University of West Bohemia in Pilsen
Department of Computer Science and Engineering
Univerzitni 8
30614 Pilsen
Czech Republic

C-Sim version 5.1

Roman Jokl, Stanislav Racek

Technical Report No. DCSE/TR-2003-17
May, 2003

Distribution: public

Technical Report No. DCSE/TR-2003-17
May 2003

C-Sim version 5.1

Roman Jokl, Stanislav Racek

Abstract

C-Sim is a programming tool for simulation of discrete processes using a method of pseudo–parallel processes. It is an extension of ANSI C language obtained by including a library of SIMULA-like types and functions packed within several program modules. The typical application area of **C-Sim** is functional validation of distributed, parallel and fault-tolerant systems and programs. **C-Sim** version 5.1 offers two possibilities how to implement pseudo-parallel processes. The first one is based on setjump()/lomgjump() functions utilizations, the second one uses POSIX threads as an implementation basis of pseudo-parallel activities.

This work was supported by the Ministry of Education of the Czech Republic, project no. MSM-235200005 - Information systems and Technologies..

Copies of this report are available on
http://www.kiv.zcu.cz/publications/

University of West Bohemia in Pilsen
Department of Computer Science and Engineering
Univerzitni 8
30614 Pilsen
Czech Republic

Contents

Chapter 1

Introduction

C-Sim is a program enhancement of the **C** language used for creating discrete simulation models based on the method of pseudo-parallel processes. It has the form of a library of basic object types and operations on them, which allows to enhance the standard object types with new attributes and methods to fulfill the needs of a concrete model.

The idea of **C-Sim** was taken from the programming language **SIMULA** and the library provides SIMULA-like resources from the system classes `SIMSET` and `SIMULATION`. The **C** language was chosen for its portability among different systems.[1]

*This document is reference manual of **C-Sim** version 5.1.*

Improvements that were made compared with the previous version 4.1:

- better readability and safety of source code,
- an additional implementation of POSIX threads based mechanism of pseudo–parallel processes and modification of the **C-Sim** kernel in order to allow the use of both mechanisms of simulation processes implementation,
- better modular structuring of **C-Sim** kernel,
- addition of new features like *semaphore module*, *message passing module*, etc.

Here we need to explain the terminology we will use in this document. All definitions will be as simple and intuitive as possible.

Real object defines a particular part of the real world, a part that is more or less constrained. A real object has real properties which are also called attributes. The remaining part of the real world is called the *neighborhood*.

Model is a purposefully constructed image of a real object. In the process of construction we usually apply: simplification – we consider only the properties which are significant for the model construction. generalization – we do not describe one particular real object, instead we describe a class of real objects with common attributes.

[1] **C-Sim** is implemented using the **ANSI C** specification.

System as we understand the term, is a purposefully defined set of items with certain relations. i.e. $S = (P, R)$, where P is a set of items and R is a set of relations between the items, representing the mutual functional relations between the items and the neighborhood of the system.

Simulation is a research method which replaces the investigated system with a simulation model upon which we perform further experiments to obtain information about the originally investigated system.

System can be divided by various aspects:

- Dynamic versus Static – whether some properties of the items have the character of memory or not. All memory-character properties describe the state of the system.
- Continuous versus Discrete – whether the time-dependent properties do change in continuous or discrete set of points on the time axis.
- Closed versus Open – whether we consider the interaction with neighborhood or not.
- Stochastic versus Deterministic – whether some properties of the items do have the character of random numbers or not.

C-Sim is designed for simulation of dynamic discrete systems (both open and closed). It is often used in simulations of stochastic systems, but may be used as well in simulations of deterministic systems.

The most common methods of real discrete system model decomposition, divided by the controlling algorithm of simulation computation, are called

- Method of event interpretation – all event that should occur in the future model evolution are stored in a list called *event calendar*. The list is sorted in ascending order by the value of event realization model time. The controlling algorithm of the simulation computation interprets the events in the calendar sequentially. If an event is the cause for another event creation, the scheduling of the new event into the calendar is part of the first event interpretation. An interpreted event is removed from the calendar.
- Method of pseudo–parallel processes – single computation parts are encapsulated in selected object types, which gain therefore an own program and the character of standalone computation processes. The activity of a process is divided into a sequence of activity phases, that are executed each at a single point in model time. To control the pseudo–parallel computation of all processes in the model again requires the existence of a calendar-like data structure. The calendar contains processes sorted in ascending order by model time – their record includes a reference into the program. The control loop of the computation executes the processes sequentially, in the order given by the calendar.

C-Sim is based on method of pseudo–parallel processes.

Every simulation uses its own time, so called *simulation time*. The changes in simulation model of a system take place at discrete points in simulation time. A discrete simulation

is divided into *simulation steps*, in every step a part of the program assigned to the first process in the calendar is executed. The value of simulation time remains constant during a step and may change jump-like between steps. The values of simulation time must form a non-descending sequence.

After the process executes the part of its program which corresponds to the currently scheduled event, it is removed from the calendar. The scheduling of a new event may be a part of the process activity, i.e. it is possible to re-insert the process into the calendar. The processes' active phases are interleaved, with any number of simulation steps between two parts of the same process. The principle of processes interleaving is depicted in figure 1.1.

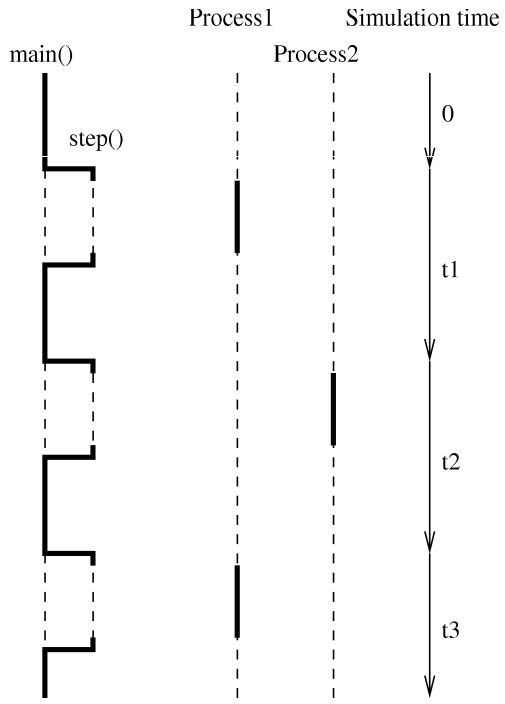

Figure 1.1: Process interleaving in the method of pseudo–parallel processes

Chapter 2

Principles Used

Like **SIMULA**, the **C-Sim** needs the description of behavior of the simulated system in the form of processes. To model objects of the real world with program objects in the process form is straightforward and natural. The **C** language, unlike **SIMULA**, has no tools for object-oriented programming but some of the basic principles can be implemented using predefined macro commands. This concerns above all the possibility of inheritance from offered basic object types. To avoid confusion of terms, inheritance will be in connection with the **C-Sim** library replaced by the term *derivation.*

Another argument for choosing the **C** language is the expected non-traditional application of discrete–time modeling (e.g. verification of properties of parallel programs, communication protocols, fault-tolerant systems and algorithms, real-time applications, etc.). Mapping of dynamic components (threads, processes) of the modeled program to the processes of the simulation program and corresponding simple translation of verified program parts from the model to the target implementation code is assumed.

2.1 Internal Structure of the Library

The **C-Sim** library is divided to three layers as seen on the picture 2.1.

The lowest layer is defined by files csim_kr.h, csim_dt.h and csim_tm.h. The files csim_dt.h and csim_tm.h contain the definitions of data types (e.g. `CSIM_BOOLEAN`, `CSIM_BYTE`, ...) described in detail in section 3.1 and type `CSIM_TIME` described in section 2.6. These data types are used globally by the whole **C-Sim** library and thus their definitions are included in the file csim.h.

Furthermore this layer also implements pseudo-parallel processes (file csim_kr.h). The file csim_kr.h was created in two versions. The directory kr_jmp contains version based on *long jumps*, the directory kr_thr contains version based on POSIX threads. Library user may choose desired implementation at compile time by defining the macro `KR_PATH` on the compiler command line. Possible values are `kr_thr/csim_kr.h` and `kr_jmp/csim_kr.h`, being `kr_jmp/csim_kr.h` as the default. File csim_kr.h is included into csim.h in the following way:

```
#ifndef KR_PATH
#define KR_PATH "kr_jmp/csim_kr.h"
#endif
#include KR_PATH
```

The second layer is defined by files csim.c and csim.h, which contain definitions of data types, macros and functions that enable to create a simulation model and to control the simulation flow. The defined types are described in section 3, macros and functions in chapter 4.

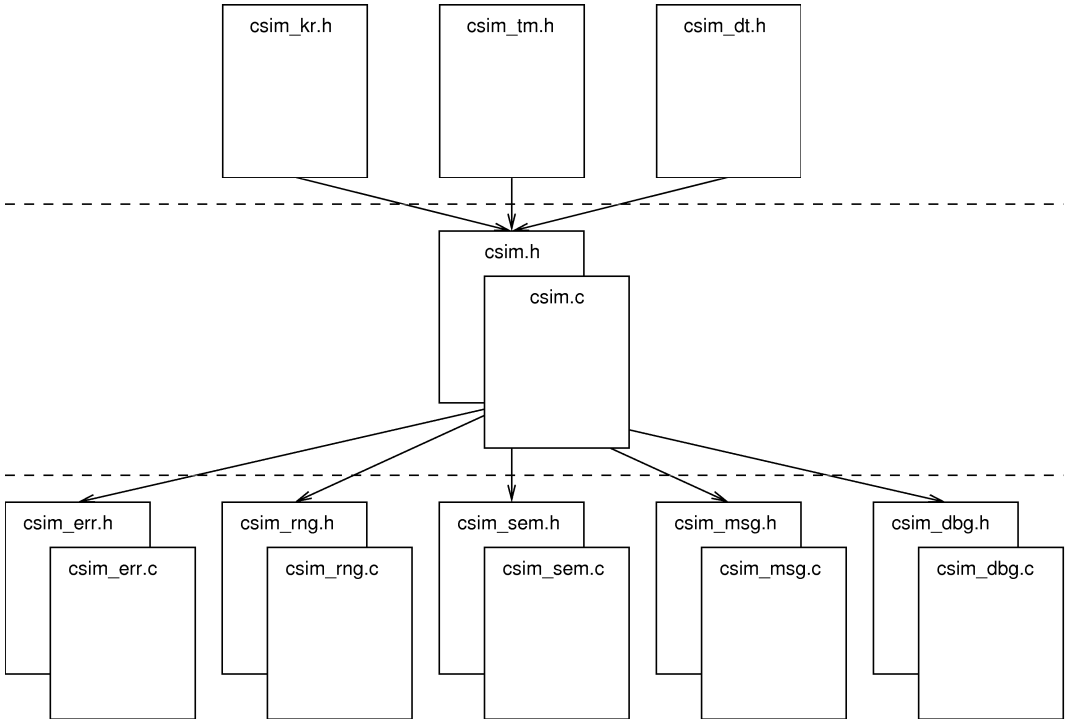

Figure 2.1: **C-Sim** layers

Third layer contains optional modules:

- random number generator – files csim_rng.c and csim_rng.h
- error code to error message conversion – files csim_err.c and csim_err.h
- console debug tools – files csim_dbg.c and csim_dbg.h
- message passing – files csim_msg.c and csim_msg.h
- semaphore – files csim_sem.c and csim_sem.h

These modules are described in section 5.

In the archive with the file Makefile is also provided as an example how to compile the library and to build applications.

