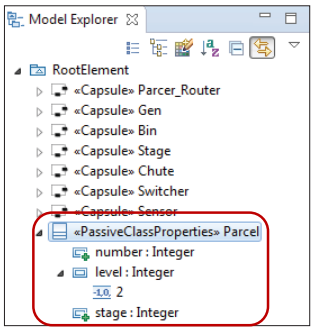






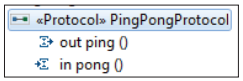
Passive Classes/Data Classes

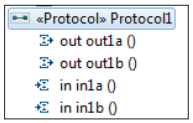
- Similar to regular classes
- Do not have independent flow of control
- Behaviour defined through operations
- Used to define data structures and operations on them

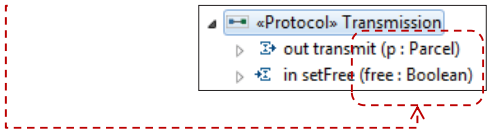


Protocols

- Provide types for ports
- Define
  - Input messages
    - Services provided by capsule owning port
  - Output messages
    - Services required by capsule owning port
  - Input/output messages
- Messages can carry data





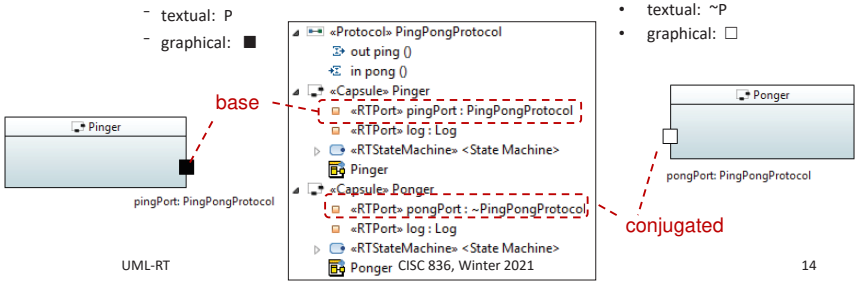


Ports

- "Boundary objects" owned by capsule
- Typed over a protocol P
- Have '**send**' operation

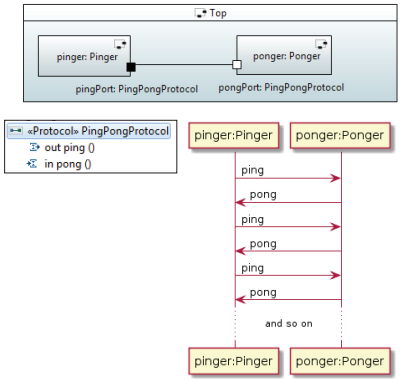
```
portName.msg(arg).send()
```

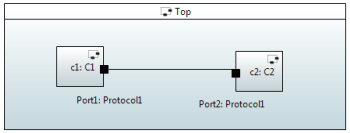
- Can be
  - base (not conjugated)
    - Direction of messages is as declared in protocol
    - Notation:
      - textual: P
      - graphical: ■
  - conjugated
    - Direction of messages declared in protocol is reversed
    - Notation
      - textual: ~P
      - graphical: □

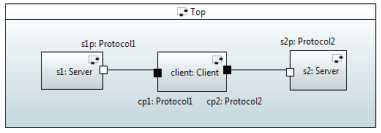


Connectors

- Connect two ports
- Ports must be compatible
  - Both are instances of same protocol
  - Either (asymmetric)
    - one is 'base' (i.e., not 'conjugated')
      - typically owned by 'client'
    - and the other is 'conjugated'
      - typically owned by 'server'
  - Or (symmetric)
    - only InOut messages

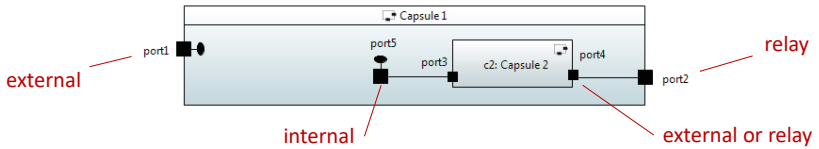




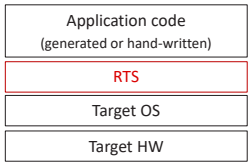


Ports: External, Internal, Relay

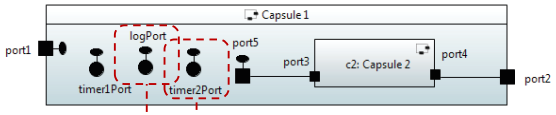
- External behaviour
  - Provides (part of) externally visible functionality (isService=true)
  - Incoming messages passed on to state machine (isBehaviour=true)
  - Must be connected (isWired=true)
- Internal behaviour
  - As above, but not externally visible (isService=false)
  - Connect state machine with a capsule part
- Relay
  - Pass external messages to and from capsule parts



