Institute of Communication, Information and Perception Technologies
Scuola Superiore Sant'Anna

Retis Real-Time Systems Laboratory

RTSIM
Real-Time system SIMulator

Mauro Marinoni

ReTiS Lab, TeCIP Institute
Scuola superiore Sant'Anna - Pisa

Agenda

- RTSIM
  - MetaSim
  - RTLib
- Examples
- Project proposals

RTSIM

Institute of Communication, Information and Perception Technologies
Scuola Superiore Sant'Anna

Retis Real-Time Systems Laboratory

RTSIM

Overview

Real-Time system SIMulator

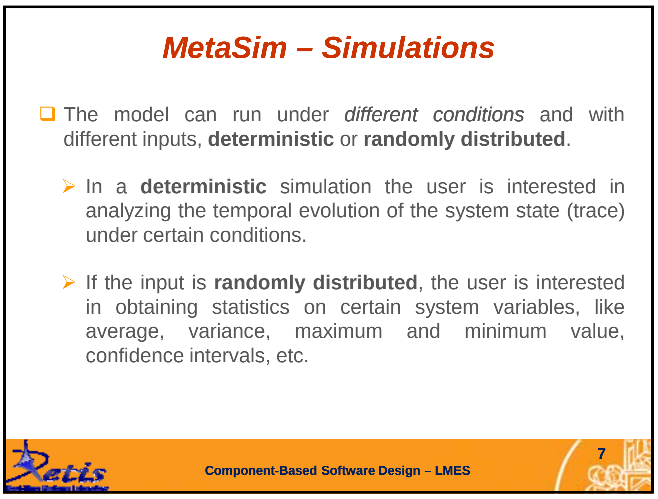
- **RTSIM** is a collection of programming libraries written in C++ for simulating real-time control systems.
- It has been developed for experimenting with new scheduling algorithms and solutions.
- It already contains some classic real-time scheduling algorithms.
- It consists of 2 components:
  - **MetaSim:** it is a generic library for simulation of discrete event systems.
  - **RTLib:** based on MetaSim, it is a library for simulating scheduling algorithms and real-time tasks.

RTSIM - Installation

- Both **MetaSim** and **RTLib** are available on *GitHub*:
  - https://github.com/glipari/metasim
  - https://github.com/glipari/rtlib2.0
- Configurable using Cmake
- Can be compiled under Unix, Mac and Windows
- Testing support written using the Catch library
- Documentation can be generated using the Doxygen tool

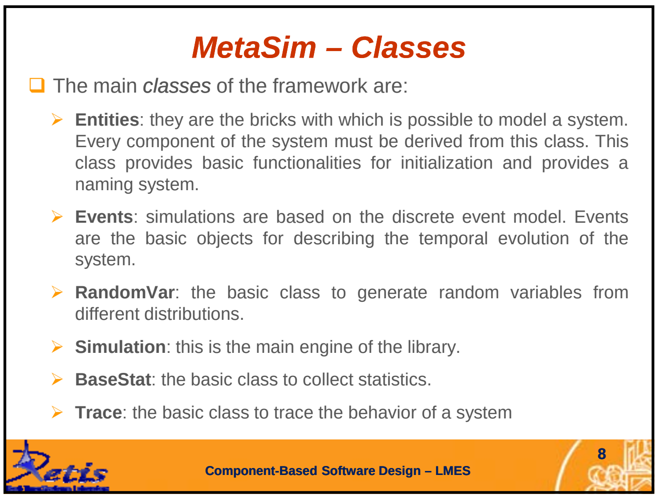
MetaSim

- **MetaSim** is a framework that allows to *develop customized simulation entities*.
- Each entity that can post discrete events in time.
- The framework also provides some useful classes in order to:
  - handle random variables;
  - collect statistical values;
  - write traces and debug output.
- The user can analyze the behavior of a system without actually having it.