2.2 Supported Object Types and User Types Derivation

As it was mentioned before, the pattern for the library was the **SIMULA** language. Basic supported types, that this library provides, are taken from the original object classes in **SIMULA**. These are the types `CSIM_LINK`, `CSIM_HEAD` and `CSIM_PROCESS`. The objects of this type are in **C-Sim** created in a dynamically allocated memory of the simulation program (heap), thus the `CSIM_DYN_MEM` object type was created as an ancestor from which the above mentioned types are derived. In contrast to **SIMULA**, the type `CSIM_HEAD` is in **C-Sim** derived from `CSIM_LINK`, in addition to type `CSIM_PROCESS`. Hence it is possible to create list of list heads (i.e. list of lists). The user may thus obtain a tool for creating complicated structures of objects. The hierarchy of **C-Sim** supported object types is depicted on figure 2.2.

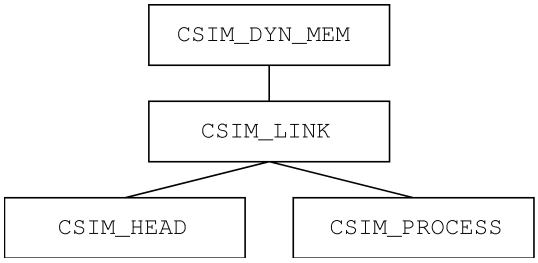

Figure 2.2: Object types hierarchy in **C-Sim**

The **C-Sim** library provides a possibility to extend the standard object types by own data items (in the object–oriented programming the *attributes*). A derivation fro the supported basic types is implemented with the aid of macros. Two groups of macros for deriving new object types were introduced in the **C-Sim** library.

The first group is preserved from the previous version of the library (i.e. **C-Sim** 3.0) and includes these macros: `csim_d_dyn_mem`, `csim_d_link`, `csim_d_head` and `csim_d_process`. The use of these macros is simple and straightforward, as it can be seen from the following example.

```
typedef struct my_dyn_mem {
  csim_d_dyn_mem;
    ...
  // user defined attributes
  struct my_dyn_mem  *p_myself;
    ...
} MY_DYN_MEM;
```

We derived a new type from the type `CSIM_DYN_MEM` by inserting the `csim_d_dyn_mem` macro into the `MY_DYN_MEM` type declaration. Macro `csim_d_dyn_mem` is derived in such a way, that it inserts into the definition of the derived type the attributes, which need to be inherited from the parent type. The macro is defined as follows:

```
#define csim_d_dyn_mem                \
  char                  *name;        \
  char                  type;         \
  char                  *check;       \
  struct csim_dyn_mem   *p_head_mem;  \
  struct csim_dyn_mem   *p_suc_mem;   \
  struct csim_dyn_mem   *p_pred_mem;  \
  CSIM_DESTRUCTOR       destructor;   \
  CSIM_VIEW             view
```

After the processing by C-language preprocessor the type `MY_DYN_MEM` definition[1] has the following form:

```
typedef struct my_dyn_mem {
  char                  *name;
  char                  type;
  char                  *check;
  struct csim_dyn_mem   *p_head_mem;
  struct csim_dyn_mem   *p_suc_mem;
  struct csim_dyn_mem   *p_pred_mem;
  CSIM_DESTRUCTOR       destructor;
  CSIM_VIEW             view

  // user defined attributes
  struct my_dyn_mem  *p_myself;
    ...
} MY_DYN_MEM;
```

The type `MY_DYN_MEM` now contains the same attributes as `CSIM_DYN_MEM`, and in addition some user defined attributes. For this type it is possible to use the same operations as defined for `CSIM_DYN_MEM` type. The example shows a type declaration with two names (`my_dyn_mem`, `MY_DYN_MEM`), which allows the structure declaration, whose individual attribute has the same type as the structure itself.

The second group of macros has the same purpose as the first one, but it uses another syntactic form. We will again show an illustrative example.

```
csim_derived_link(my_link)
    ...
  // user-defined attributes, e.g.
  struct my_link *p_myself;
```

[1] It is a user-defined data type in **C** language. With this type are logically (i.e. not with an *encapsulation construction*) connected operations, which are executable on it and implemented as "normal" **C** functions. Therefore we will use the concept *object type* for better understandability, even if it does not fulfill some characteristics, that are required in the object-oriented paradigm (*explicit encapsulation, method polymorphism, etc.*).

```
    ...
csim_end_derived(MY_LINK);
```

This example is expanded with **C** preprocessor to the same code as the previous example. From this it follows, that the parameters of the opening and closing program bracket of type declaration are the first and the second name of the declared structure. Therefore according to the **C**-convention we introduce for both identifiers the same names differentiated only with case of letters. If we want to use a pointer of the same type as the structure itself, it is again necessary to use the first name with the keyword `struct`. The identifiers of all macros in this group are `csim_derived_dyn_mem`, `csim_derived_link`, `csim_derived_head`, `csim_derived_process` and the closing bracket is `csim_end_derived`. Parameter is always the name of the new type. For the opening bracket it is the first name (lower case), the closing bracket use the second name, that is then used in variable definition of the introduced type. Detailed definitions of the individual macros are present in the header file csim.h.

2.3 Implementation of Dynamic Memory Management

The simulation tasks have specific requirements on the method of *dynamic memory allocation.* Interactively controlled simulation run can be any time interrupted for repeated run and then started again after the change of its parameters. The memory allocated during the first simulation run must be released properly. If the library user was given the possibility of an uncontrolled manipulation with the dynamic memory, a simple repetition of the simulation run would not be possible. Therefore we will further mention and explain the accepted rules for the use of the dynamic memory in **C-Sim** programs.

- There was introduced the `CSIM_DYN_MEM` type, from which all basic types are derived. This type contains the items, which make it possible to insert the objects of this type and of the derived types into the bidirectionally chained list.
- At the moment of interrupt of the simulation run, the list allows to deallocate all objects it holds. From this there follows the restriction that forbids the user to directly allocate the dynamic memory,[2].
- If the user insists on the dynamic memory allocation during the simulation run, he must include the necessary data as items of the new type, which is derived from the type `CSIM_DYN_MEM`.
- New object can be created using the `csim_new_dyn_mem()` function, which inserts the object into the *list of dynamically generated objects.*

As this restriction would force the user to create always a new type, e.g. for dynamic arrays of different lengths, the **C-Sim** library introduces the possibility of defining an *object destructor.* The object destructor is called always when the object derived from the type `CSIM_DYN_MEM` is deallocated from the memory. It also allows to define in the structure of a supported type the

[2]For example, by the use of the function `malloc()`

pointers to dynamically generated items, which are not derived from the `CSIM_DYN_MEM` type and hence they are not inserted into the list of dynamic memory blocks. The type declaration for dynamic arrays of different lengths would then be derived from the type `CSIM_DYN_MEM`, but it would not contain as an item the array of concrete length, but only the pointer to a dynamic array of any length. This array is then allocated by the user himself using e.g. the `malloc()` function, but he is obliged to define for such an object the method of destructor, which guarantees releasing the user-allocated memory after the destruction of the object.

Further we will present the definitions of macros for creation of new object types derived from the basic types provided by the **C-Sim** library and the definition of object destructor.

The macros for creating new dynamic objects:

```
#define csim_new_dyn_mem(TYPE_NAME)                                          \
  (TYPE_NAME *) csim_init_dyn_mem((CSIM_DYN_MEM *) malloc(sizeof(TYPE_NAME)))

#define csim_new_link(TYPE_NAME)                                             \
  (TYPE_NAME *) csim_init_link((CSIM_LINK *) malloc(sizeof(TYPE_NAME)))

#define csim_new_head(TYPE_NAME)                                             \
  (TYPE_NAME *) csim_init_head((CSIM_HEAD *) malloc(sizeof(TYPE_NAME)))

#define csim_new_process(TYPE_NAME, PROC_PROG)                               \
  (TYPE_NAME *) csim_init_process(                                           \
    (CSIM_PROCESS *) malloc(sizeof(TYPE_NAME)), (PROC_PROG))
```

These macros allow to create new objects both of the basic types and of derived types. The attributes of the basic types are immediately initialized, and if the object is of the derived type, only the attributes derived from the basic types are initialized.[3] The `csim_init_...()` functions can be used separately as well.

Functions for deallocation of the objects dynamic memory of basic and derived types:

```
CSIM_RESULT csim_dispose_dyn_mem(CSIM_DYN_MEM *p_dyn_mem);

CSIM_RESULT csim_dispose_link(CSIM_LINK *p_link);

CSIM_RESULT csim_dispose_head(CSIM_HEAD *p_head);

CSIM_RESULT csim_dispose_process(CSIM_PROCESS *p_process);
```

For every object type supported by **C-Sim** a destructor function can be created, which is called when destructing the object. Any explicit call of a destructor within a simulation program is not allowed.

[3]The initialization of the remaining attributes must be done by the user himself. For this purpose, he can naturally use his own initialization function (replacing the *constructor*) of the corresponding type.

The following macros in the form of program brackets are introduced for the definition of destructor function.

```
#define csim_begin_destructor(TYPE_NAME, DESTR_NAME)                    \
void DESTR_NAME(void *p_void)                                           \
{                                                                       \
  TYPE_NAME *p_my = (TYPE_NAME *) p_void;

#define csim_end_destructor                                             \
}
```

Attributes of an object are accessible inside the destructor through the `my.attribute` construction. The identifier `my` is defined as a macro and uses the destructor's parameter `p_void`. It allows to have in the destructor body a straightforward and intuitive access to the attributes of the object, for which the destructor is defined. Destructor is attached to the object by the macro `set_destructor()`.

```
#define csim_set_destructor(P_DYN_MEM, DESTRUCTOR)                      \
  (P_DYN_MEM)->destructor = (DESTRUCTOR)
```

The type for object destructor is introduced in the following way.

```
typedef void ((*CSIM_DESTRUCTOR) (void *p_void));
```

A survey of dynamic memory management

- All objects of basic types and types which are derived from them must be created dynamically using the predefined macros. All memory allocated by the user must be a part of types derived from the basic type `CSIM_DYN_MEM`. Systematic use of this type allows to maintain the list of all objects and to cancel them in case of repeated simulation run.
- If the user intends to use his own dynamic variables, he can define them as pointer–typed attributes of a new type derived from `CSIM_DYN_MEM`. If an object type contains pointers to dynamic objects derived from `CSIM_DYN_MEM` the user must not create a destructor for them.
- The second possibility is to use pointer-typed attributes representing dynamically created arrays (e.g. allocated by the function `malloc()`). In this case the user must create for such object type a destructor function that will release the memory, to which the attributes refer.
- Explicit destruction of the process record of an active process is not allowed. All objects derived from `CSIM_DYN_MEM` can be disposed simultaneously (typically at the end of every experiment) by calling the `csim_clear_mem()` function.

2.4 Implementation of Pseudo-parallel Processes

The used method of discrete simulation is based on the use of *pseudo-parallel processes*. The process programs are also called *cooperating routines*, shortly *coroutines*. Coroutines are functions, which do transfer control between themselves, but do not explicitly state any superiority of the caller over the called one. The called coroutine continues its execution from the point where it ended the last time it transferred control to another coroutine.

The execution of a program, that comprises two coroutines, goes as follows:

1. the first coroutine starts,
2. the first coroutine does part of the computation and then transfers control to the second coroutine,
3. the second coroutine does part of the computation and then transfers control back to the first coroutine,
4. the first coroutine continues with the computation from the point where it previously ended, then again transfers control to the second coroutine,
5. the computation finishes once one of the coroutines ends.

The point of program, where a coroutine transfers control to another one and where it continues once it retains the control is called *reactivation point*.