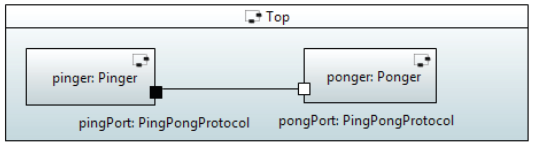
Ports: System

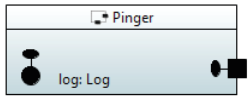


- Connects capsule to Runtime System (RTS) library via corresponding system protocol
- Provides access to RTS services such as
  - Timing: setting timers, time out message
    - `timer2Port.informIn(RTTimespec(10, 0));`
      `// set timer that will expire in 10 secs and 0 nanosecs`
    - When timer expires, '`timeout`' message will be sent over `timer2Port`
  - Log: sending text to console
    - `logPort.log("Ready to self-destruct")`
  - Frame: incarnate, destroy capsule instances

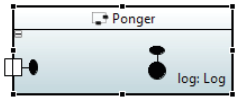


Example: PingPong



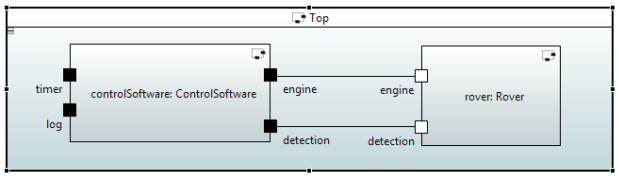


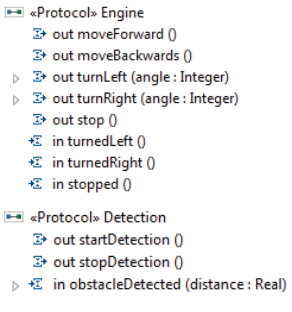
pingPort: PingPongProtocol

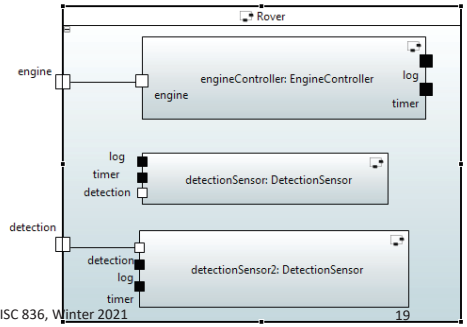


pongPort: PingPongProtocol

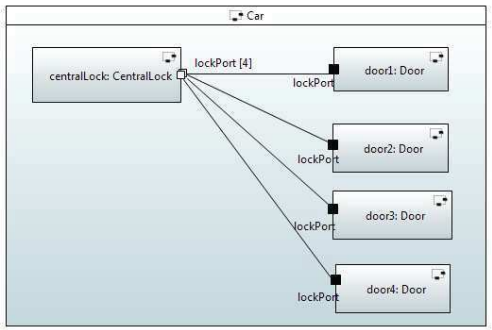
Example: Rover

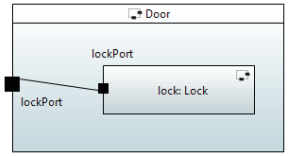




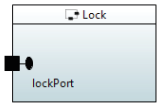


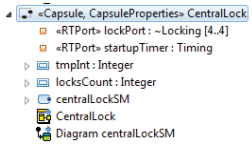
Example: Door Lock System





lockPort [4]





UML-RT/Papyrus-RT: Part II

- Core concepts
  - Structural modeling
  - Behavioural modeling

State Machines

States

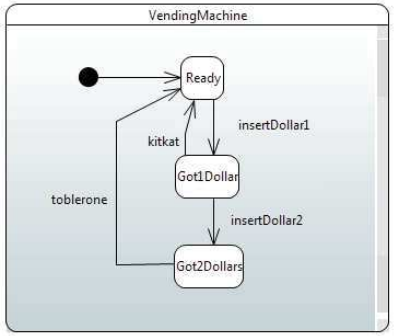
- Capture relevant aspects of history of object
- Determine how object can respond to incoming messages
- May have invariants associated with them

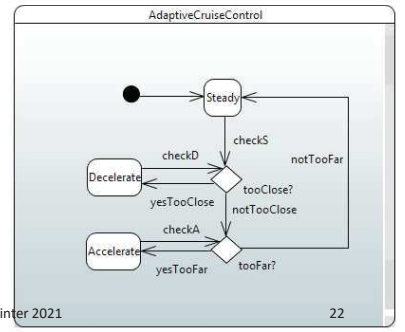
Pseudo states

- Don't belong to description of lifetime of object
  - ⇒ object cannot be 'in' a pseudo state
- Helper constructs to define complex state changes

Transitions

- Describe how object can move from one state to next in response to message input

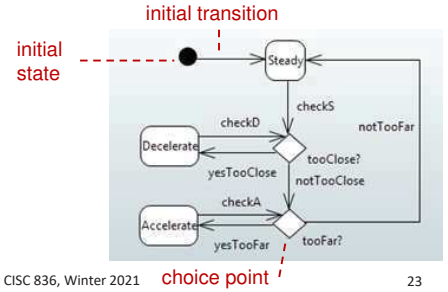




States I: Simple and Pseudo

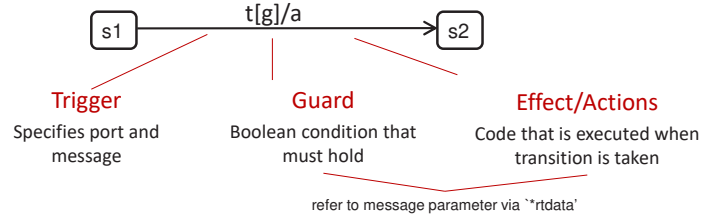
- States
  - Kinds:
    - Simple
    - Later: composite (in hierarchical state machines)
  - May contain
    - Entry action (written in action language)
    - Exit action (written in action language)

