Simpatica

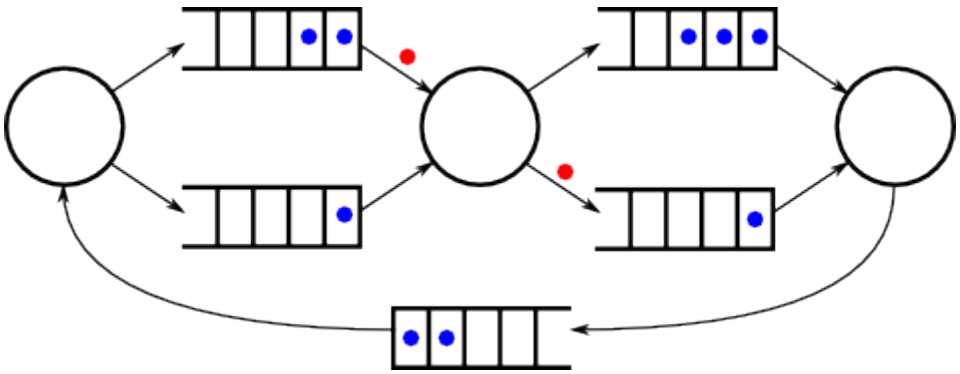
[Portuguese version]

SIMPATICA is a small library for building discrete event simulations. Its main features are:

- Written in ANSI C.
- Its internal structure is simple and easy to understand, allowing its use in the classroom.
- Extensively tested in Linux environments (32 and 64 bits).
- Based on the actors/messages paradigm.
- Conceived aiming a low memory footprint; this allows to build large simulations, with thousands of active entities (we built models with up to 150,000 simultaneous tasks in a computer with 4 Gb of RAM).
- Excellent performance, as it implements its own user-level threads, which do not depend on the operating system scheduling.
- The scheduler queue is implemented using a binary heap, allowing it to achieve a good performance in large-scale simulations.
- Open source software (GNU GPL License)

The SIMPATICA library allows to build discrete event simulations based on the actors/messages paradigm. According to this paradigm, a simulation model is composed by a set of actors or tasks that communicate among them using messages. This library implements three kinds of entities:

- **Task**: is an active entity, whose behavior is defined by a C function (exactly like a thread body). Each task has an unique identifier in the simulation. Tasks can produce and consume messages, which are transferred between tasks through queues.
- **Queue**: is a message repository, in which messages are stored according the arrival date (FIFO ordering). Tasks can put messages on the queues and get messages from them. Tasks and queues are independent: any task can get/put messages in/from any queue.
- **Message**: is a C structure (`struct`) whose content is entirely defined by the model builder. This provides modeling flexibility. The library keeps an unique ID for each message, among other information (see the `msg_attr` function below).



Files

- Version 0.7

Interface

This library provides a set of ANSI C functions to build simulation models. By convention, most functions abort the simulation execution if an error occurs, printing an error message in `stderr` and returning an 1 status to

the operating system. This radical approach was adopted to minimize the risk of undetected errors that can interfere in the simulation results. For this same reason, most functions have no return value (they return `void`).

```
void init_simulation (int maxTasks, int maxQueues)
```

Initializes the internal structures needed for each simulation. This function should be called only once, at the beginning of C program containing the model.

The `maxTasks` and `maxQueues` parameters define respectively the maximum number of tasks and queues that will be created by the model (they are used to allocate the memory used by those structures).

```
void run_simulation (double maxTime)
```

Runs the simulation until the simulation clock achieves the `maxTime` value. It should be called after the model is defined.