A simulation based on parallel processes method is built up of simulation steps. In each step a part of the computation is executed exactly between two reactivation points of one coroutine. A simulation program thus contains a main loop, in which the function `csim_step()` that processes one simulation step is repeatedly called. The main loop itself is a coroutine and this coroutine creates and destroys all the other coroutines as needed by the library function calls.

One simulation step runs the following algorithm:

1. Main loop executes a process that is placed at the first place in the calendar and stops itself. In case the process has been already executed before, main loop transfers control to it using the macro `csim_switch_to_process()`, otherwise it creates and executes new process using the macro `csim_create_process()`.
2. Once the process reaches a reactivation point, it transfers control back to main loop using the macro `csim_switch_to_step()` and stops itself.
3. If the process reaches its end defined by the macro bracket `csim_end_program`, it transfers control to main loop using the macro `csim_return_to_step()` and stops itself.

The processes are running in so-called model time, which in contrast to the real time, can be arbitrarily slowed down or accelerated.

The "life" of processes in model time has a discrete character—*active phases* of the process are performed in a discrete value of the model time (i.e. they have a zero duration in model time, execution of one active phase of one process is realized in one simulation step) and between these phases there are *phases of inactivity* with non-zero duration in model time.

Basic distinguished states of the processes are:

Passive – the process is not planned (the process has no record in the calendar).

Planned – the time of the next activation is determined (the process has a record in the calendar).

Active – the process is within some phase of activity (record of the process is the first one in the calendar).

Terminated – program of the process has ended (and cannot be run again), data of the process is still accessible.

The transition from one state to another occurs after one of the following operations has been performed in active process (these are the operations provided for by the **C-Sim** library):

`passivate()` – active process changes its state into **passive** (it is removed from the calendar and the next process form the top of calendar is activated).

`hold()` – active process changes to state **planned** and the next process from the calendar is activated. The length of the next inactivity phase is a parameter of this operation.

`activate()` – active process x changes the state of other process y from state **passive** to **planned**. The active process x continues its activity as **active**. The parameters of this operation are the reference to y and the time of activation.

`cancel()` – active process x changes the state of another process y from state **planned** to **passive**. The active process x remains active. Parameter of this operation is the reference to y.

When implementing pseudo-parallel processes in **C-Sim**, which behave according to the above description, next steps must be performed:

- A type `CSIM_PROCESS` for objects with their own "life" needs to be created.
- A possibility to create user-derived type should be given (see above).
- A possibility to define a program (i.e., description of a "life") of an instance of `CSIM_PROCESS` type should be offered to the user.
- A possibility to interleave the "life" of all active objects (always with continuation from the point of last termination of the run – reactivation point) needs to be implemented.

The first point is fulfilled by defining the type `CSIM_PROCESS` that describes *process data record.*

The third point was solved using the macros `csim_program`, `csim_end_program` and `csim_end_process` defined in the file csim.h. Different implementation methods require somewhat different fields in the `CSIM_PROCESS` structure. These "dependent" fields are added also in a way of derivation using the macro `csim_process_kr` (the second point).[4]

The fourth point is fulfilled by defining macros `csim_create_process()` and `csim_cancel_process()` for process creation and destruction; `csim_switch_to_process`, `csim_switch_to_step()` and `csim_return_to_step()` for process control switching. These macros are used to implement functions `passivate`, ..., etc., so the library user needn't to use them directly.

2.4.1 Implementation Based on Long Jumps

One method of implementation is based on the functions `setjmp()` and `longjmp()`, which are a part of the standard **C**-language libraries on different computers and operating systems. These functions allow storing the *process context* (contents of processor's registers) into the predefined structure and later return to the stored process status. They also provide for switching the process context without using the nonstandard operations, what improves the portability, security and readability of the library code.

The function `setjmp()` stores the current context into the structure of the type `jmp_buf`. Function `longjmp()` then uses this structure as a parameter and restores the contents of the registers to the initial values. This will return the program run back to reactivation point.

- At the first call of `setjmp()` function the structure containing the current context is set.
- When the `longjmp()` function is then called, the program returns control back to the reactivation point, which was the first parameter, and function `setjmp()` returns the value used as the second parameter of the `longjmp()` function. By this value it is then decided, whether this is a return or a new run, which must set new reactivation point.
- The `setjmp()` function has thus one parameter, which is a structure of the `jmp_buf` type, where the context information is stored. It also returns the value of the `int` type, which is `0`, if the call sets the context, or the value of the second parameter of the `longjmp()` function, which made the return possible.
- Function `longjmp()` has two parameters. The first one is the `jmp_buf` structure and the second one is of the `int` type and determines the return value of `setjmp()` function when the return is performed.

A macro that implements a reactivation point is defined as follows:

[4]The process of derivation is described in detail in section 2.2

```
#define csim_switch_to_process(process)                                    \
  if (setjmp(csim_sqs_point()->rollback) == 0) {                           \
    longjmp(process->rollback, 1);                                         \
  }
```

However, a function (i.e. program of the process) that uses the functions `setjmp()` and `longjmp()`, cannot ensure that the values of it's parameters, local (automatic) variables and return address are unchanged.

All these values are stored on a stack and the jump to another part of the program code can overwrite them. This has several important consequences. The function calling `setjmp()` and `longjmp()` functions does not have the storing of its parameters guaranteed, and should not therefore require any parameter. If it is needed, the function should process them before calling any of the *critical functions* (i.e. `setjmp()` and `longjmp()`). The same situation is with the local variables of the function.[5]

The last problem is the corruption of the return address. Hence it is necessary to make sure, that the function using the setting or return to reactivation point, cannot end with the `return` statement. Otherwise there might occur a jump to a non-existing address and collapse of the whole program.

It means that operations that change the state of running process by calling `setjmp()` and `longjmp()` (e.g. `passivate()`) had to be written as macros. Moreover it is not possible to call a state-changing operation like `passive()` inside a blocking function (like e.g. `receive_message()`).

2.4.2 Implementation Based on POSIX Threads

The second method of implementation of the coroutines is based on the use of *threads*. A thread is a semi-process, that has its own stack, and executes a given piece of code. Unlike a computational process, the thread normally shares its memory with other threads. All threads are executed in parallel (e.g. using time slices).

To retain maximum portability of the code, POSIX threads were chosen as an interface to threads (IEEE standard 1003.1).

The `pthreads` library introduces three basic types of objects:

- threads
- synchronization objects (mutexes)
- condition variables, used for passive waiting of threads until specific conditions are met

New data types introduced by the `pthreads` library are:

- `pthread_t` for threads

[5] in the program class `auto`

- `pthread_mutex_t` for mutexes
- `pthread_cond_t` for condition variables

Moreover for each of the presented three basic data types an associated data type also exists within so-called attribute object that contains adjustable properties.

Every thread executes its own function with predefined prototype:

```
void* prog_name (void* arg);
```

The same thread function may be executed by several threads simultaneously. Local variables defined in the function are thread local variables and they are unique for each thread. Threads are created in the ready state.

A thread is created for each process of the pseudo–parallel processes method. For the implementation it is however necessary to assure that only one thread is being executed at a time. In the process record the following fields are present to accomplish this:

- `thread_id` – thread identifier
- `mutex` – mutex to assure exclusive access to the fields in process record
- `cond_var` – to assure mechanism of condition variable
- `thread_run` – condition variable the thread waits for when another thread is being executed

The currently running process has the `thread_run` field in its process record set to `TRUE`, all other threads have it set to `FALSE` and wait for the value to change.

Switching the processes is done as follows:

1. Currently running thread grabs the mutex for its process record and sets the field `thread_run` to `FALSE`.
2. Then it also grabs the mutex for process record of the planed process (or step function), sets its `thread_run` to `TRUE`, sends a signal for conditional variable and releases the mutex of the planed process. At this moment, both threads are being executed simultaneously.
3. The former thread now starts to wait for the condition `thread_run == TRUE`.
4. The newly executed thread releases its mutex (while it waits for the condition variable, its associated mutex is locked) and continues with process program execution.

A macro that contains a reactivation point is now defined as follows:

```
#define csim_switch_processes(from, to)                                    \
do {                                                                       \
  CSIM_PROCESS *_p_from, *_p_to;                                           \
  _p_from = (CSIM_PROCESS *) from;                                         \
  _p_to   = (CSIM_PROCESS *) to;                                           \
                                                                           \
  pthread_mutex_lock(&(_p_from->mutex));                                   \
  _p_from->thread_run = FALSE;                                             \
  pthread_mutex_lock(&(_p_to->mutex));                                     \
  _p_to->thread_run = TRUE;                                                \
  pthread_mutex_unlock(&(_p_to->mutex));                                   \
  pthread_cond_signal(&(_p_to->cond_var));                                 \
                                                                           \
  while (_p_from->thread_run != TRUE)                                      \
    pthread_cond_wait (&(_p_from->cond_var), &(_p_from->mutex));           \
  pthread_mutex_unlock(&(_p_from->mutex));                                 \
} while (0)
```

2.5 Comparison of both implementations

As described above, the use of `setjmp()` and `longjmp()` functions causes an invalidation of the stack, what in turn leads to invalidation of automatic local variables, parameters and return address in process's function. Therefore the long-jump based implementation sets some restricting rules for a potential library user, when creating programs of process–like objects. On the other side long-jump based implementation is effective enough, it means that big number of processes can be created and the computation runs much more rapidly. The rules for use long-jump based implementation are described in section 6.3.

In contrast the POSIX threads based implementation, due to the fact that each thread has its own stack, allows an unrestricted use of automatic variables and nested function calls. It enables to create a construction like `receive_message()` as function (using internally `csim_passivate()`) that when called from a process's program it causes context switching.

By following the long-jump implementation rules for writing programs, we get a program that will work correctly with both available implementations.

The long jumps functions are a part of the standard **C** language library what ensures better portability of the long-jump based implementation than the POSIX threads one.

2.6 Time Representation

The processes in the calendar are placed in ascending order by the planed time of their activity. To represent time a new data type `CSIM_TIME` is introduced in **C-Sim**. By default the type `CSIM_TIME` is defined in the file csim_tm.h as follows:

```
typedef double CSIM_TIME;
```

The `double` type is a 64-bit floating–point number, where one bit represents sign, 11 bits represent exponent and the rest 52 bits is the mantis, which in turn allows to represent floating–point numbers in the range $\pm 1.8 \times 10^{308} \div 2.2 \times 10^{-308}$ with the accuracy of around 15 decimal digits. This may seem enough but in case of a very long simulation (speaking of model time, not real time) the rounding may cause problems associated with inaccurate time.

The value of model (simulation) time is increased every step by its length, which is typically a random variable with a given mean value E and variance σ^2. After n steps (where n is reasonably large) the value of simulation time is equal to nE. For $n \approx 10^{15}$ the mean length of a step E is approximately equal to the the accuracy limit of the floating–point number representation. This may cause problems with insertion of processes to the calendar. The problem will occur sooner for higher vales of σ^2.

This is the main reason why the `CSIM_TIME` type definition is separated to the file csim_tm.h. Such approach makes it possible to replace the `double` type with any other desired type assuming that the following operations are implemented as well:

- `csim_time_add` – addition
- `csim_time_sub` – subtraction
- `csim_time_mul` – multiplication
- `csim_time_div` – division
- `csim_time_cmp` – comparison
- `csim_time_to_dbl` – conversion to double
- `dbl_to_csim_time` – conversion from double
- `CSIM_ZERO_TIME` – static initialization to zero value
- `csim_zero_time` – expression with the value of zero

For the basic double type the operations are defined as follows:

```
#define csim_time_add(op1, op2) ((op1) + (op2))
#define csim_time_sub(op1, op2) ((op1) - (op2))
#define csim_time_mul(op1, op2) ((op1) * (op2))
#define csim_time_div(op1, op2) ((op1) / (op2))
#define csim_time_cmp(op1, op2) ((op1) > (op2) ? 1 : (op1) < (op2) ? -1 : 0)
#define csim_time_to_dbl(time) ((double) time)
#define dbl_to_csim_time(time) ((CSIM_TIME) time)
```

2.7 Run-time Errors Checking

Because the run of the simulation program is dynamically partitioned into simultaneously running processes[6], the run-time error can manifest itself after a great amount of model time

[6]Which often use the random number generator for planning their activity phases.

and can be then hard to detect. That is the reason why the **C-Sim** library stresses the error handling and preventive checks. Every parameter of the **C-Sim** library functions is checked inside the function. The checking verifies whether the value is within the allowable bounds. In case of pointers to objects also the internal integrity is verified (i.e., whether the pointer really points to a valid object of the respective type).