- Pseudo states I
  - initial
  - choice point



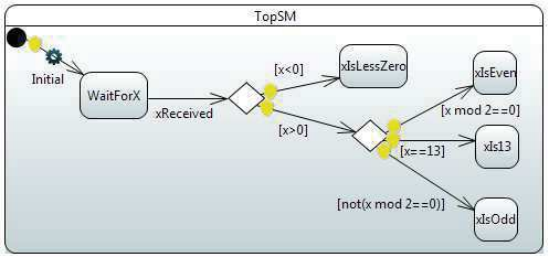
Transitions

- Kinds:
  - Basic
  - Later: group (in hierarchical state machines)
- Consists of
  - Triggers
    - Transitions out of pseudo states (initial, choice) don't have triggers
    - Transitions out of non-pseudo state should have at least one trigger
  - Guards (optional, written in action language)
    - Transitions out of initial state should not have guards
  - Effect/Actions (optional, written in action language)

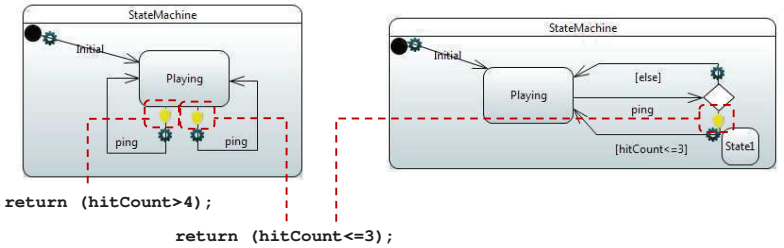


Transitions Into and Out of Pseudo States

- Initial
  - Incoming transition: impossible
  - Outgoing transition: no guard, no trigger, but can have action code
- Choice point
  - Incoming transitions: can have guard, triggers, action code
  - Outgoing transitions:
    - No trigger, but should have guard
    - Guards should be pairwise disjoint (i.e., non-overlapping)
    - Collection of guards should be exhaustive



Guards on Transitions out of Basic States




```
return (hitCount>4);
return (hitCount<=3);
```

- Dangerous: Easy to make mistakes
  - Hard to put trigger and guard info in name of transition
  - Forget that guards are there, what exactly they are
  - Have non-exhaustive or overlapping guards
- Better to use choice points

Action Language

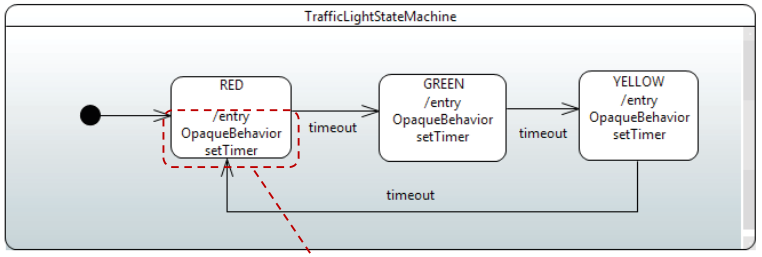
- Language used in
  - guards to express Boolean expressions
  - entry action, exit action, transition effects to read and update attribute values, send messages
- Typically: C/C++, Java

⇒ State machines are a hybrid notation combining
  - graphical notation for state machines and
  - textual notation for source code in actions

⇒ UML and UML-RT State Machines
  - different from, e.g., Finite Automata
  - closer to 'extended hierarchical communicating state machines' [6]

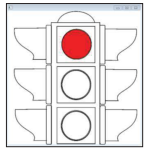
[6] R. Alur. Formal Analysis of Hierarchical State Machines. Verification: Theory and Practice. 2003.

Example: Action Code, Timers, Logging




```
$ ./TopMain.exe
Controller "DefaultController" running.
address: localhost, port: 8080
Switched to red
Switched to green
Switched to yellow
Switched to red
Switched to green
Switched to yellow
Switched to red
```

```
timer.informIn(UMLRTTimespec(5,0));
log.log("Switched to red");
```

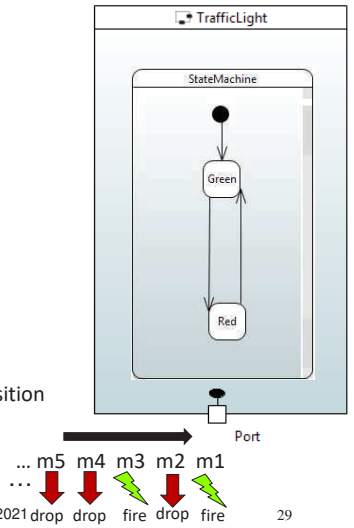


Execution Semantics I

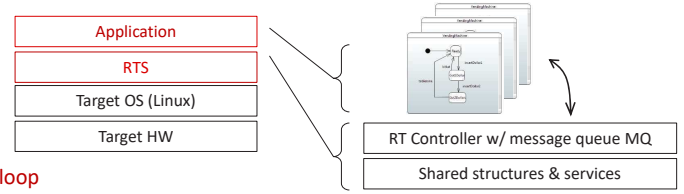
// machine is in stable state configuration

