Beyond Code: An Introduction to Model-Driven Software Development (CISC 836, Fall 2021)

UML-RT Sample Models

Instructions on how to import these models into RSARTE can be found on the Assignment 1 page.

Models created specifically to illustrate certain constructs in UML-RT or features of RSARTE

1. **PingPong 1:**
   - most basic
   - [PingPong 1 to PingPong 7.zip]
2. **PingPong 2:**
   - attributes and choice points
   - 'Pinger' uses 'exit()' (in transition to state 'End'); for this to be possible 'Implementation Preface' of 'Pinger' (select 'Pinger' and under 'Properties' click 'C++ General') must contain '#include <stdlib.h>'
3. **PingPong 3:**
   - timers
4. **PingPong 4:**
   - command line arguments and random number generator
   - to be able to use the functions 'rand()' and 'time()' from C++ library 'time.h', the 'Implementation Preface' of 'Top' (select 'Top' and under 'Properties' look for 'C++ General') must contain '#include <stdlib.h>' and '#include <time.h>'
5. **PingPong 5:**
   - illustates a client/server relationship: client sends two request 'inc(x)' and 'doub(x)' in alternating fashion; server responds with 'result(x)'
6. **PingPong 6:**
   - version of PingPong5 that illustrates the 'hidden state' anti-pattern: 'Client' keeps track of which message to send next via a flag ('waitingForInc')
7. **PingPong 7:**
   - composite states, group transitions, and timers
8. **PingPong 8:**
   - composite states, history (via transition 'timeout2' in 'Client'), unexpected messages (due to state 'Interupt' in 'Client')
   - try:
     1. set attribute 'INTERRUPTS_ENABLED' in 'Client' to 'true'
     2. add transition 't' from 'Interrupt' to 'Error' with 'result' as trigger
     3. prevent 't' from ever being enabled by adding a guard that always evaluates to 'false'
   - [PingPong 8 to PingPong 10.zip]
9. **Samek example**
   - UML-RT implementation of the hierarchical state machine on page 11 of [Sam09], which uses UML State Machines, not UML-RT.
   - In the context of this example, we note the following differences between state machines in UML and UML-RT:
     - Notationally, we see that in UML state machines transitions can cross the boundary of a composite state without the use of entry and exit points.

        - In terms of the execution semantics, in UML a transition 't' ending at the boundary of a composite state 's' always takes the state machine to the initial state of 's', whereas in UML-RT they are treated as an implicit return to history, i.e., 't' takes the machine back to the most recently active subset of 's' and only to the intial state of 's' if 's' has not been entered before.
    - In the model, connecting the harness `h` to the capsule part `c` means that a fixed sequence of messages will be fed into `c`. Alternatively, messages can be fed into `c` with the model debugger via the relay capsule part `r`. To this end, `c` must be connected to `r` and, after starting the model debugger, essages be fed into `r`'s `relayControlPort`.
    - [SamekExample.zip]
10. **PingPong 9**
    - sending messages with more than one parameter using a data class
    - class 'Args' packages all parameters for message 'init' from 'Top' to 'Pinger'
    - in 'Top' and 'Pinger', dependency to class 'Args' must be set with 'Kind In Header' set to 'none' and 'Kind in Implementation' set to 'inclusion'
11. **PingPong 10**
    - correct version of PingPong 8 using 'defer/recall'
12. **PingPong 11**
    - timer, measuring response times
    - [PingPong11.zip]
13. **Multplicity 1**
    - port replication
    - to set the multiplicity of a port 'p' to 'n..m', select 'p', then click 'Advanced' under 'Properties'. Find 'Lower' and 'Upper' and set them to 'n' and 'm', respectively. Setting the multiplicity of a capsule part is similar (select the part and set 'Lower' and 'Upper' under 'Properties' and 'Advanced'.
    - [Multiplicity 1 to Multiplicity 2.zip]
14. **Multplicity 2**
    - port replication and capsule replication
15. **DeferRecall 1**
    - defer/recall
    - [DeferRecall.zip]
16. **GiveChange**
    - Version v0: the vast majority of the functionality is put into the effect of a transition; as a result, the state machine is simple, but the action code is complex and contains several loops
    - Version v1: 'Changer' state machine makes the 3 stages of the process (giving toonies, giving loonies, and checking for zero) visible; 'tick' messages necessary to advance the computation; 'Changer' capsule does not contain any structure
    - Version v2: 'Changer' capsule contains three parts ('toonies', 'loonies', and 'zerocheck'), one for each of the states of the process
    - [GiveChange.zip]
17. **Multi-threading**
    - Small, condensed example illustrating the effect of running two capsules on different threads. Takes two command line arguments: the first is used to indicate if capsule parts `b` and `c` should be run on the same thread (argument "same") or on different threads (argument "different"); the second determines how long the execution of the transition in capsule part `b` will take; e.g.,
      ```
      ./executable.exe -URTS_DEBUG=quit -UARGS "different" 25000
      ```
    - [ThreadAllocationMatters.zip]
18. **Console input, multi-threading, and animation**
    - Example illustrating how to read input from the console during execution. Model implements a simple traffic light whose delay can be changed at runtime by the user. An integer is read in by an optional capsule `InputInt` upon request from the `TrafficLight` capsule and then used when setting the timers used to transitions between the states. Project comes with two transformation configurations: one single-threaded and another in which capsule `InputInt` runs in its own thread. In single-threaded executions, the traffic light 'freezes' while capsule `InputInt` is waiting for input, while in the multi-threaded executions the traffic light continues to operate