For each of basic object types from the **C-Sim** library there is defined an enumeration type, which defines all possible states of the object or the pointer to the object during the check–further referred to as *check states*. For checking the pointers to objects the functions `csim_check_...()` and `csim_..._state()` are available. The functions forming the first group deliver the information, whether the object is or is not correct. The functions forming the second group returns then the value of enumeration type, which determines the check states of the object.

If a discrepancy is discovered during the check of parameters, the function calls the error operation `csim_exception()` with error code as its parameter. This operation sets the error flag and, if possible, jumps back to the `csim_step()` function to end the simulation step. The `csim_step()` function indicates, that an error occurred, by returning the `FAILURE` value.[7] In this case it is possible to terminate the main loop of simulation and call `csim_error()` to detect the reason. By calling the `csim_error_status()` function we can test anytime, whether or not an error occurred and then the `csim_error()` function can return a detailed information about the error. Moreover it is possible to call the function `csim_debug()` in order to analyze the state of data structures preserved at the time when the error occurred.

For debugging the simulation program we can use also the conventional debugging tools and methods. The basic possibility is to perform the listing of selected data after each simulation step. Better debugging tools (step mode, breakpoints, listing the values of selected variables) are usually provided by the **C** language development environment. However, it must be kept in mind, that the usability of these tools is strongly limited by the fact, that the run-time error may occur after a long run time.

More details about the check states of the objects and error handling will be found in the paragraph 3.7.

[7] Return value is of type `CSIM_RESULT` described in section 3.1

Chapter 3

Data Types in C-Sim Library

3.1 Simple data types

The **C-Sim** library provides several integer data types. For these data types their width is assured on 16-bit and 32-bit platforms. The definition relies on the value of the constant `UINT_MAX`. Detailed description of the data types is in table 3.1.

Type	Sign & Width	16-bit platform	32-bit platform
`CSIM_BYTE`	8-bit signed	`signed char`	`signed char`
`CSIM_UBYTE`	8-bit unsigned	`unsigned char`	`unsigned char`
`CSIM_WORD`	16-bit signed	`signed int`	`signed short int`
`CSIM_UWORD`	16-bit unsigned	`unsigned int`	`unsigned short int`
`CSIM_LONG`	32-bit signed	`signed long int`	`signed int`
`CSIM_ULONG`	32-bit unsigned	`unsigned long int`	`unsigned int`

Table 3.1: **C-Sim** integer types

Furthermore the following two useful types are also defined for logical values:

```
typedef enum {
  SUCCESS,
  FAILURE
} CSIM_RESULT;

typedef enum {
  FALSE,
  TRUE
} CSIM_BOOLEAN;
```

The last simple type is `CSIM_TIME`, which is described in detail in the section 2.6.

3.2 Object Type CSIM_DYN_MEM

3.2.1 Object Structure

This type enables dynamic object creation allocated in the memory area called heap. All other basic object types are derived from this type to allow **C-Sim** to create objects dynamically.[1] Functions working with the type `CSIM_DYN_MEM` are described in section 4.1. Enumeration type for possible check states of the object or pointers to an object of the type `CSIM_DYN_MEM` is described in section 3.7.

Type `CSIM_DYN_MEM` contains the necessary links needed for maintaining the list of all dynamically generated objects and basic attributes (i.e. data items) common to all data types. Declaration of the `CSIM_DYN_MEM` type:

```
#define csim_d_dyn_mem              \
  char                 *name;       \
  char                 type;        \
  char                 *check;      \
  struct csim_dyn_mem  *p_head_mem; \
  struct csim_dyn_mem  *p_suc_mem;  \
  struct csim_dyn_mem  *p_pred_mem; \
  CSIM_DESTRUCTOR      destructor;  \
  CSIM_VIEW            view

typedef struct csim_dyn_mem {
  csim_d_dyn_mem;
} CSIM_DYN_MEM;
```

Attributes of the objects of this type have the following meaning:

`name` – the user has the possibility to name individual objects; this name can be used in the function `csim_view_dyn_mem()` or otherwise in the visualization layer of the application, it can be set by the function `csim_set_name()`, which allocates necessary memory and copies the name to it, upon release of the object or by another call to `csim_set_name()` this memory is automatically released as well,

`type` – determines the object type,

- M ... `CSIM_DYN_MEM` and derived types
- L ... `CSIM_LINK` and derived types
- H ... `CSIM_HEAD` and derived types
- P ... `CSIM_PROCESS` and derived types

`check` – pointer used for verification of all pointers to this object,

[1] Reasons, advantages and disadvantages of this restriction are described in section 2.3.

`p_head_mem` – pointer to the list of all dynamically generated objects,

`p_suc_mem` – pointer to the successor in the list,

`p_pred_mem` – pointer to the predecessor in the list of all objects,

`destructor` – pointer to the destructor function, that allows to deallocate the memory dynamically allocated to new (added) attributes,

`view` – function that displays an object. It can be used for debugging purposes. In the Console debug module there are the standard functions `csim_view_...()`. To set the pointer to the `view` function it is necessary to use the following macro:

```
#define csim_set_view(P_DYN_MEM, VIEW) (P_DYN_MEM)->view = (VIEW)
```

3.2.2 Object Construction and Destruction

To generate a new object `CSIM_DYN_MEM` or a derived type it is necessary to use the predefined macro `csim_new_dyn_mem()`. For example:

```
p_dyn_mem = csim_new_dyn_mem(MY_DYN_MEM);
```

`p_dyn_mem` is a pointer to the newly generated object (if its value is `NULL`, there isn't enough memory),

`MY_DYN_MEM` is the type of the generated object.

Function `csim_dispose_dyn_mem()` serves for releasing the memory and cancellation of an object.

3.3 Object Type CSIM_LINK

3.3.1 Object Structure

This type allows to create the objects that may be inserted into the two-way circular lists. It contains the pointers needed for maintaining the list and a pointer to the so-called *list head*. Furthermore it contains the item `enter_time`, which is used for generating statistics of the behavior of the objects of the type `CSIM_HEAD`. Objects of this type can be created only dynamically, and they are thus derived from the type `CSIM_DYN_MEM`. In addition to creation and cancellation, another possible operations with the objects of the type `CSIM_LINK` include verification of the pointer and inner integrity of the object, object displaying, insertion to the end of a list, deleting from a list, insertion before and after another object in a list. Functions are described in section 4.2. The enumeration type for possible check states of an object or pointer to an object type `CSIM_LINK` is described in section 3.7.

For the introduction of derived types the following macros are used: `csim_d_link` or brackets `csim_derived_link` and `csim_end_derived` (see section 2.2). They guarantee the inheritance

of the supported type items into the derived type. It is thus possible to derive a proper type with concrete added data items from a general element type in a list. Declaration and explanation of the meaning of individual items of the type CSIM_LINK follow:

```
#define csim_d_link                 \
  csim_d_dyn_mem;                   \
  struct csim_link     *suc;        \
  struct csim_link     *pred;       \
  struct csim_head     *head;       \
  CSIM_TIME            enter_time

typedef struct csim_link {
  csim_d_link;
} CSIM_LINK;
```

suc – pointer to the successor in the list,

pred – pointer to the predecessor in the list,

head – pointer to the list head,

enter_time – time of insertion of an element into the list; it is used for processing the statistics of object type CSIM_HEAD behavior.

3.3.2 Object Construction and Destruction

For the generation of a new object of the type CSIM_LINK or derived types, a predefined macro csim_new_link() must be used. This inserts the new object into the list of dynamic memory blocks and initializes basic items of the type CSIM_LINK. For example:

```
p_link = csim_new_link(MY_LINK);
```

p_link is a pointer to the newly generated object (if its value is NULL, there isn't enough memory),

MY_LINK is the type of the generated object.

Function csim_dispose_link() serves for releasing the memory and cancellation of an object.

3.4 Object Type CSIM_HEAD

3.4.1 Object Structure

Objects of the type CSIM_HEAD represent a *list* of objects with the type CSIM_LINK (or a type derived from CSIM_LINK). They are kind of a representation of the list and maintain basic

information about it. Their data items are the pointers to the first an the last element of the list, actual list length and actual statistic data. The basic operations delivered over the type CSIM_HEAD are in addition to creating and cancellation of an object also finding the first and the last element in the list, test for emptiness of the list and replenishing the list. Functions are described in section 4.2.

The type CSIM_HEAD is derived from CSIM_LINK, that is derived from CSIM_DYN_MEM. From the fact, that CSIM_HEAD is derived from CSIM_LINK,[2] there follows the possibility to insert a CSIM_HEAD-typed object into (other) lists of the same or derived type, and hence the possibility to create more complicated list structures (e.g. trees). From the fact, that the type CSIM_HEAD is (indirectly) derived from CSIM_DYN_MEM there follows the possibility (and at the same time necessity) to create the objects of this type dynamically. Declaration of the type CSIM_HEAD follows:

```
#define csim_d_head                 \
  csim_d_link;                      \
  CSIM_LINK       *p_first;         \
  CSIM_LINK       *p_last;          \
  CSIM_UWORD      lq;               \
  CSIM_ULONG      arrival_cnt;      \
  CSIM_TIME       last_time;        \
  CSIM_TIME       sum_tw;           \
  CSIM_TIME       sum_lw;           \
  CSIM_TIME       sum_time;         \
  CSIM_BOOLEAN    statistics

typedef struct csim_head {
  csim_d_head;
} CSIM_HEAD;
```

The attributes used in the declaration have the following meaning:

p_first – pointer to the first item in the list,

p_last – pointer to the last item in the list,

lq – current length of the list,

arrival_cnt – count of items inserted into the list when statistics is enabled,

last_time – time of a last modification of the list length,

sum_tw – sum of all item waiting times within the list,

sum_lw – list's history, i.e. sum of current list lengths weighted by their duration,

sum_time – sum of times, when statistics was enabled,

[2] In contrast to **SIMULA/Simset**, where CSIM_LINK and CSIM_HEAD have a common ancestor LINKAGE.

`statistics` – enables/disables the statistics,

- when the global variable `statistics` is set to `TRUE`, list statistics is calculated,
- when the global variable `statistics` is set to `FALSE`, list statistics is calculated when this attribute is set to `TRUE` (i.e., only for the selected lists).

To explain the meaning of the items added for monitoring of the statistics, it is necessary to add, that the **C-Sim** library provides the computation of some statistical values for objects of the type `CSIM_HEAD` (and derived types as well), namely *mean list length* – L_w and *mean time spent by an element in the list* – T_w. It is possible to compute the statistics for all objects, or to switch it on/off selectively only for some of them. The list of functions controlling the collection of the data and statistics processing can be found in 4.5.

3.4.2 Object Construction and Destruction

To generate a new object of the type `CSIM_HEAD` (or a derived type) we must use a predefined macro `csim_new_head()`. For releasing the memory and canceling the object we can use the `csim_dispose_head()` function. Usage of these functions is the same as for an object of the type `CSIM_LINK`. Also the information about the possible check states of the object or pointer to the object can be found in section 3.7.