1. Message m1 has arrived and is dispatched
2. If dispatching enables no transition, m1 is 'dropped'
3. If dispatching enables transition t,
   - Source state of t active,
   - message matches trigger of t, and
   - guard evaluates to 'true'
4. then transition t executed
   a. Execute exit action of source state of t (if any)
   b. Execute action code of t (if any)
   c. Execute entry code of target state of t (if any)
5. If target of t is pseudo state,
   continue by choosing and executing outgoing transition (i.e., goto 5.)

// machine in stable state configuration



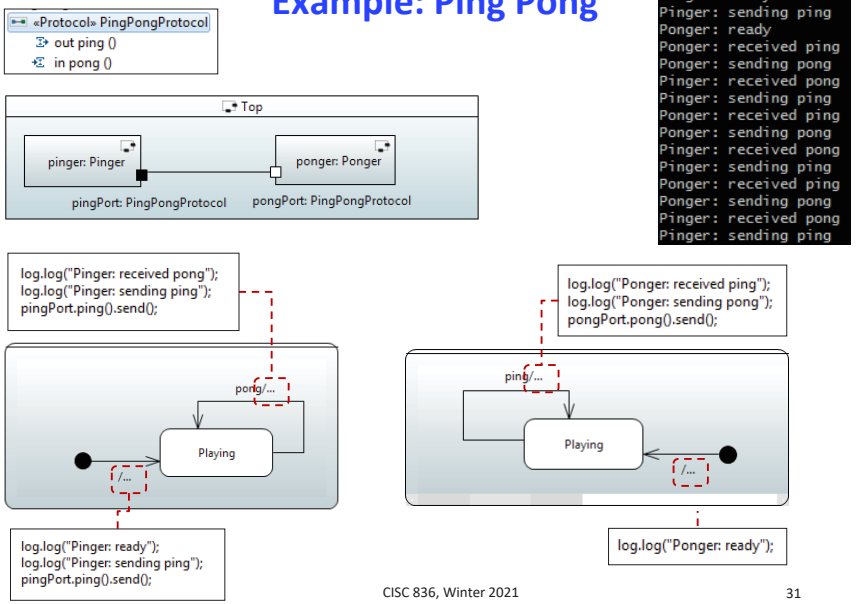
Execution Semantics I (Cont'd)



Controller main loop

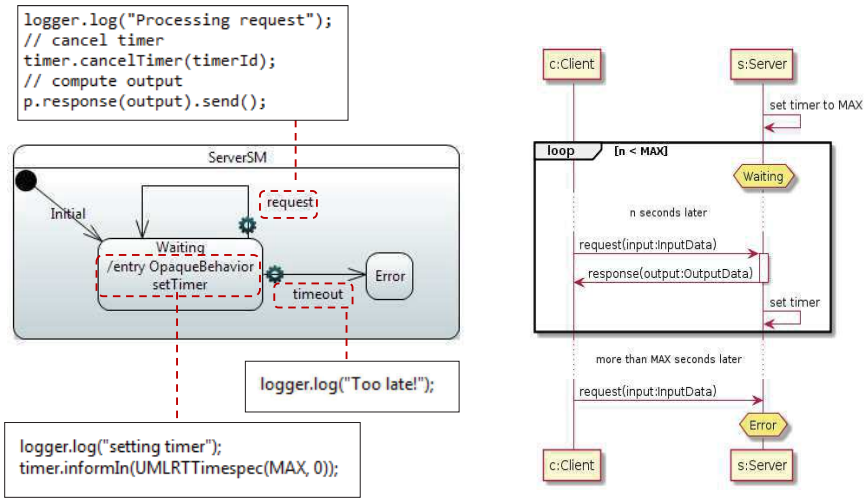
```
WHILE (1) {
  m = dequeue(MQ);
  IF can find transition t such that enabled(m,t) THEN
     targetState = execChain(t);
     mark targetState as active;
  ELSE
     report 'Unexpected message m';
}
WHERE
enabled(m,t) = source(t) is active, trigger(t) matches m, and eval(guard(t))='true'
execChain(t) = execute exit of source(t), if any;
               execute effect of t, if any;
               execute entry of target(t), if any;
               WHILE target(t) is choice point {
                  find t' such that source(t')=target(t) and eval(guard(t'))='true';
                  execute effect of t', if any;
                  execute entry of target(t'), if any;
                  t = t';
               }
               RETURN target(t);
```


Example: Ping Pong




```
$ ./TopMain.exe
Controller "DefaultCon
Pinger: ready
Pinger: sending ping
Ponger: ready
Ponger: received ping
Ponger: sending pong
Pinger: received pong
Pinger: sending ping
Ponger: received ping
Ponger: sending pong
Pinger: received pong
Pinger: sending ping
Ponger: received ping
Ponger: sending pong
Pinger: received pong
Pinger: sending ping
```