This function can be called several times, to make a simulation to progress in time steps. The simulation clock will then advance from its last value to the `maxTime` value:

```
... // initialize and create tasks/queues
run_simulation (1000) ;  // runs the simulation from t=0 to t=1000
... // process partial results
run_simulation (2000) ;  // continues the simulation until t=2000
... // process partial results
```

```
void kill_simulation ()
```

Finishes a simulation, cleaning all data structures allocated to it (tasks, queues, and messages). This call is used to restart the library, allowing to run several simulations in the same C program, sequentially.

```
void trace_interval (double startTime, double stopTime)
```

Enables trace messages when the simulation time is in the interval [`startTime`, `stopTime`]. Trace messages inform about the events processed by the library, and are sent to the standard output (`stdout`).

```
uint task_create (void (*taskBody)(void *), void* startArg, int stackPages)
```

Creates a new task. The `taskBody` parameter indicates a function that defines the task behavior; this function will receive the `startArg` parameter when starting its execution. Each task is identified in the simulation by a positive integer ID, returned from this call.

The `stackPages` parameter indicates the number of memory pages to allocate for this task's stack. Its value should be greater than zero. Memory pages are usually 4,096 bytes long, but this depends on the system.

It is important to say that stacks too big consume more memory, and this lowers the maximum number of entities the simulation can handle. On the other hand, stacks too small can lead to memory access errors, sometimes undetected. The stack size depends on the task behavior: the total size of local variables, functions called by the task, etc. Simple models generally work well with stacks having 1-3 pages, but this should be evaluated in each case.

PS: the `printf`/`sprintf`/`fprintf` functions can use a lot of stack space, so tasks that use them should have bigger stacks. As the tracing facility uses such functions, stack sizes should be bigger if such facility is to be used in a simulation.

Return value: the ID of the task just created. IDs are positive integers allocated in sequence (1, 2, 3, ...). The

scheduler itself has ID 0.

```
void task_exit ()
```

Exits the current/active task (the task which is running when this function is called), freeing its resources. This function should be called when the task finishes its execution.

```
void task_destroy (int task_id)
```

Destroys the task indicated as a parameter, freeing its resources.

```
void task_sleep (double t)
```

Makes the current task to sleep during `t` simulation time units. In the meanwhile, other tasks can run.

```
void task_passivate ()
```

Makes the current task to sleep indefinitely, until being awaken by another task through the `task_activate` call (see below).

```
void task_activate (int task_id, double waitTime)
```

Awakes the indicated task after `waitTime` simulation time units, counting from the current simulation time. The `waitTime` parameter can be set to zero, to awake that task now. This call only awakes tasks that are sleeping after a `task_passivate` call, having no effects on other tasks.

```
int task_id ()
```

Retrieves the unique ID of the current task, which is a positive integer (1, 2, 3, ...). If called within the scheduler (outside of a task) returns 0.

```
double time_now ()
```

Retrieves the current value of the simulation time.

```
int queue_create (int capacity, int policy)
```

Creates a new message queue with the given capacity (maximum number of massages the queue can hold) and ordering policy. In the current version, both parameters are ignored: messages are ordered by they arrival date and queue maximum size is only limited by the available memory.

```
int queue_destroy (int queue_id)
```

Destroys the queue indicated, deleting all the messages it contains and also the data structures that implement it.

```
void queue_stats (uint queue_id, uint *size, uint *max,
                  double *mean, double *var, ulong *put, ulong *got)
```

Retrieves the following information about the indicated queue, calculate from t=0:

- `size`: number of messages currently in the queue
- `max`: maximum observed queue size
- `mean`: mean size of the queue
- `var`: variance of the queue size

- `put`: number of messages put in the queue (using msg_put)
- `got`: number of messages got from the queue (using msg_get)

For each field, it should be given the address (pointer) of a variable that will receive the corresponding value, or NULL (0) to ignore it. For instance, the following call retrieves the current number of messages in the queue `q1`:

```
queue_stats (q1, &size, 0, 0, 0, 0, 0) ;
```

```
void* msg_create (short size)
```

Creates a new message with the given `size`. Each message create receives an unique ID in the simulation. The message size is user-defined, so this call returns only a void pointer to a memory area, to be cast to the user-defined type. Usually messages are defined as structs containing the fields necessary to the simulation model. Message size can be zero, in the case no user-defined fields are needed. Internally, the library maintains some tags for each message, that can be retrieved using the `msg_attr` call (see below).