3.5 Object Type CSIM_PROCESS

Process is a computational activity, running in pseudo–parallel mode with other processes. Every process is a pair:

- program,
- process data record.

Several processes may share an identical program code (which must thus be designed as *reentrant*). On the other hand, a data item belongs exclusively to a single process. The type `CSIM_PROCESS` declares structure of process data record (or precisely its standard part). It is derived from the type `CSIM_LINK`, which allows simply to create process lists within the simulation model. The user may in a simple way, described in section 2.2, create his own process types.

To the operations connected with the type `CSIM_PROCESS` belong (in addition to creating and cancellation of an object) namely operations for changing the state of the running process or for changing the state of another process from the running process (the states are described in 3.5.2). The functions implementing the operations are described in section 4.3. The enumeration type for possible check states of the object or pointer to the object type `CSIM_PROCESS` is described in section 3.7.

3.5.1 Process Program

Process program is a function in **C** language, having one parameter and returning no result. The parameter is a pointer to data record of the process, but for the library user this parameter is transparent (i.e. it is used internally). The user defines the program using the pair of program brackets introduced as two macros. The opening bracket has the syntax `csim_program(TYPE_NAME, PROG_NAME)` and its definition is as follows:

```
#define csim_program(TYPE_NAME, PROG_NAME)                              \
void *PROG_NAME(void *p_void)                                           \
{                                                                       \
  TYPE_NAME *p_my = (TYPE_NAME *) p_void;
```

Immediately following the opening bracket of the program we may define other internal data of the process.[3] They are typically used for "auxiliary" variables that don't preserve a value to the next reactivation of the process. The rules for their use depend on the memory class, in which these variables are defined. Memory for `automatic` variable is allocated in a stack, memory for `static` variable is firmly allocated in data segment of the simulation program during all its run. The rules for stack–allocated data depends on the chosen case of pseudo-parallel processes implementation (see 2.4).

Variable `p_my` defined in macro `csim_program` contains in any given moment the pointer to the presently active process (i.e. to its data structure) which runs according to the defined program. The simulation algorithm automatically switches `p_my` to the active process. Variable `p_my` does not need to preserve its value between the activity phases, because after every context switching it is initialized by the pointer value to the presently active process. It is thus defined in memory class `auto`. Using the library macro

```
#define my (*p_my)
```

a reference may be done in the program process to the data record by the construction `my.attribute`, similarly as to the structure items. The identifier `my` thus refers to data structure of the process.

C-Sim enables two ways of ending a process program depending on whether or not the data record is destroyed immediately after the program end. If the data is to be destroyed, the macro `csim_end_process` will be used, otherwise `csim_end_program`. In the second case the process state is set to `CSIM_TERMINATED` and cannot be activated any more. Process data record is accessible until it is destroyed by calling `csim_dispose_process()`.[4]

3.5.2 Process States

The process during its "life" may stay in different states. The process state can be determined using function `csim_state()`, which returns the value of enumerated type (i.e. process state), defined in the following way:

[3] I.e., data, whose name is usable only in the program.

[4] In **SIMULA** the objects are released only when they aren't referenced any more.

```
typedef enum {
  CSIM_NEW_PASSIVE, CSIM_NEW_PLANNED, CSIM_PLANNED, CSIM_ACTIVE,
  CSIM_PASSIVE, CSIM_TERMINATED, CSIM_ZOMBIE
} CSIM_PROC_STATE;
```

`CSIM_NEW_PASSIVE` – new passive process, which has not yet been active, and therefore will be executed from the program beginning,

`CSIM_NEW_PLANNED` – new planned process, which has not yet been active, therefore it will be executed from the program beginning,

`CSIM_PLANNED` – planned process, which will run from a reactivation point,

`CSIM_ACTIVE` – process, whose program is active and running,

`CSIM_PASSIVE` – process, which is waiting for planning its further activity (from another process),

`CSIM_TERMINATED` – process, which terminated execution of its program using `csim_end_program`.

`CSIM_ZOMBIE` – process, which terminated execution of its program using `csim_end_process` and its process record will be destroyed immediately after the return to `csim_step()`.

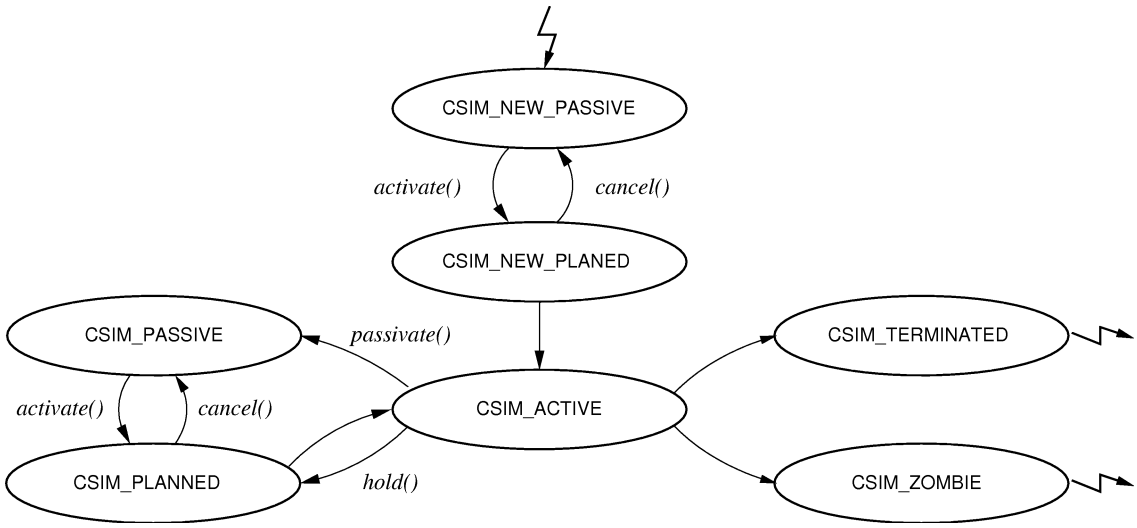

Figure 3.1: Process state transition diagram

3.5.3 Process Data Record – type CSIM_PROCESS

For the data record of a process the basic type `CSIM_PROCESS` is introduced in **C-Sim**. Its items contain the information needed for management of pseudo-parallel computing and recording statistic characteristics of process behavior. Declaration of the type `CSIM_PROCESS` follows:

```
#define csim_d_process                    \
  csim_d_link;                            \
  csim_process_kr;                        \
  CSIM_PROC_PROGRAM    program;           \
  struct csim_process  *suc_in_sqs;       \
  struct csim_process  *pred_in_sqs;      \
  CSIM_PROC_STATE      state;             \
  CSIM_TIME            evtime;            \
  CSIM_TIME            last_time;         \
  CSIM_TIME            non_passive_time;  \
  CSIM_TIME            passive_time;      \
  CSIM_BOOLEAN         statistics

typedef struct csim_process {
  csim_d_process;
} CSIM_PROCESS;
```

The items in the type declaration have the following meaning:

`program` – pointer to the process program, which is started together with the first start of a process (i.e., if the process state is `CSIM_NEW_...`), otherwise process continues from reactivation point,

`suc_in_sqs` – pointer to the successor in scheduling calendar,[5]

`pred_in_sqs` – pointer to the predecessor in calendar,

`state` – process state,

`evtime` – reactivation time, that determines process order in calendar,

`last_time` – time of last insertion into the calendar,

`non_passive_time` – sum of times, when the process was planned,

`passive_time` – sum of times, when process was passive,

`statistics` – determines, whether the statistics will be performed (for explanation, see the description of equally named attribute of the type `CSIM_HEAD`).

Macro `csim_process_kr` inserts implementation dependent part of the process record.

3.5.4 Object Construction and Destruction

A new object of the type `CSIM_PROCESS` (and of derived types as well) must be generated with a predefined macro `csim_new_process()`. For example:

[5] `sqs` is from **SIMULA**'s *sequencing set.*

```
p_process = csim_new_process(MY_PROCESS, my_program);
```

`p_process` – is pointer to the generated object (if its value is `NULL`, there isn't enough memory),

`MY_PROCESS` – is type of generated object,

`my_program` – is pointer to the program, according to which the process will run; this pointer is typed as `CSIM_PROC_PROGRAM`.[6]

Parameters of the macro are the type name, which must be either `CSIM_PROCESS` or derived type, and program name, i.e., name used somewhere in the corresponding program "opening bracket" `csim_program` .

The created process is passive, i.e., the data exists, but the program has not been started yet. Its activation must be made from main program or from another process by means of any activation function (see 4.3).

If a process program is terminated with the "bracket" `csim_end_process`, the object (i.e. data) of the process is released immediately after the process termination. If, on the contrary, the process program is terminated with the macro `csim_end_program`, the process data record isn't automatically released and in case of necessity can be destroyed by calling function `csim_dispose_process()`.

3.6 Initialization of Attributes Values

Primary initialization of the data items taken over from the basic types is done automatically when generating an object using a proper macro `csim_new_...()`. Explicit initialization is possible with functions `csim_init_...()` (see section 4.1). Initialization of user-defined attributes (i.e. attributes added when declaring the derived types) must be done by the **C-Sim** user either by direct access to the items through the pointer representing the object or (better) by means of an user–defined initialization function.

Direct access to the basic data types attributes unfortunately cannot be prevented.[7] However, the **C-Sim** users should be warned, that although the access to the attributes "inherited" from the basic types is possible, it can in no case be recommended. Great hazard of possible object data inconsistency and occurrence of hard-to-debug errors arise. This is one of the reasons, why in the **C-Sim** library the inner integrity check is used despite of its time consumption.

[6] The name `PROG_NAME` defined by means of `csim_program(TYPE_NAME, PROG_NAME)` has the type `CSIM_PROC_PROGRAM` automatically.

[7] **C** language, in contrast to e.g. **C++**, has no tool for this.

3.7 Error enumeration types

Each parameter of **C-Sim** library functions is checked before use.[8] When the parameters are of pointer type, the validity of the pointer and the internal integrity of the referenced object is checked.

For each basic type of object, an enumeration type is defined, specifying the possible states of an object, or a pointer to an object during the check. To check the pointers to objects the `csim_check_...()` and `csim_..._state()` functions are defined. The functions from the first group inform whether or not the object is correct. Functions from the second group return a value of enumeration type, which determines, in which of the check states is the object.

Enumeration type used to describe states of the `CSIM_DYN_MEM` object during the check:

```
typedef enum {
  CSIM_D_DEFECTIVE, CSIM_D_ILLEGAL, CSIM_D_IS_NULL,
  CSIM_D_IS_RANK, CSIM_D_NOT_RANK
} CSIM_DYN_MEM_CHECK;
```

Explanation of individual values:

`CSIM_D_DEFECTIVE` – object is corrupted for unknown reason,

`CSIM_D_ILLEGAL` – pointer to the object does not correspond to its check value `char *check` set by object's generation,

`CSIM_D_IS_NULL` – pointer to checked object is `NULL`,

`CSIM_D_IS_RANK` – object is in the list of dynamic blocks,[9]

`CSIM_D_NOT_RANK` – object is not in the list of dynamic memory blocks.

Enumeration type used to describe states of the `CSIM_LINK` object during the check:

```
typedef enum {
  CSIM_L_DEFECTIVE, CSIM_L_ILLEGAL, CSIM_L_IS_NULL,
  CSIM_L_IS_RANK, CSIM_L_NOT_RANK
} CSIM_LINK_CHECK;
```

Values `CSIM_L_DEFECTIVE`, `CSIM_L_ILLEGAL` and `CSIM_L_IS_NULL` have the similar meaning as for the type `CSIM_DYN_MEM_CHECK`. Explanation of other values:

`CSIM_L_IS_RANK` – object is inserted into some list (this means `suc`, `pred` and `head` are not `NULL`),

[8] Checked are especially parameters of "public" library functions. The private library functions have limited checks, so they should not be used directly in the simulation programs.