- Start the Java animation of the traffic light before the model
- Also illustrates 'internal transitions' which is used in the composite state 'Active' in traffic light to receive the user input
- [TrafficLightWithConsoleInputAndAnimation.zip]

19. **Plugin capsules**
    - Three models to illustrate the use of plugin capsules (and binary search). Capsule `Number` randomly picks a secret number within an interval that a collection of optional `Guesser` capsules have to guess. The `Top` capsule gives the guessers a turn in a round-robin fashion. To guess, a guesser plugs in an instance of `Number`, and then asks for a hint. A hint is an indication whether or not the secret number is less than or greater than the previously guessed number, thus narrowing the interval that the secret number must be in. The guesser then issues a guess to the `Number` capsule by picking the number in the middle of the interval. If the guess is correct, the game ends. If the guess is not correct, the interval is updated and the next guesser gets a turn
    - The number of guesser capsule instances and the upper bound of the interval (initially, the lower bound will always be 0) can be supplied as command line arguments
    - The number of guesses required is bounded above by the base-2 logarithm of the initial size of the interval. Note that the number of guesses required is independent of the number of guessers. E.g., in case of the interval 0..1000, no more than 10 guesses should be required regardless of how many guessers were used.
        - Version v1: basic
        - Version v2: as v1, but also illustrates how 'Top' can continue to communicate with 'Number' even when 'Number' is plugged in: during the guessing, 'Top' can send a 'reset' message which will cause the plugged in capsule to change its secret. Sending of 'reset' depends on random choice made by 'Top' whenever number connects to a guesser (reset probability 10%)
        - Version v3: as v2, but also illustrates capsule specialization (subclassing): The specializing capsule ('numberPlusCheat') has an additional port ('cheatP') which is used through additional transitions in the state machine. Which kind of capsule (basic: 'Number', cheating possible: 'NumberPlusCheat') is used is determined by 'Top' which in its initial transition incarnates one of these and then passes the id to the importing guesser. The guesser uses this id for the import, but also to determine which class the instance is an instances of (using 'classOf()' and 'className()') and, thus, to see if the instance plugged in supports cheating or not.
    - [MoreOrLess_v1_to_v3.zip]