Return value: a void pointer to the new message.

```
void msg_destroy (void *msg)
```

Destroys the message indicated as parameter. All messages should be destroyed after their usage, for freeing memory to other needs. This is particularly important for large and/or long simulations.

```
void msg_put (int queue_id, void* msg)
```

Puts the indicated message at the end of the indicated queue (appends it).

```
void* msg_get (void *msg)
```

Removes the given message from the queue where it currently is, returning a pointer to it (the same pointer informed as parameter, its value does not change).

```
void* msg_wait (int queue_id, double timeOut)
```

Waits for a message in the queue indicated as parameter. The current task is suspended until receiving a message or a time-out. The `timeOut` value is the maximum amount of time to wait for the message, and can be set to `INFINITY` for indefinite waiting. It returns a pointer to the received message, or NULL, in the case of a time-out. Important: the message received is not automatically removed from the queue.

```
void* msg_first (int queue_id)
void* msg_last (int queue_id)
void* msg_prev (void* msg)
void* msg_next (void* msg)
```

Allows to “navigate” in a given queue. They return pointers to messages ,or NULL, in the case the requested message does not exist.

```
void msg_attr (void *msg, long *id, double *birth, double *sent,
               long *creator, long *sender, int *queue)
```

Retrieve the following information about the given message:

- `id`: unique identifier for the message (ID)
- `birth`: message creation time
- `sent`: time when the message was last sent
- `creator`: creator task's ID

- `sender`: last sender task's ID
- `queue`: current message queue ID, or 0, if the message is not in a queue

For each field, it should be given the address (pointer) of a variable that will receive the corresponding value, or NULL (0) to ignore it (see `queue_stats`).

Usage

Using this library is quite simple: one needs to write a C program using the library functions to define the simulation model and the simulation parameters. Then one should compile and link it with the library to get an executable file that implements the simulation:

```
$ cc simpatica.c model.c
$ a.out
```

The file `simpatica.c` contains the simulation library code. The file `model.c` contains the simulation model itself.

Simulation example

Hereafter is presented a model in which 1,000 “source” tasks continuously send messages to a queue “q1”. a “sink” task removes the messages from q1 and calculates the time spent between the message's birth and its death (removal from the queue and destruction). When the simulation finishes, the program calculates the mean of messages' lifetimes. It also presents the mean queue size and its variance.

The source code for the model is (to be saved as file `model.c`):

model.c

```
#include <stdio.h>
#include <stdlib.h>
#include "simpatica.h"

// queue descriptor
int q1 ;

// variables to calculate mean of messages' lifetimes
long num_msgs = 0 ;
double sum_times = 0.0 ;

// messages are structs with user-defined fields
typedef struct msg_t
{
   int value ;  // a random value, just to give an example
} msg_t ;

// source tasks's body
void sourceBody (void *arg)
{
   msg_t *msg ;

   for (;;)
```

```
   {
      // creates a new message
      msg = (msg_t*) msg_create (sizeof (msg_t)) ;

      // fills the message with a random value
      msg->value = random ();

      // puts the message in q1 queue
      msg_put (q1, msg) ;

      // sleeps for a random amount of time
      task_sleep (15 + random() % 5) ;
   }
}

// sink task's body
void sinkBody (void *arg)
{
   msg_t *msg ;
   double creation_time ;

   for (;;)
   {
      // waits for a message and removes it from q1 queue
      msg = (msg_t*) msg_get (msg_wait (q1, INFINITY)) ;

      // gets the message creation date
      msg_attr (msg, 0, &creation_time, 0, 0, 0, 0) ;

      // simulation time elapsed for processing the message
      task_sleep (1) ;

      // accumulate times
      sum_times += (time_now() - creation_time) ;
      num_msgs ++ ;

      // destroys the message to free its resources
      msg_destroy (msg) ;
   }
}

int main ()
{
   int i ;
   double mean, variance ;

   // prepares a simulation for 1001 tasks and one queue
   init_simulation (1001,1) ;

   // creates 1000 "source" tasks
   for (i=0; i< 1000; i++)
      task_create (sourceBody, NULL, 2) ;

   // creates one "sink" task
   task_create (sinkBody, NULL, 2) ;
```

```
    // creates the q1 queue
    q1 = queue_create (0, 0) ;

    // executes the simulation until t=50000 time units
    run_simulation (50000) ;

    // print results
    printf ("Mean time between msg production/consumption: %0.3f\n",
            sum_times / num_msgs) ;

    // prints mean queue size and its variance
    queue_stats (q1, 0, 0, &mean, &variance, 0, 0) ;
    printf ("Queue size: mean %0.3f, variance %0.3f\n", mean, variance) ;

    // frees simulation resources
    kill_simulation () ;

    exit(0) ;
} ;
```