[9] Probably. We can only be sure that pointers `p_suc_mem`, `p_pred_mem` and `p_head_mem` are not `NULL`.

CSIM_L_NOT_RANK – object is not inserted in any of the lists (suc, pred and head are NULL).

Enumeration type used to describe states of the CSIM_HEAD object during the check:

```
typedef enum {
  CSIM_H_DEFECTIVE, CSIM_H_ILLEGAL, CSIM_H_IS_NULL,
  CSIM_H_EMPTY, CSIM_H_NOT_EMPTY
} CSIM_HEAD_CHECK;
```

Explanation of individual values:

CSIM_H_EMPTY – list is empty (p_first and p_last are NULL),

CSIM_H_NOT_EMPTY – list is not empty (pointers p_first and p_last are not NULL).

Enumeration type used to describe states of the CSIM_PROCESS object during the check:

```
typedef enum {
  CSIM_P_DEFECTIVE, CSIM_P_ILLEGAL, CSIM_P_IS_NULL, CSIM_P_IS_TERMINATED,
  CSIM_P_IS_RANK, CSIM_P_NOT_RANK
} CSIM_PROC_CHECK;
```

Explanation of individual values:

CSIM_P_IS_RANK – process is in the calendar (probably). We can only be sure that suc_in_sqs and pred_in_sqs are not NULL. Process state may be CSIM_NEW_PLANNED, CSIM_PLANNED or CSIM_ACTIVE,

CSIM_P_NOT_RANK – object is not in the calendar (suc_in_sqs and pred_in_sqs are NULL), Process state may be CSIM_NEW_PASSIVE or CSIM_PASSIVE.

CSIM_P_IS_TERMINATED – process has ended. Process state may be CSIM_TERMINATED or CSIM_ZOMBIE.

Described checking mechanism can detect most wrong manipulations with objects during the simulation run. The check function will then call the error operation csim_exception() with an error code. The csim_exception() function will set the error flag and then returns to csim_step() function if possible. By this the current simulation step is ended. The csim_step() function indicates error in the terminated simulation step by its return value (set to FAILURE).

By calling the function csim_error_status() we can further determine, whether an error occurred during the simulation step and the csim_error() function will return detailed information about this error. Error information has the following structure:

```
typedef struct {
  CSIM_UWORD      error_code;
  CSIM_PROCESS    *p_proc_error;
  void            *p_void;
} CSIM_ERROR;
```

where the single items are:

`error_code` – integer value that describes the error,

`p_proc_error` – pointer to a process, during whose execution the error occurred,

`p_void` – pointer to an object that caused the error (it can be used for a user-written object visualization and check).

Value of `error_code` is generated by the following macro:

```
#define CSIM_ERR_CODE(func, err) (CSIM_UWORD) ((err << 8) | func)
```

Error code comprises two parts. Within the lower eight bits there is a code, which determines the function where the error was detected or optionally also which parameter caused it. In the higher bits there is the current state of the object – result of the `csim_..._state()` function. Therefore it is important that the symbolic constants `CSIM_?_DEFECTIVE`, `CSIM_?_ILLEGAL` and `CSIM_?_NULL` retain the same numeric value for all named types `CSIM_..._CHECK`. These higher bits of error code are equal to `0` for all errors that were not caused by the pointer to the object.

To ease the translation of the error code to a more friendly text message (usable as error message displayed on the console), the function `csim_error_msg()` is defined as a part of Error Messages module 5.1.

User can call `csim_exception()` function in the process program for user-detected errors and/or nonstandard model behavior, error code should then be in the interval <`CSIM_USER_ERROR_CODE`, 255>.

Chapter 4

Library Functions Reference

This chapter describes all functions that are supported by **C-Sim** library. Every function has a built-in check of input parameters, and/or further checks of suitability of context of its use. All objects have an internal check attribute `check` that is set in the initialization section and authenticates referencing pointer validity and internal integrity of the pointed object and nonzero value of the input pointer.

Most of library functions return value of `CSIM_RESULT` type by which is possible to recognize an error state. For detailed information on the mechanism of the run-time checks see section 3.7. List of exceptions (or more accurately symbolic integer values of error codes), that can occur during the call of individual functions, is a part of csim.h file.

If an object is in the parameter list, the corresponding formal parameter must be typed as a pointer to a corresponding basic type. A function can be naturally called also over an object of any derived type, however it is necessary to explicitly cast the inserted pointer.[1] Example:

```
MY_HEAD *p_queue;  MY_LINK *p_elem;
  ...
csim_into((CSIM_LINK *) p_elem, (CSIM_HEAD *) p_queue);
```

4.1 Memory Management Functions

```
CSIM_RESULT csim_init_mem(void);
```

Initialization of dynamic memory and **C-Sim** global variables. Return `FAILURE` in case of insufficient memory.

```
void csim_clear_mem(void);
```

Cancels the list of dynamic memory blocks and releases the dynamically allocated memory.

[1] Unlike **C**, object-oriented programming languages, that support the method polymorphism, perform this pointer conversion automatically.

```
CSIM_DYN_MEM *csim_init_dyn_mem(CSIM_DYN_MEM *p_dyn_mem);
```

Initialization of the object p_dyn_mem.

```
CSIM_LINK *csim_init_link(CSIM_LINK *p_link);
```

Initialization of the object p_link.

```
CSIM_HEAD *csim_init_head(CSIM_HEAD *p_head);
```

Initialization of the object p_head.

```
CSIM_PROCESS *csim_init_process(CSIM_PROCESS *p_process,
                                CSIM_PROC_PROGRAM proc_prog);
```

Initialization of the object p_process.

```
CSIM_RESULT csim_dispose_dyn_mem(CSIM_DYN_MEM *p_dyn_mem);
```

Releases the memory used by the object p_dyn_mem.

```
CSIM_RESULT csim_dispose_link(CSIM_LINK *p_link);
```

Releases the memory used by the object p_link.

```
CSIM_RESULT csim_dispose_head(CSIM_HEAD *p_head);
```

Releases the memory used by the object p_head.

```
CSIM_RESULT csim_dispose_process(CSIM_PROCESS *p_process);
```

Releases the memory used by the object p_process.

4.2 Operations with two–way circular lists

The library provides similar functions as the **SIMULA** language. Operations are defined as functions and macros in **C** language, and in contrast to the functions—methods of object type, they have a pointer to the actual object in the parameter list.

```
CSIM_RESULT csim_into(CSIM_LINK *p_link,
                      CSIM_HEAD *p_head);
```

Inserts element p_link at the end of the list p_head.

```
CSIM_RESULT csim_out(CSIM_LINK *p_link);
```

Removes element from the list.

```
CSIM_LINK *csim_first(CSIM_HEAD *p_head);
```

Returns pointer to the first element in the list. If an error occurs or the list is empty, the function returns NULL.

```
CSIM_LINK *csim_last(CSIM_HEAD *p_head);
```

Returns pointer to the last element in the list. If an error occurs or the list is empty, the function returns NULL.

```
CSIM_BOOLEAN csim_empty(CSIM_HEAD *p_head);
```

Checks whether the list is empty. Returns FALSE if it isn't empty and TRUE if it is empty or an error occurred.

```
CSIM_WORD csim_cardinal(CSIM_HEAD *p_head);
```

Returns actual list length or -1 if an error occurs.

```
CSIM_RESULT csim_clear(CSIM_HEAD *p_head);
```

Clears the list. Since the integrity of user data may be corrupted, items forming the list aren't destroyed. Hence this responsibility remains with the programmer, only the programmer may decide, how to correctly cancel individual items.

```
CSIM_RESULT csim_follow(CSIM_LINK *p_link_what,
                        CSIM_LINK *p_link_where);
```

Inserts element p_link_what after another element p_link_where.

```
CSIM_RESULT csim_precede(CSIM_LINK *p_link_what,
                         CSIM_LINK *p_link_where);
```

Inserts element p_link_what before another element p_link_where.

4.3 Operations with Processes

These operations serve for scheduling processes and provide access to the basic information about individual processes. Most of them cannot be applied on currently active process.

```
CSIM_RESULT csim_activate_at(CSIM_PROCESS *p_process,
                             CSIM_TIME t);
```

Schedules process to the time `t`. In order to use this function, the process must not be scheduled so far. To reschedule an already scheduled process use the `csim_reactivate()` function.

```
CSIM_RESULT csim_activate_delay(CSIM_PROCESS *p_process,
                                CSIM_TIME del_t);
```

Schedules process to the time (`csim_time() + del_t`). In order to use this function, the process must not be scheduled so far. To reschedule an already scheduled process use the `csim_reactivate_delay()` function.

```
CSIM_RESULT csim_reactivate_at(CSIM_PROCESS *p_process,
                               CSIM_TIME t);
```

Re-schedules an already scheduled process to new time `t`.

```
CSIM_RESULT csim_reactivate_delay(CSIM_PROCESS *p_process,
                                  CSIM_TIME del_t);
```

Re-schedules an already scheduled process to new time `csim_time() + del_t`.

```
CSIM_RESULT csim_cancel(CSIM_PROCESS *p_process);
```

Removes process from calendar. The argument `p_process` must not point to the current active process, otherwise an error occurs.

```
CSIM_PROC_STATE csim_state(CSIM_PROCESS *p_process);
```

Returns process current state.

```
CSIM_BOOLEAN csim_idle(CSIM_PROCESS *p_process);
```

Test of the process state. Returns `TRUE` if the process is passive, and `FALSE` for all other states including errors.

```
CSIM_TIME csim_evtime(CSIM_PROCESS *p_process);
```

Returns time, at which the process is scheduled.

```
CSIM_PROCESS *csim_next_proc(void);
```

Returns pointer to the second process in scheduler. If there is no such process in calendar, it returns `NULL`.

```
CSIM_TIME csim_time(void);
```

Returns actual value of model time.

```
CSIM_PROCESS *csim_current(void);
```

Returns pointer to the first record in calendar. Inside `csim_step()` it is the running process, otherwise it is the last running process).

```
CSIM_PROCESS *csim_sqs_point(void);
```

Returns pointer to the **C-Sim** calendar.

```
CSIM_DYN_MEM *csim_mem_point(void);
```

Returns pointer to the **C-Sim** heap.

4.4 Functions for the Control of Simulation Run

These operations serve for activity transfer among individual processes. Operations `csim_hold()`, `csim_passivate()` and `csim_wait()` are defined as macros as it was explained in the section 2.4.

```
CSIM_RESULT csim_step(void);
```

Realizes one simulation step. Transfers control to the first process in calendar. In the long-jump implementation must be called always at the same level of stack.

```
csim_hold(CSIM_TIME del_t)
```

Defined as macro. Plans current process to start after `del_t` time interval and transfers control back to the `csim_step()` function. Must be called directly in the process's prime function.

`csim_passivate()`

As `csim_hold()`, but passivates current process.

`csim_wait(CSIM_HEAD *p_head)`

As `csim_passivate()`, moreover inserts current process into `p_head`.

Note: The `csim_step()` function must be used only within the `main()` program function - it typically serves to implement the main loop of simulation. Other process-switching functions given here have to be used in a program of simulation process.

4.5 Statistical Functions

This group of functions controls computation of statistical values characterizing the behavior of individual objects. Statistical parameters are monitored for objects of the type `CSIM_HEAD` and `CSIM_PROCESS`. Statistics may be controlled globally for all objects or locally for individual objects. Statistics in the library **C-Sim** includes for the computation of average length of individual lists and mean time, which an element spends within the list (object of the type `CSIM_HEAD`). For objects of the type `CSIM_PROCESS` the mean time of staying in the state `CSIM_PLANNED` and mean time of staying in the state `CSIM_PASSIVE` are computed.