Models of classic examples from the literature

1. **Sieve of Eratosthenes**
    - Model to compute prime numbers between 2 and some user-provided upper bound <max> using Sieve of Erathostenes.
    - Invoke with './executable.exe -URTS_DEBUG=quit -UARGS <max>' or './executable.exe -URTS_DEBUG=quit' in which case <max>=100 by default. Current multiplicity constraints allow for the creation of 10,000 instances of the 'Factor' capsule (in which case <max>=104729)
    - Illustrates the use of optional capsules, replication (for ports and parts), and unwired ports (which are wired manually using explicit SAP/SPP registration). It also shows how to pass data into the incarnated capsule without the use of messages (data is additional argument of 'incarnate' and made available as 'rtdata' in the initial transition of the incarnated capsule).
    - Dynamically created optional capsules instances (called 'factor[i]' in 'Top' with 0<i<10000) form a pipeline whose first element is connected to a generator capsule (called 'gen') that outputs the numbers between 2 and <max>. Each capsule checks if the incoming number is divisible by one of the prime numbers already found. If a number n reaches the end of the pipeline (i.e., the right-most factor capsule), n is prime and passed to 'Top' for output; 'Top' will also create a new capsule which will use n as factor and be linked into the pipeline as new last (right-most) element.
    - [Sieve.zip]
2. **Fibonacci numbers**
    - Model to compute Fibonacci numbers the naive, exponential way

- Illustrates optional capsules (w/ incarnation arguments), unwired ports with manual, dynamic wring (using SPP/SAP), and replication (ports and parts, [1..10000] by default)
- Dynamically created optional capsule instances (called 'fib[i]' in 'Top with 0<i<10000) form a tree whose root element is connect to 'Top'. Each 'fib[i]' first checks the number `n` it is to compute. If `n` is 0 or 1, the result is returned right away. Else, it asks 'Top' to create two new instances (one to compute fib(n-1) and the other to compute fib(n-2)), waits for the results produced by these instances, and then returns their sum.
- Id `i` of each instance `fib[i]` is passed into the instance as argument to `incarnate`; id is used to compute SPP/SAP topic used to connect instance with its parent and two children (if any).
- Command line arguments: `n`, i.e., number to compute the Fibonacci number of
- Example: computing fib(18) requires 8361 capsule instances; 'fib(19)' requires more than 10K instances and thus crashes the application
- [Fibonacci.zip]

3. **Matrix multiplication**
   - UML-RT implementation of Cannon's algorithm for 2D meshes. Uses optional capsules and dynamically wired (unwired) ports to create a suitably connected processor grid at runtime.
   - Other UML-RT features used are choice points, operations, data classes, port and capsule replication, and defer/recall. To allow for more of the algorithm steps to become visible on the level of the state machines, it also uses 'tick' messages that a capsule sends to itself to trigger the next transition (as in GiveChange v1)
   - [MatMult.zip]
4. **Dining philosophers**
   - Three models implementing different versions of the classical Dining Philosophers problem.
   - Version v1: Illustrates fixed capsules w/ wired ports, enumeration types (`PickUpStrategy`), dependencies (`Phil` using `PickUpStrategy`), random number generation, and 'getName()' to determine name of capsule instance.
   - Version v2: Illustrates optional capsules w/ wired ports, command line arguments, use of `msg->sap()` to determine port that triggering message came in on.
   - Version v3: Illustrates optional capsules w/ unwired ports (wired manually by application using SPP/SAP w/ dynamically computed topic), passing of initialization arguments via `incarnate` (received in initial transition of incarnated instance).
   - [DiningPhils_v1_to_v3.zip]
5. **Alternating bit protocol**
   - Model implementing the classical Alternating Bit Protocol.
   - Top capsule 'TopABP_conn' uses an instance of 'ConnectorABP' to send messages from port 'inP' to 'outP'. The instance is configured via an initialization message 'init' that contains the probability of the loss of a message ('lossProbability') and the retransmission delay ('rtxDelay') as argument.
   - [AlternatingBitProtocol.zip]

Last modified: Tue Sep 21 2021 12:38:51