Example: Timers



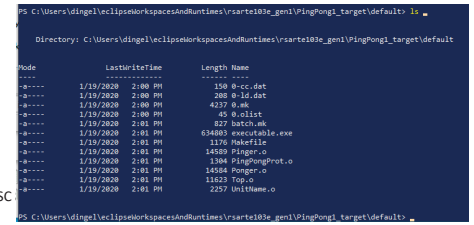
RSARTE

- Download and installation
  - Queen's version:
    - https://jahed.ca/rsarte
  - Java 8, 64 bits
- Q&A forums
  - CISC 836 pages on OnQ at http://onq.queensu.ca/

RSARTE (Cont'd)

- Use
  - (model, generate, build, run)^*
- Generated code
  - `<workspace>/<projectName>_target/`
- RTS
  - `<RSARTE installation dir>/rsa_rt/C++/TargetRTS/`
- Building generated code
  - executable:
    - `<workspace>/<projectName>_target/default`
- Running generated code
  - from inside RSARTE
  - from command line

```
>> ./executable.exe -URTS_DEBUG=quit
```



RSARTE (Cont'd)

- Tips and tricks
  - Common mistakes
    - Forgot: 'send' statement or trigger

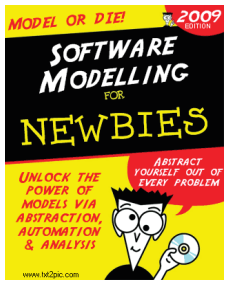
```
logP.log("[Pinger] sending first ping");
pingP.ping();
```

    - Exexcution results in a 'stackdump'? C++ issue in action code, e.g.,

```
int delay = 500000000;
logP.log("[%s] twiddling for %d nanoseconds", delay, "Pinger");
// wait for 0.5 seconds; 10^9 nsec = 1 sec
timingP.informIn(RTTimespec(0, delay));
```

    - When using 'Code View':
      - don't confuse tabs:
        - for transitions: 'effect' vs 'guard'
        - for states: 'entry' vs 'exit'
      - ensure changes saved properly
- Examples

Beyond Code:
An Introduction to Model-Driven Software Development (CISC836)



UML-RT and RSARTE: Part III

Juergen Dingel
January 2021

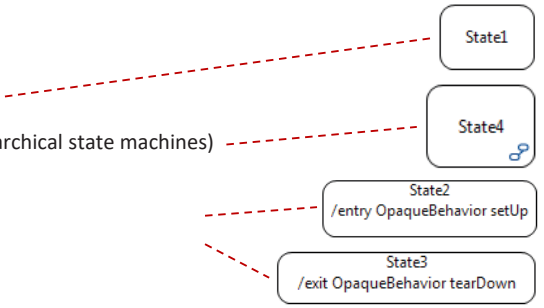
UML-RT/RSARTE: Part III

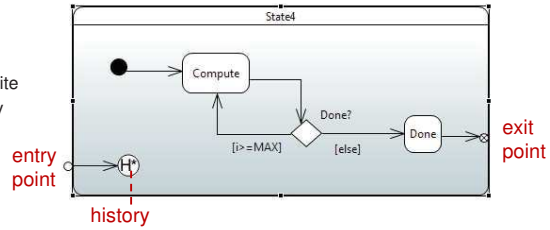
- More on
  - State machines
    - States
      - Simple
      - Composite
    - Pseudo states
      - Initial
      - Choice point
      - Entry point
      - Exit point
      - History
      - Junction
  - Execution semantics
    - Run-to-completion
- Design guidelines

States II: Composite and Pseudo

- States
  - Kinds:
    - Simple
    - Composite (in hierarchical state machines)
  - May contain
    - Entry action
    - Exit action
- Pseudo states
  - initial
  - choice point
  - history
  - entry point
  - exit point
  - junction point