Model compilation is done through the following command line:

```
$ cc simpatica.c model.c
```

The model execution generates the following output:

```
$ ./a.out
-- Simulation initialized, Simpatica version 0.7, 06/out/2007 (mem: 8Kb)
-- Simulation in interval t=[0.000, 50000.000), 1001 tasks
-- Simulation time:  5000.000,     299593 events,  10% done in     0 secs (mem:
22005Kb)
-- Simulation time: 10000.000,     598584 events,  20% done in     1 secs (mem:
35552Kb)
-- Simulation time: 15000.000,     897715 events,  30% done in     2 secs (mem:
49105Kb)
-- Simulation time: 20000.000,    1196897 events,  40% done in     2 secs (mem:
62660Kb)
-- Simulation time: 25000.000,    1496007 events,  50% done in     3 secs (mem:
76212Kb)
-- Simulation time: 30000.000,    1795129 events,  60% done in     3 secs (mem:
89765Kb)
-- Simulation time: 35000.000,    2094218 events,  70% done in     4 secs (mem:
103316Kb)
-- Simulation time: 40000.000,    2393259 events,  80% done in     5 secs (mem:
116864Kb)
-- Simulation time: 45000.000,    2692314 events,  90% done in     5 secs (mem:
130414Kb)
-- Simulation time: 50000.000,    2991445 events, 100% done in     6 secs (mem:
143967Kb)
-- Simulation completed in 6 seconds (mem: 143967Kb)
Mean time between msg production/consumption: 24583.696
Queue size: mean 1445994.474, variance 696500121941.198
-- Simulation killed (mem: 0Kb)
```

In the example above, lines beginning by "--" show running messages automatically generated by the library.

All such messages are sent to the standard error output (`stderr`) and can be intercepted using shell redirection. Furthermore, a compilation option allows to inhibit the generation of such messages (see below).

Compilation options

There are some compilation options that allow to activate/deactivate additional tests and/or messages. Such options are mainly useful for debugging the library itself, not for model debugging.

- HEAPCHECK: periodically verifies the integrity of the scheduler.
- HEAPDEBUG: generates screen messages about scheduler operations.
- STACKCHECK: periodically verifies the current task's stack integrity.
- STACKDEBUG: generates messages about stack usage by the current task.
- NOTESTS: inhibits all consistency tests. It can be used to improve the simulation performance, once the model is fully tested and free of errors.
- QUIET: inhibits the generation of library messages on `stderr`.

These options can be used as follows:

```
$ cc -DDEBUGHEAP simpatica.c model.c
```

Concurrency

In systems containing several tasks executing simultaneously, there is a risk of concurrent tasks to produce errors or inconsistent results. In this library, tasks are implemented as lightweight cooperative user-level threads. In such implementation model, a thread only looses processor control to another thread in specific code locations, when strictly needed.

In this library, a thread execution is suspended only when it requests an operation that can have an impact on the simulation time: `task_sleep` and `msg_wait`. All other operations are unblocking and thus are free from concurrency risks. From the viewpoint of any task, the contents of global variables and queues only can change during the execution of the `task_sleep` or `msg_wait` calls, and never in other locations of its code.

From:
https://wiki.inf.ufpr.br/maziero/ - **Prof. Carlos Maziero**

Permanent link:
https://wiki.inf.ufpr.br/maziero/doku.php?id=software:simulation

Last update: **2020/08/18 19:22**