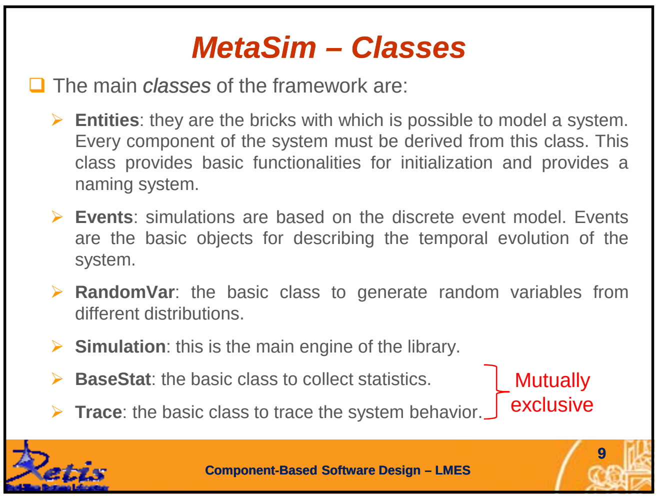
MetaSim – Simulations

- The model can run under *different conditions* and with different inputs, **deterministic** or **randomly distributed**.
  - In a **deterministic** simulation the user is interested in analyzing the temporal evolution of the system state (trace) under certain conditions.
  - If the input is **randomly distributed**, the user is interested in obtaining statistics on certain system variables, like average, variance, maximum and minimum value, confidence intervals, etc.



MetaSim – Classes

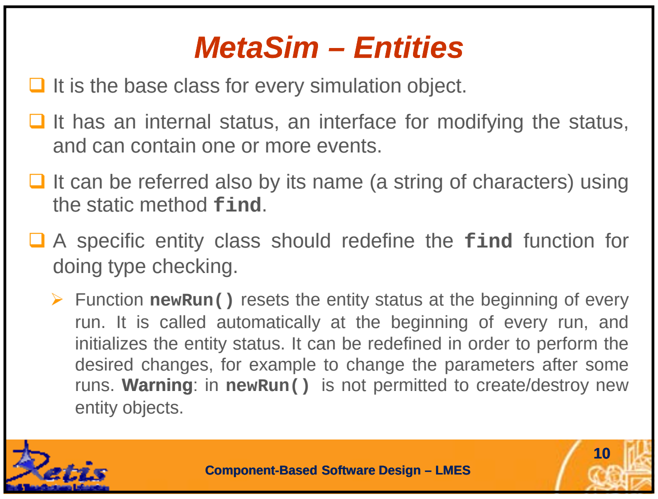
- The main *classes* of the framework are:
  - **Entities**: they are the bricks with which is possible to model a system. Every component of the system must be derived from this class. This class provides basic functionalities for initialization and provides a naming system.
  - **Events**: simulations are based on the discrete event model. Events are the basic objects for describing the temporal evolution of the system.
  - **RandomVar**: the basic class to generate random variables from different distributions.
  - **Simulation**: this is the main engine of the library.
  - **BaseStat**: the basic class to collect statistics.
  - **Trace**: the basic class to trace the behavior of a system



MetaSim – Classes

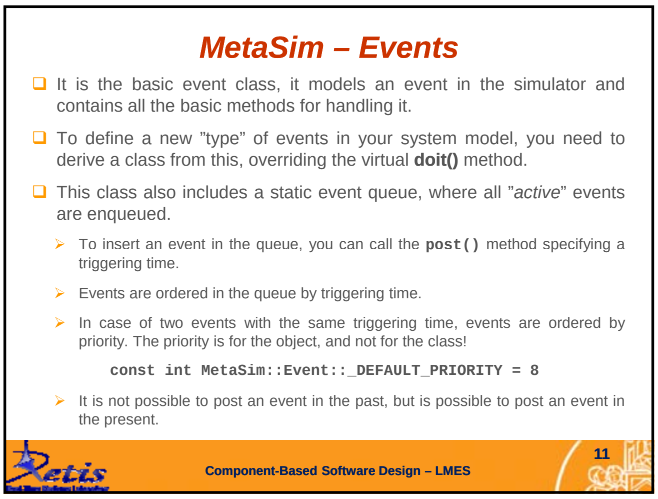
- The main *classes* of the framework are:
  - **Entities**: they are the bricks with which is possible to model a system. Every component of the system must be derived from this class. This class provides basic functionalities for initialization and provides a naming system.
  - **Events**: simulations are based on the discrete event model. Events are the basic objects for describing the temporal evolution of the system.
  - **RandomVar**: the basic class to generate random variables from different distributions.
  - **Simulation**: this is the main engine of the library.
  - **BaseStat**: the basic class to collect statistics.
  - **Trace**: the basic class to trace the system behavior.