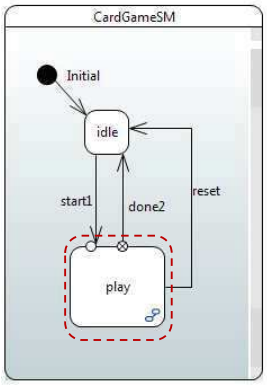
in composite states only

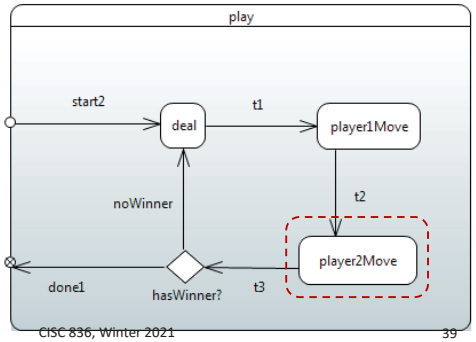




Group Transitions

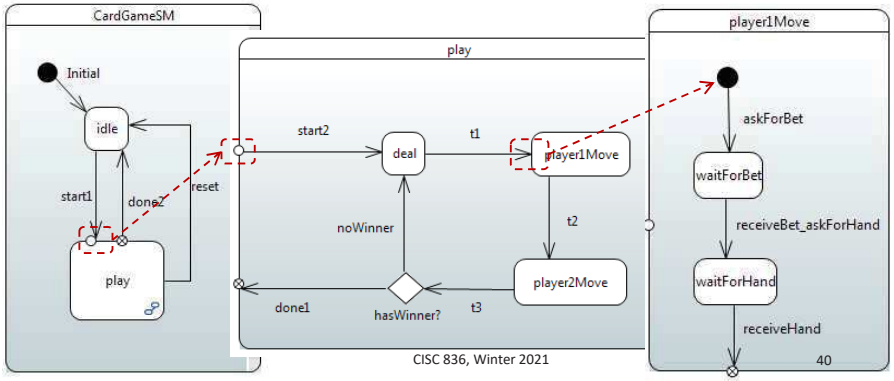
- Source state is composite
- Example:
  - Start configuration <'play','player2Move'>
  - Execute transition 'reset':
    - exit code 'player2Move', exit code 'play', effect 'reset', entry code 'idle'
  - End configuration <'idle'>





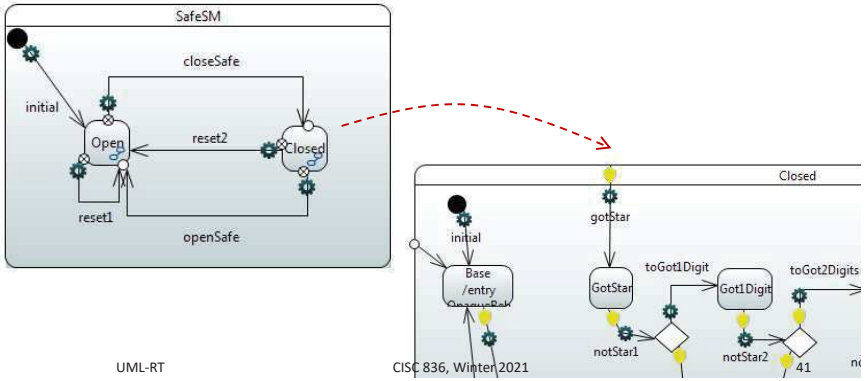
State Configuration

- States can be active: flow of control resides at state
- If a substate is active, its containing superstate is, too
- State configuration: list of active states
- Stable state configuration: no pseudo states and ends in basic state
- Example: <'play', 'player1Move', 'waitForHand'>



Entry and Exit Points

- Required boundary pseudo states for transitions crossing boundaries of composite states
- Transition ending at entry point w/o outgoing transitions: implicit return to history



Run-to-Completion

- The event processing of state machines follows 'run-to-completion' semantics
- Dispatching of message triggers execution of possibly entire chain of transitions ('exec' on previous slide)
- Execution lasts until stable state configuration has been reached (last state in transition chain not a pseudo state)
- During transition execution, no other message will be dispatched

⇒ execution triggered by message treated as one unit

⇒ no 'interleaved' processing of messages

⇒ less potential for bugs

Execution Semantics II

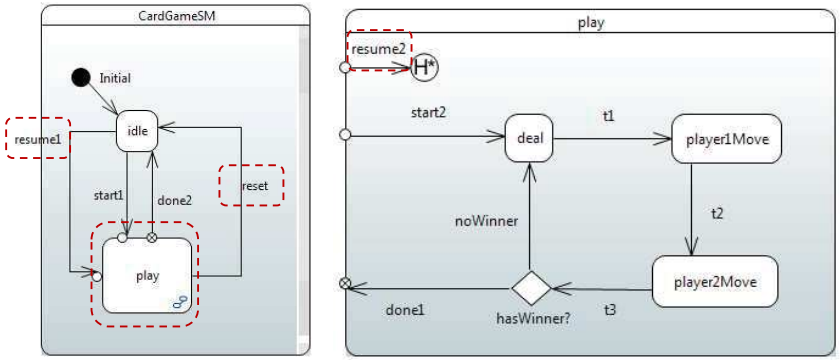
Controller main loop

```
WHILE (1) {
  m = dequeue(MQ);
  IF can find transition t such that enabled(ssc,m,t) THEN ssc = exec(ssc,t);
  ELSE report 'Unexpected message m';
}
WHERE
enabled(ssc,m,t) = (1) source(t) is active, (2) trigger(t) matches m,
                   (3) eval(guard(t))='true', and
                   (4) source(t) does not contain any other state satisfying (1),(2),(3)
exec(ssc,t) = LET ssc=<s1, …, si-1, si, si+1, …, sn> where si=source(t) IN
                FOR j=n to i+1 {execute exit of sj};
                targetOfChain = execChain(t);
                sk = leastCommonAncestor(source(t), targetOfChain);
                LET <sk, s'1, …, s'm> be containment hierarchy where s'm=targetOfChain IN
                   RETURN <s1, …, sk-1, sk, s'1, …, s'm>
execChain(t) = execute exit of source(t), if any;
               execute effect of t, if any;
               execute entry of target(t), if any;
               WHILE target(t) is pseudo state {
                  find t' such that source(t')=target(t) and eval(guard(t'))='true';
                  execute exit of state(source(t')), if any;
                  execute effect of t', if any;
                  execute entry of state(target(t')), if any;
                  t = t';
               }
               RETURN target(t);
```