C-Sim embedded statistics processing properties are aimed mainly for purposes of a quick experimentation without great requirements as for precision of results. Overflowing of used data values is not handled and/or reported. To obtain serious statistics values, the user should implement his/her own statistics processing.

```
void csim_stat_on(void);
```

Enables global statistics. It is performed on all objects, which have given this possibility.

```
void csim_stat_off(void);
```

Disables global statistics. Only objects with enabled local statistics will then be monitored.

```
CSIM_BOOLEAN csim_stat_status(void);
```

Returns state of global statistics flag; `TRUE`/`FALSE` – enabled/disabled.

```
CSIM_RESULT csim_h_stat_on(CSIM_HEAD *p_head);
```

Enables local statistics for a list of the type `CSIM_HEAD`.

```
CSIM_RESULT csim_h_stat_off(CSIM_HEAD *p_head);
```

Disables local statistics for a list of the type CSIM_HEAD.

```
CSIM_RESULT csim_init_h_stat(CSIM_HEAD *p_head);
```

Initializes variables serving for the list statistic.

```
CSIM_BOOLEAN csim_h_stat_status(CSIM_HEAD *p_head);
```

Returns state of local statistics flag; TRUE/FALSE – enabled/disabled.

```
CSIM_RESULT csim_h_stat(CSIM_HEAD *p_head,
                        double *p_Lw,
                        double *p_Tw);
```

Returns results of the statistics of the object p_head. Argument p_Lw is the average length of the list and argument p_Tw is the mean time that an element waits within the list.

```
CSIM_RESULT csim_p_stat_on(CSIM_PROCESS *p_process);
```

Enables local statistics for the object p_process.

```
CSIM_RESULT csim_p_stat_off(CSIM_PROCESS *p_process);
```

Disables local statistics for the object p_process.

```
CSIM_RESULT csim_init_p_stat(CSIM_PROCESS *p_process);
```

Initializes variables serving for the process statistic.

```
CSIM_BOOLEAN csim_p_stat_status(CSIM_PROCESS *p_process);
```

Returns state of local statistics flag; TRUE/FALSE – enabled/disabled.

```
CSIM_RESULT csim_p_stat(CSIM_PROCESS *p_process,
                        double *p_Ts,
                        double *p_Tr);
```

Returns results of statistics of the object CSIM_PROCESS. Argument p_Ts is the mean time spent in planned state and argument p_Tr is the mean time spent in passive state.

4.6 Control and Run–Time Error Handling Functions

```
void csim_error(CSIM_ERROR *p_error);
```

Fills the structure p_error with the current error that interrupted the simulation.

```
CSIM_BOOLEAN csim_error_status(void);
```

Checks if an error occurred, in this case returns TRUE, otherwise FALSE.

```
CSIM_DYN_MEM_CHECK csim_dyn_mem_state(CSIM_DYN_MEM *p_dyn_mem);
```

Returns check result of the object p_dyn_mem.

```
CSIM_LINK_CHECK csim_link_state(CSIM_LINK *p_link);
```

Returns check result of the object p_link.

```
CSIM_HEAD_CHECK csim_head_state(CSIM_HEAD *p_head);
```

Returns check result of the object p_head.

```
CSIM_PROC_CHECK csim_process_state(CSIM_PROCESS *p_process);
```

Returns check result of the object p_process.

```
CSIM_BOOLEAN csim_check_dyn_mem(CSIM_DYN_MEM *p_dyn_mem);
```

Returns result of a check of the object p_dyn_mem.

```
CSIM_BOOLEAN csim_check_link(CSIM_LINK *p_link);
```

Returns result of a check of the object p_link.

```
CSIM_BOOLEAN csim_check_head(CSIM_HEAD *p_head);
```

Returns result of a check of the object p_head.

```
CSIM_BOOLEAN csim_check_process(CSIM_PROCESS *p_process);
```

Returns result of a check of the object p_process.

Chapter 5

Optional Modules Description

This chapter describes the optional modules of the **C-Sim** library. These modules are not mandatory to a functional simulation but often are very useful. The user can use a chosen module simply by adding the module header file using the `#include` directive.

5.1 Error Messages Module

This module provides the function `csim_error_msg()` that translates the given numerical error code into a human readable text. The returned value is pointer to a statically allocated string. Function prototype:

```
char *csim_error_msg(CSIM_UWORD code);
```

The message comprises two parts, just like the error code, both parts are stored within statically initialized arrays of strings, the corresponding parts of the error code are used as indexes into these arrays. Every execution of this function copies the error message selected by the argument `code` into a static buffer. The function `csim_error_msg()` then returns pointer to this buffer.

5.2 Random Number Generator Module

Because **C-Sim** and other similar tools are often used in stochastic processes based simulations, a random number generator is necessarily an important part of such tools. The most common method of random number generation is so called *congruent method*, which is based upon formulae of this type

$$y_i = C_0 + \sum_{j=1}^{m} C_j y_{i-j} \quad mod \quad M$$

where the *mod* operator delivers the remainder after an integer division and $C_0 \ldots C_m$ are constant generator parameters. Thus the generated item y_i depends on the m preceding items.

It follows that the maximum generated number is $M - 1$ and the generated random number sequence is periodical and reproducible. The periodicity and reproducibility are the reasons why this method is called *pseudo–random* number generation. To be able to reproduce the sequence of generated numbers it is an important feature in the process of debugging.

Programs may use the standard library function `rand()` for the purpose of random number generation. The new module offers, in addition to the functionality provided by the standard function, to create multiple instances of a generator, i.e. it is possible to use any number of independent sequences of random numbers. This feature can be utilized when separating the simulated system into several encapsulated parts. Only one sequence of random numbers is created in every part and the parts may be then debugged separately. Because of the reproducibility, all parts will behave exactly the same way in every simulation run. It is possible to add or remove any part of the system without influencing the generated random numbers in the other parts.

5.2.1 Data types

The generator is fully specified by

- generator state,
- congruent function.

The generator state comprises an array of twenty five 32-bit unsigned integers. Due to the dynamic memory management a new object, `CSIM_RNG_STATE`, was defined.

```
csim_derived_dyn_mem(csim_rng_state)
  CSIM_ULONG rng_array[CSIM_RNG_N];
  int index;
csim_end_derived(CSIM_RNG_STATE);
```

It is necessary to use the macro `csim_new_rng()` for memory allocation. The only argument this macro takes is the initialization value. The resources allocated by a generator are released by the macro `csim_dispose_rng()`.

```
#define csim_new_rng(seed)                                              \
  (CSIM_RNG_STATE *) csim_init_rng(                                     \
    (CSIM_RNG_STATE *) malloc(sizeof(CSIM_RNG_STATE)), seed);

#define csim_dispose_rng(rng)                                           \
  csim_dispose_dyn_mem((CSIM_DYN_MEM *) rng);
```

5.2.2 Functions

The congruent function `csim_rand()` returns 32-bit unsigned integers uniformly distributed in the $< 0, CSIM\_RAND\_MAX >$ interval. The argument to this function is pointer to

a CSIM_RNG_STATE instance. It is possible to retrieve a floating point number in the range $< 0, 1 >$ by dividing the returned value with the symbolic constant CSIM_RAND_MAX. For example:

```
double r_value;
CSIM_RNG_STATE *rng;

rng = csim_new_rng(1);

r_value = (double) csim_rand(rng) / (double) CSIM_RAND_MAX;
```

Moreover the module provides functions for the generation of random numbers with other than uniform distribution. These function are based upon the method of uniform-to-other distribution transformation, i.e. all use the function csim_rand() internally. The necessary argument to these functions is again the pointer to an instance of CSIM_RNG_STATE.

Function Prototypes:

```
CSIM_RNG_STATE *csim_init_rng(CSIM_RNG_STATE *p_rng,
                              CSIM_ULONG seed);
```

Initializes the specified RNG structure. Initial value seed must be nonzero.

```
CSIM_ULONG csim_rand(CSIM_RNG_STATE *p_rng);
```

Generates a single random number in the range $< 0, CSIM\_RAND\_MAX >$

```
double csim_negexp(CSIM_RNG_STATE *p_rng,
                   double lambda);
```

Generator of exponential distribution.

```
double csim_uniform(CSIM_RNG_STATE *p_rng,
                    double a,
                    double b);
```

Generator of the uniform distribution on the interval $< a, b >$.

```
CSIM_BOOLEAN csim_draw(CSIM_RNG_STATE *p_rng,
                       double p);
```

Returns TRUE with probability p (FALSE with probability $1 - p$).

```
double csim_gauss(CSIM_RNG_STATE *p_rng,
                  double sigma,
                  double center);
```

Generator of the normal (Gaussian) distribution.

5.3 Semaphore Module

One assumed kind of **C-Sim** application is a verification of parallel programs that use a kind of multithreading (i.e. *shared memory* interaction technique). Such programs require synchronization objects that are typically used in order to construct critical sections. This module provides a basic synchronization object – an *integer semaphore.*

The semaphore requires the following structures to operate properly:

- counter of available semaphore locks; a zero value signifies a locked semaphore
- queue of processes waiting for the release of a lock by an active process

The following operations upon a semaphore are supported:

- lock – if the value of counter is greater than zero, the counter is decremented by one and the process continues its execution. Otherwise the process is added to the end of the queue of waiting processes and its state is changed to passive[1].
- unlock – if the queue of waiting processes is not empty, the first process in the queue[2] is scheduled to current simulation time and removed from the queue. Otherwise the value of counter is incremented by one.

All operations upon a real semaphore must be atomic. Due to the pseudo–parallel execution of processes in **C-Sim**, the atomicity of operations is of no importance here.

5.3.1 Data Types

The data structure of an integer semaphore is derived from the `CSIM_HEAD` type.

```
typedef struct csim_semaphore {
  csim_d_head;
  CSIM_UWORD count;
} CSIM_SEMAPHORE;
```

The macro `csim_new_semaphore()` may be used to create an integer semaphore. The argument to this macro specifies the initial value of the semaphore counter (typically 1). The macro `csim_dispose_semaphore()` may be used to free all resources allocated by the semaphore.

```
#define csim_new_semaphore(cnt)                                          \
 csim_init_semaphore((CSIM_SEMAPHORE *) malloc(sizeof(CSIM_SEMAPHORE)), cnt);

#define csim_dispose_semaphore(sem)                                      \
  csim_dispose_head((CSIM_HEAD *) sem);
```

[1] lock is a blocking operation
[2] blocked by the lock operation

5.3.2 Functions

```
csim_lock_sem(p_sem);
```

Defined as a preprocessor macro.[3] Locks the given semaphore for the current process. If no lock is available, the process sleeps till it may continue.

```
void csim_unlock_sem(CSIM_SEMAPHORE *p_sem);
```

Unlocks the given semaphore. Any locked semaphore must be released after the protected resource is no longer needed to avoid a deadlock.

```
CSIM_SEMAPHORE *csim_init_semaphore(CSIM_SEMAPHORE *p_sem,
                                    CSIM_UWORD cnt);
```

Initializes the given semaphore.

5.4 Message Passing Module

This module implements the inter–process communication using the method of message passing. Communication can be classified into several groups:

- synchronous – the first participant to the communication waits until the second participant/process is ready to communicate, then, after the message transmission takes place, the processes continue independently.
- asynchronous – the sending process stores its message in a message queue and continues execution. The receiving process reads the message from this queue or waits for the message in the case where no message is ready.