Mutually exclusive



MetaSim – Entities

- It is the base class for every simulation object.
- It has an internal status, an interface for modifying the status, and can contain one or more events.
- It can be referred also by its name (a string of characters) using the static method `find`.
- A specific entity class should redefine the `find` function for doing type checking.
  - Function **`newRun()`** resets the entity status at the beginning of every run. It is called automatically at the beginning of every run, and initializes the entity status. It can be redefined in order to perform the desired changes, for example to change the parameters after some runs. **Warning**: in **`newRun()`** is not permitted to create/destroy new entity objects.

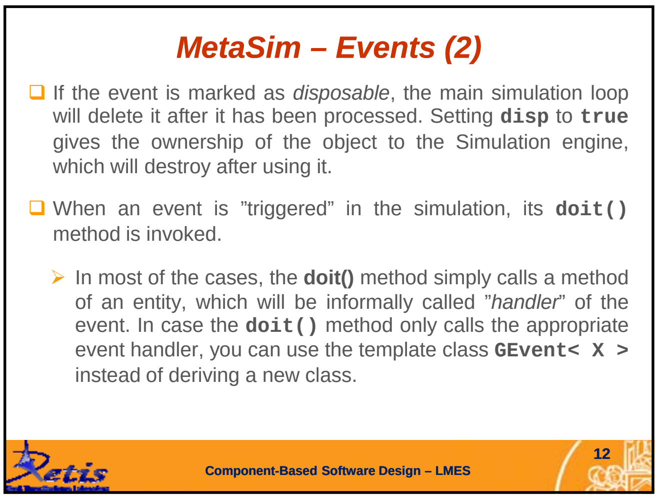


MetaSim – Events

- It is the basic event class, it models an event in the simulator and contains all the basic methods for handling it.
- To define a new ”type” of events in your system model, you need to derive a class from this, overriding the virtual **doit()** method.
- This class also includes a static event queue, where all ”*active*” events are enqueued.
  - To insert an event in the queue, you can call the **`post()`** method specifying a triggering time.
  - Events are ordered in the queue by triggering time.
  - In case of two events with the same triggering time, events are ordered by priority. The priority is for the object, and not for the class!

```
const int MetaSim::Event::_DEFAULT_PRIORITY = 8
```

  - It is not possible to post an event in the past, but is possible to post an event in the present.

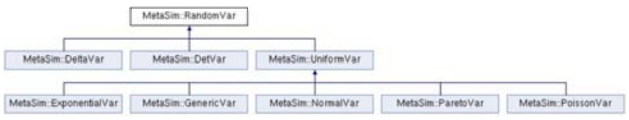


MetaSim – Events (2)

- If the event is marked as *disposable*, the main simulation loop will delete it after it has been processed. Setting **`disp`** to **`true`** gives the ownership of the object to the Simulation engine, which will destroy after using it.
- When an event is ”triggered” in the simulation, its **`doit()`** method is invoked.
  - In most of the cases, the **doit()** method simply calls a method of an entity, which will be informally called ”*handler*” of the event. In case the **`doit()`** method only calls the appropriate event handler, you can use the template class **`GEvent< X >`** instead of deriving a new class.

MetaSim – RandomVar

- This class implements a random variable: some derived classes provide more detailed implementations depending on the type of random variable.
- The class diagram is…



MetaSim – Simulation

- This singleton implements the simulation engine and some debugging facilities.
- The main function is **`run(Ticklenght, size_t runs)`** that is responsible for running the simulation for one or more times.
  - After defining all the objects in a simulation, this function should be invoked for running the simulation.
  - Example: **`Simulation::run(10000, 10);`**
- At the beginning and at the end of each run, initialization and finalization are called. They are named **`Entity::newRun()`** and **`Entity::endRun()`**, respectively.
- The **`getTime()`** returns the current *globalTime* in the simulation.
- The random seed is not initialized at every run, but it is left as it is.

MetaSim – BaseStat

- This class is used to collect statistical values.
- The first two levels of abstraction are implemented, while the third is left to the programmer, depending on his needs.
  - **Level 0 (BaseStat)** implements the functions for doing statistic: it is initialized with the number of experiments to be done, and has an array to record the data at the end of the simulation.
    - It has two pure virtual function, **`probe()`** and **`record()`**, so no object of this class can be instantiated.
    - It has some function to get the final stats, like **`getMean()`** and **`getConfInterval()`**.
  - **Level 1 (StatMax)** implements the function for collecting statistic; for example, the StatMax class record the maximum value during an experiment.
    - The user can add his own classes to this level, and he must implement the **`record()`** function and the **`initValue()`** function.
  - **Level 2** implements the **`probe()`** for a single event. The user must write it depending on the variable he needs to measure. When implementing a class of this level, the user must write the probe function, which has to call the **`record()`** function with the appropriate value.