UML2.5 Spec, Section 14.2.3
http://www.omg.org/spec/UML/2.5/PDF

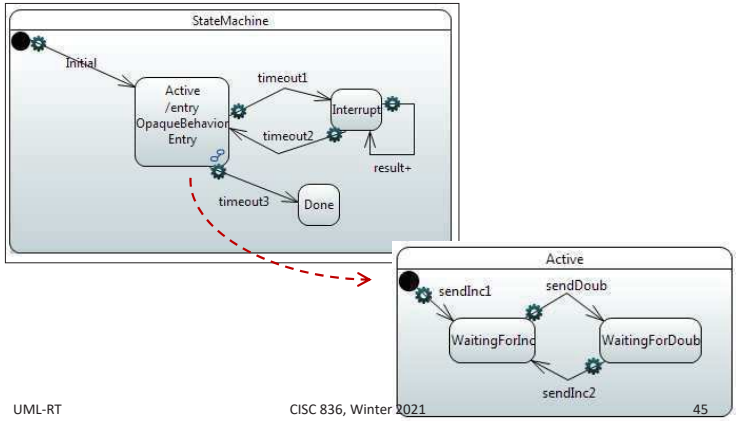
History

- Re-establish full state configuration that was active when containing state was active most recently
- If entering state for first time, go to initial state
- Example: from <'play', s> to <'play', s> with 'reset' 'resume1'

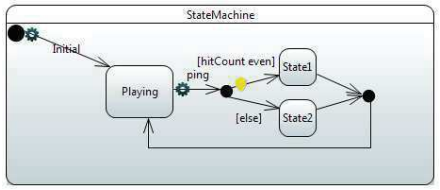


History (Cont'd)

- History pseudo state does not need to be given explicitly
- Transition ends at boundary of composite state: Implicit return to history

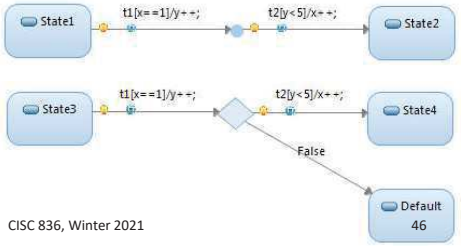


Junction Points



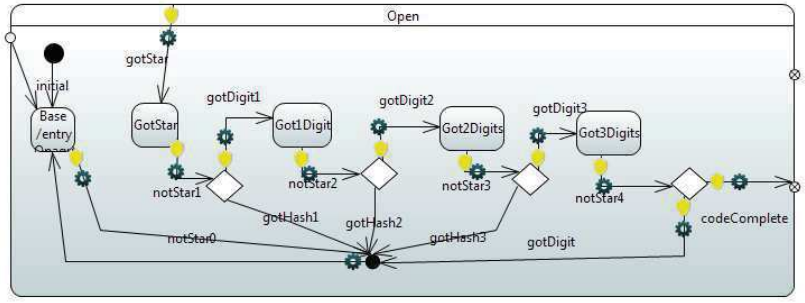
- Can be used to split and merge control flow
- Warning:
  - Static evaluation: All guards on transitions connected by junction points evaluated BEFORE first transition is taken
  - Transitions taken only when fully enabled path exists
- Choice points
  - Dynamic evaluation:

  Guards evaluated as transitions are executed
- Pros/cons?



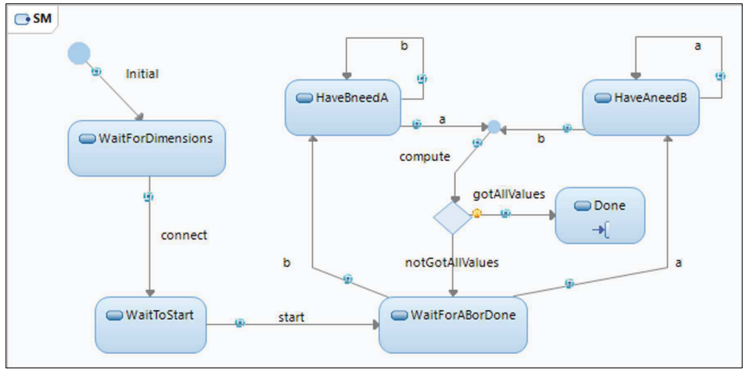
Junction Points (Cont'd)

- Merge useful to avoid duplication of action code



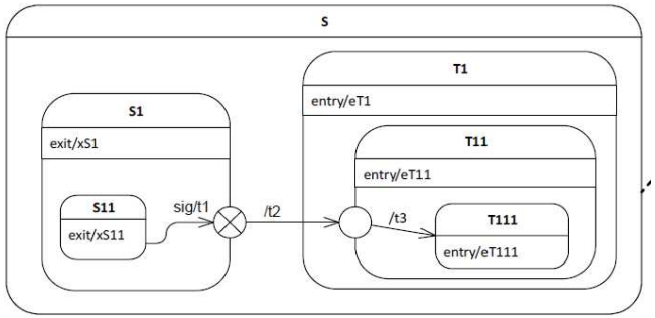
Junction Points (Cont'd)

- State machine of processor in matrix multiplication:



Putting It All Together I

- Assume
  - State configuration: <S,S1,S11>
  - Message ‘sig’ dispatched to state machine
- What happens?