MetaSim – Trace

- This class allows programmers to trace variables on a stream. By default it opens a binary stream.
- The derived class TraceASCII put the traced values into an ASCII stream.

INSTITUTE OF COMMUNICATION, INFORMATION AND PERCEPTION TECHNOLOGIES

Scuola Superiore Sant'Anna

Retis Real-Time Systems Laboratory

RTLIB

RtLib

- **RtLib is a library for Real-Time Kernels Simulation written in C++**
- RtLib provides software modules needed to simulate a real-time kernel;
- It is based on MetaSim, so all classes are derived from Entity.

RtLib – Modules

- Some basic modules are:
  - **RTKernel**: The base module of a real-time operating system.
  - **Scheduler**, **TaskModel**: Modules that provide methods to handle task' s priority queues and scheduling parameters.
  - **Task**: Base class that implements task functionalities.
  - **Instr** : The base class for every pseudo-instruction.
  - **TextTrace**: Module that let each event to be traced in a text file.

RtLib – Modules (2)

- There are other simulation entities provided by RTLib that are:
  - CPU
  - PollingServer
  - SporadicServer
  - Grub
  - Supervisor
  - Resource
  - ResManager

RtLib – RTKernel

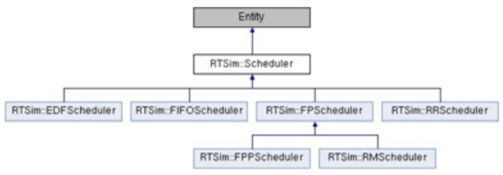
- An implementation of a real-time single processor kernel. It contains:
  - a pointer to one CPU;
  - a pointer to a Scheduler, which implements the scheduling policy;
  - a pointer to a Resource Manager, which is responsible for resource access related operations and thus implements a resource allocation policy;
  - the set of task handled by this kernel.
- This implementation is quite general: it lets the user of this class the freedom to adopt any scheduler derived form Scheduler and a resource manager derived from ResManager or none.
- Also the implementation for real-time multiprocessor kernel is available.

RtLib – RTKernel (2)

- Method **`dispatch()`** compares currently executing task with the first in the ready queue. If they are different, it forces a context switch.
  - The corresponding **`schedule()`** and **`deschedule()`** functions of the two tasks are called.
- Function **`onArrival()`** is invoked from the task *onArrival* function, which in turn is invoked when a task arrival event is *triggered*. It inserts the task in the ready queue and calls **`dispatch()`**;
- Function **`onEnd()`** is invoked from the task *onEnd* function, which in turn is invoked when a task completes the execution of the current instance.
  - It removes the task from the ready queue, set current executing pointer to NULL and calls **`dispatch()`**.

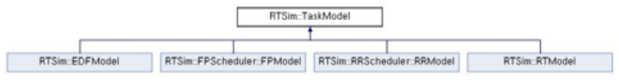
RtLib – Scheduler and TaskModel

- Scheduler is an abstract class and cannot be instantiated. This class models a generic real-time scheduler and it implements the Scheduler interface.
- Basically, this and the derived classes manage a priority queue in a convenient manner, and offer a clean interface toward the kernel and the resource manager.
- The class keeps internally a repository of all tasks that can be scheduled by this scheduler.



RtLib – Scheduler and TaskModel (2)

- Every time a task is "added" to the scheduler, an appropriate TaskModel object is built, which contains the scheduling paramenters of the task.
  - In this way, we clearly separate the task parameters (like period, deadline, wcet, etc.) that are contained in the task class, from the scheduling parameters that are contained in the TaskModel derived classes.
- Tipically a scheduler contains a queue of task model' s instances. The responsibility of this class is to maintain the queue.
- Class TaskModel contains the scheduling parameters and a pointer to the task. It is used by a scheduler to store the pointer to the task and the set of scheduling parameters.
- Each scheduler has its own task model. So the class inheritance trees of the task models and of the schedulers are similar.



RtLib – Task

- This class models a cyclic task.
- A **cyclic task** is a task that is cyclically activated by a timer (e.g., periodic task) or by an external event (e.g., sporadic or aperiodic task).
- This class models a "*run-to-completion*" semantic. At every activation (also called arrival), an instance of the task is executed.
- The task executes all the instructions in the sequence until the last one, and then the instance is completed (*task end*).
- At the next activation, the task starts executing a new instance, and the instruction pointer is reset to the beginning of the sequence.

RtLib – Task (2)

- When a job arrives (`onArrival()`), the corresponding deadline is set (the class Task has no deadline parameter).
  - It adds deadline event to check deadline misses, with the possibility to abort the simulation in case of deadline miss (depending on the abort parameter in the constructor).
  - In order to create a Task the programmer must provide two essential parameters:
    - Interarrival-time time between consecutive activations. Set it to NULL if you want just one activation.
    - Relative Deadline Used to calculate the absolute deadline, when the task arrives.

RtLib – Task (3)

- Another important feature is to provide pseudo-code to the created task:

```
function insertCode( const string )
```

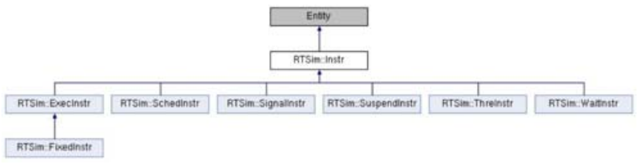
- It parses and inserts instructions into this task. The input string must be a sequence of instructions separated by a semicolon (also for last instruction). The instruction' s types will be described in the related subsection.
- Here is just an intuitive example of code:

```
t1.insertCode("fixed(4);wait(Res1);delay(unif(4,10));signal(Res1);delay(unif(10,20));");
```

  - In this case, the task performs 5 instructions;
    - the first one lasts 4 ticks;
    - the second one is a wait on resource Res1;
    - the third one has variable execution time, uniformely distributed between 4 and 10 ticks;
    - the fourth one is a signal on resource Res1.
    - the last instruction has variable execution time uniformely distributed between 10 and 20 ticks

RtLib – Instruction

- The base class for every pseudo instruction, that represents the code that a task executes.
- An instruction is identified by an execution time (possibly random) and by a certain optional functionality.
- A task contains a list of instructions, that are executed in sequence.



RtLib – TaskStat

- Using statistical modules provided by MetaSim (i.e., *StatMean*, *StatMax*, *StatMean*) it is possible to create custom statistics related to task.
- Task statistics already implemented are:
  - **PreemptionStat< Measure >** Computes the preemption count for each job
  - **GlobalPreemptionStat** Computes the total number of preemptions.
  - **FinishingTimeStat< Measure >** Computes the finishing time of each job.
  - **LatenessStat< Measure >** Computes the lateness of each job of the task attached.
  - **TardinessStat< Measure >** Computes the finishing time normalized by the relative deadline.
  - **UtilizationStat< Measure >** Computes the utilization of each job.
  - **MissPercentage** Computes the miss percentage.
  - **MissCount** Computes the number of deadline misses (single task or the entire task set).
  - **ConsumedPower< Measure >**
  - **SavedPower< Measure >**
- Some of them are created depending on the template class (Measure) that is passed to it, in order to allow programmers to collect the *mean*, *max*, *min* of the desired parameters within all the jobs of all the tasks that are attached to that class.

INSTITUTE OF COMMUNICATION, INFORMATION AND PERCEPTION TECHNOLOGIES

Scuola Superiore Sant'Anna

Retis Real-Time Systems Laboratory

EXAMPLES

MetaSim Examples

- Few examples are provided for:
  - Queue
  - Markov
  - Ethernet

MetaSim Examples

- Few examples are provided for:
  - Queue
  - Markov
  - Ethernet

RTLib Examples

- Examples are provided for:
  - CBS
  - EDF
  - GRUB
  - RM
  - …

INSTITUTE OF COMMUNICATION, INFORMATION AND PERCEPTION TECHNOLOGIES Scuola Superiore Sant'Anna

Retis Real-Time Systems Laboratory

PROJECT PROPOSALS

Available projects

- Design and develop new algorithms for RTSIM
  - Compare different solutions to handle resources reservation under resource sharing
  - Implements semi-partitioned multiprocessor scheduling algorithms

Mauro Marinoni - m.marinoni@sssup.it

thank you!

http://retis.sssup.it/people/nino