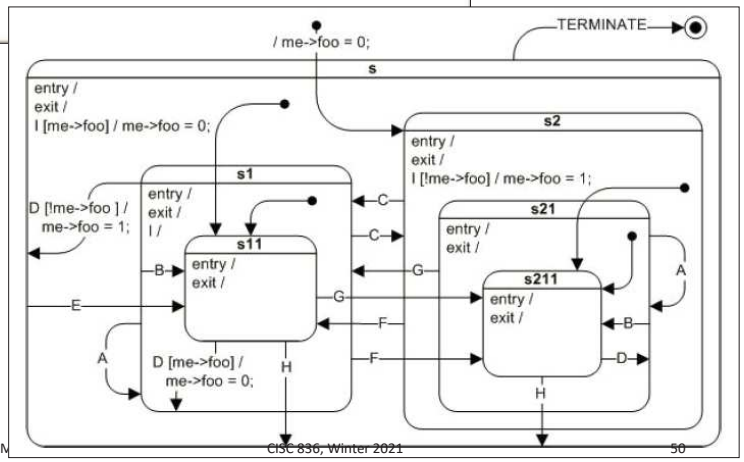


[UML2.5.1] UML Specification v2.5.1. Dec 2017. Page 381
https://www.omg.org/spec/UML/2.5.1/PDF

Putting It All Together (Cont’d)

```
QHsmIst example, built on Sep 25 2007 at 09:11:31,
QEP/C: 3.4.01,
Press ESC to quit...
top-INIT;s-ENTRY;s2-ENTRY;s2-INIT;s21-ENTRY;s211-ENTRY;
>G: s21-G;s211-EXIT;s21-EXIT;s2-EXIT;s1-ENTRY;s1-INIT;s11-ENTRY;
>I: s1-I;
>A: s1-A;s11-EXIT;s1-EXIT;s1-ENTRY;s1-INIT;s11-ENTRY;
>D: s1-D;s11-EXIT;s1-EXIT;s-INIT;s1-ENTRY;s11-ENTRY;
>D: s11-D;s11-EXIT;s1-INIT;s11-ENTRY;
>C: s1-C;s11-EXIT;s1-EXIT;s2-ENTRY;s2-INIT;s21-ENTRY;s211-ENTRY;
>E: s-E;s211-EXIT;s21-EXIT;s2-EXIT;s1-ENTRY;s11-ENTRY;
>E: s-E;s11-EXIT;s1-EXIT;s1-ENTRY;s11-ENTRY;
>G: s11-G;s11-EXIT;s1-EXIT;s2-ENTRY;s21-ENTRY;s211-ENTRY;
>I: s2-I;
>I: s-I;
>+: Bye, Bye!
```

[Sam09] M. Samek. A Crash Course in UML State Machines. Article based on Chapter 2 of the book Practical UML Statecharts in C/C++ 2nd Edition. March 2009.



UML-RT: Design Guidelines

- General
  - Names
    - Descriptive, correct (syntactically and semantically), consistent
  - Readable, clear layout of models
  - Remove/cancel what is not needed anymore (timers, capsule parts)
  - Avoid duplication (through, e.g., operations, junction points for merging, entry and exit code)
- Capsules
  - Low coupling, high cohesion (look at connectors, message traffic, protocols)
  - Avoid overly deeply nested capsule definitions
- State machines
  - Avoid unreachable states and transitions
  - Avoid overly deeply nested composite states
  - Avoid composite states with only one substate

UML-RT: Design Guidelines (Cont’d)

- Action code
  - Short, simple, terminating, readable, reachable (i.e., not dead)
  - Avoid ‘hidden’ states (e.g., flags and complex control flow)
- Junction points
  - Only use for merging
- Transitions
  - Guards: short, simple, readable, side-effect-free
  - Out of choice points: at least two, guards exhaustive and exclusive, no trigger
  - Out of initial, entry, exit, junction: no guard, no trigger
  - Out of non-pseudo state: no guards
  - Use different kinds (external, local, internal) appropriately
  - Avoid dropped, ‘unexpected’ messages
  - Make copy of complex message parameters upon receipt
  - Can’t cross ‘state boundaries’ w/o going through an entry or exit point

UML-RT: Design Guidelines (Cont'd)

- Correct use of constructs and services offered by UML-RT, RTS or C++
  - Random number generator
    - Initialize once at startup (e.g., using `srand(time(0))`)
  - Replication
- Observability
  - Insert informative log statements at suitable places to facilitate reasoning about the model (debugging, error localization)

    Format:

    `Logger.log("[Name of capsule part](Name of state)…(Name of substate) info")`

    where 'info' describes
    - message and/or data received, or
    - attribute values
  - Consider use of command-line parameters to facilitate testing