The communication can be also divided by the type of addressing:

- symmetric – the information about the sender as well as the receiver is a part of the message.
- asymmetric – the message contains only information that identifies one of the communicating processes (typically the receiver).
- indirect – the message does not contain any address information. Both sender and receiver work anonymously with a message queue.

This module implements asynchronous communication with any type of addressing. The user may choose the appropriate addressing type by the means of an argument passed to the send and receive functions. Both the receiver and the sender may be uniquely specified by the pointer to their process record.

A message construction requires the following information to be specified:

[3] Uses internally the passivate operation. For details see paragraph 2.4.

- sender – this field is automatically set when sending the message. The receiver may explicitly define one particular sender[4] or use the symbolic constant `CSIM_ANY_MSG_SENDER`[5] to receive from any process.
- receiver – the sender may specify a single particular receiver for the message or use the symbolic constant `CSIM_ANY_MSG_RECEIVER` instead.
- type – is a 32-bit unsigned integer. The message will be delivered only if a bitwise AND operation upon sender and receiver type values gives a nonzero result. It follows that the type cannot be set to zero. The symbolic constant `CSIM_ANY_MSG_TYPE` may be used to receive messages of any type.

In this implementation the delivery of a message is defined as "passing of a pointer to an instance of `CSIM_MESSAGE` (or a derived type) from one process to another." The sending process has to create a dynamic instance of the message, initialize the contents and send the message. The receiving process receives and reads the message and then disposes the allocated dynamic memory.

To perform the message passing mechanism two queues are required: a message queue and a queue of waiting processes. The message queue contains messages with a well defined structure that is described later. The queue of processes contains records with the following information:

- sender – pointer to the process record of requested sender,
- receiver – pointer to the process record of receiver; this field may eventually serve to schedule a receiver process waiting for a message,
- type – requested message type
- pointer to a space reserved for the pointer to the message[6]

The items "sender", "receiver" and "type" have been already described above and they serve to specify all required information for the chosen type of addressing.

The following functions are provided for message passing:

1. send – searches the queue of waiting processes for the specified receiver. If a receiver is found, pointer to the message is written to the reserved space and the receiver is scheduled to current simulation time[7]. Otherwise the message is stored in the message queue.
2. receive – searches the queue of messages for a message matching the specified criteria. If a message is found, its pointer is written to the reserved space. Otherwise the process is added to the queue of waiting processes and its state is changed to passive[8].

[4] using a pointer to its process record
[5] in this implementation the value of `NULL`
[6] pointer to pointer to `CSIM_MESSAGE`
[7] the receiver was blocked by a call to receive
[8] receive is blocking operation

The message queue and waiting processes queue have to be created and initialized before the first utilization of message passing. After message passing it is no longer needed and the allocated resources must be disposed. These services are provided by the functions `csim_init_msg()` and `csim_clear_msg()`.

The initialization of this module must be postponed until the initialization of dynamic memory in **C-Sim**, i.e. after the function `csim_init_mem()` is called.

5.4.1 Data Types

The type `CSIM_MESSAGE` is derived from the type `CSIM_LINK` and thus allows to create a queue of messages easily. The type contains information for all possible kinds of addressing described above. Its definition is

```
#define csim_d_message          \
  csim_d_link;                  \
  CSIM_ULONG       msg_type;    \
  CSIM_PROCESS     *sender;     \
  CSIM_PROCESS     *receiver

typedef struct csim_message {
  csim_d_message;
} CSIM_MESSAGE;
```

The defined object `CSIM_MESSAGE` does not contain the transmitted data in any form, i.e. it is an "empty" message. New user-defined message types may be derived using the macro `csim_d_message`. It is also possible to use the macro

```
#define csim_derived_message(TYPE_NAME) typedef struct TYPE_NAME {          \
                                          csim_d_message;
```

The principle behind these macros is described in section 2.2. For generating objects of type `CSIM_MESSAGE` (or derived types) the macro `csim_new_message` is defined. To dispose a created object a similar macro `csim_dispose_message` may be used:

```
#define csim_new_message(TYPE_NAME)                                          \
 (TYPE_NAME *) csim_init_message((CSIM_MESSAGE *) malloc(sizeof(TYPE_NAME)));

#define csim_dispose_message(msg)                                            \
  csim_dispose_link((CSIM_LINK *) msg);
```

5.4.2 Functions

```
CSIM_RESULT csim_init_msg();
```

Initialization of the Message Passing module.

```
void csim_clear_msg();
```

Frees all resources allocated by this module.

```
csim_receive_msg(r_msg, r_msg_type, r_sender);
```

Defined as macro.[9] Starts the receiving of a message. The message is written into the `r_msg` argument, which is pointer to a message. In the long-jump implementation of **C-Sim** the actual argument cannot be a local (automatic) variable as the pointer would be destroyed.

```
void csim_send_msg(CSIM_MESSAGE *msg,
                   CSIM_ULONG msg_type,
                   CSIM_PROCESS *receiver);
```

Sends the specified message to the receiver process.

```
extern CSIM_MESSAGE *csim_init_message(CSIM_MESSAGE *msg);
```

Initializes any user-derived message with the default starting values.

5.5 Console Debug Module

This module provides multiple functions labeled `csim_view_...()` which are used to display the current state of an object on the standard output device.[10] The user has to write the address of the appropriate function into the field `view` of every object, best immediately after the creation of the object, e.g.:

```
MY_CSIM_LINK *link;

link = csim_new_link(MY_CSIM_LINK);
link->view = csim_view_link;
```

The function `csim_debug()` allows the user to interactively browse the list of allocated dynamic memory and to display the state of every object using its `view` function. The argument `p_dyn` passed to the function is pointer to the first displayed object. If this argument is equal to `NULL`, the first object in the dynamic memory list to be displayed.

```
void csim_view_dyn_mem(void *p_void);
```

Shows the `CSIM_DYN_MEM` object.

[9] Receiving can block process. More details can be found in section 2.4

[10] using the standard `printf()` function

```
void csim_view_link(void *p_void);
```

Shows the CSIM_LINK object.

```
void csim_view_head(void *p_void);
```

Shows the CSIM_HEAD object.

```
void csim_view_process(void *p_void);
```

Shows the CSIM_PROCESS object.

```
void csim_debug(CSIM_DYN_MEM *p_dyn);
```

Interactive function for viewing the whole list of dynamically created objects at the time point of its call.

Chapter 6

Basic Rules How to Use the C-Sim Library

6.1 Structure of Simulation Program

A simulation program has the structure like any other program written in **C** language. Because the usage of the **C-Sim** library macros somewhat changes the source code, it is recommended, at least for the initial experiments, to use the following structure of the program.

1. At the beginning of the source code the compiler directive is to be placed:

   ```
   #include "csim.h"
   ```

   Inside the csim.h file may be further `#include` directives, through which other needed header files[1] are included. Header files of all optional **C-Sim** modules that the program uses need to be included.

2. Definitions of the used data types follow. They are mostly derived from basic **C-Sim** object types supported by the library (see 2.2). It is naturally possible to use also other user–defined types.

3. The definition of global data of the simulation model. These are the variables accessible from all processes of the model. They are mostly parameters and pointers to stable objects of the model.

4. Operations upon the introduced object types. Because there are no resources for assigning these operations to the given object types (e.g. with an encapsulation construction), it is recommended to mark them at least with a comment. Those operations have the form of **C** language functions of any type, but one parameter should be a pointer to the object type that the operation belongs to.

5. Definitions of programs of individual processes of the simulation model. Programs of the processes are constructed via the `csim_program`, `csim_end_process` and

[1] The header files of the standard **C** language libraries and necessary **C-Sim** headers.

`csim_end_program` macros, for details see section 3.5. Internally, those programs have the form of **C** language functions. A program which can be used by several processes must be constructed as reentrant.

It means that all the data items belonging exclusively to a process (i.e. its local data) should be defined as the items of the process data type. Within the program (C language function) connected with this type the local data items must be accessed only by means of operators `my` *or* `p_my`*, e.g.* `my.item` *or* `p_my->item` [2].

6. Definition of arbitrary functions that can be used for result printing, initialization and so on. It is adequate to define a function for model initialization. The initialization function should contain the necessary initial setting of all global data, creation and initialization of stable objects[3] and at least one statement, which will activate at least one of the created processes.

   It is recommended to check results of ***C-Sim*** *functions calls within the program initialization part (see examples in chapter 7), because the* ***C-Sim****'s standard exception mechanism (i.e. return from* `csim_step()`*) doesn't work here.*

7. Finally we introduce the definition of the main() function, which controls the simulation run. This function contains the basic simulation loop that triggers the scheduled processes using the csim_step() function from **C-Sim** library. End of this loop needs to be user-constructed (e.g. after reaching a chosen value of model time) is equivalent to the end of one simulation experiment.

8. In the simulation program several simulation experiments can be performed, according to an arbitrary decision procedure. Every experiment should be started with csim_init_mem() function call to initialize the dynamic memory of the application and finished with the sim_clear_mem() function call to release the dynamic memory. After memory initialization the model initialization function should be called (with parameters changed for a new experiment run).

6.2 Process's Operations Usage

There are some rules for using operations described in sections 4.3 and 4.4.

- The `csim_step()` function may be called only outside processes' programs, it is called typically in the basic loop within the `main()` function (see examples in 7).
- `csim_hold`, `csim_passivate` and `csim_wait` macros may be called only in some process's program.
- Functions for activation of processes may be called both inside and outside process's program. Inside the process's program their call doesn't influence the state of the currently active process.

[2] When we are using thread-based implementation of pseudo-parallel processes it suffices to define local data items as automatic variables within the program function. But in order to preserve the program portability between **C-Sim** implementation versions it is better to keep the above stated convention.

[3] A stable object is here an object with "life length" equal to the duration of the simulation experiment.

- Other functions may be used without limitations.

6.3 Long-Jumps Based Implementation Restrictions

The long-jump implementation of pseudo-parallel processes causes corruption of stack during each context switch. This means that the following entities are invalidated in a function representing process's program:

- local variables of **C** class *auto*
- parameters
- return address

Solutions to these problems are:

- Local variables should be used only temporarily for computation between context switches. Other data should be located in the process's record which is located on the heap – in the process's program they can be accessed through `p_my->data` or `my.data`.
- Parameters should be accessed only before the first context switch.
- Macros for the process context switching should be called only at the level of function representing process's program.
- The `csim_step()` function should be called within the main program function all the time at the same level of nesting.

6.4 Backward compatibility

The changes to **C-Sim** were applied with regard to best possible backwards compatibility with the previous library version. Occasional incompatibilities are caused by the separating of optional modules and by adding of several new properties. Namely:

- An object does not have a `view` function attached by default. The `csim_view_...()` and `csim_debug()` functions were moved into the Console Debug module. The reason is that simulation programs often use a visualization layer and standard console input and output is not available.
- Due to the addition of the random number generator, the prototypes of functions for generating numbers of various distributions had to be changed. Compared to the previous version, these functions take as argument an additional pointer to object `CSIM_RNG_STATE`. The benefit of this solution is the possibility to create an arbitrary number of independent random number sequences.
- Due to the separation into multiple modules, the user has to include the appropriate header files using the `#include` preprocessor directive.

- The method of generating an error code has been redesigned. The numerical values of error code as well as the symbolic error names do not match with the previous version. The conversion of an error code to an error description is handled by the function `csim_error_msg()` from Error Messages module.

Moreover, all exported identifiers were renamed by adding the prefix `csim_...`. For an easy translation of existing programs, which use **C-Sim** version 4.1 or older, it is possible to:

- use the supplied text-filter on the original source codes, translating them[4] with good probability into sources compatible with **C-Sim** version 5.1, or
- include a header file for backward compatibility. This header file defines the old identifiers as macros which use the new identifiers instead.

6.5 C++ compatibility

Generally it is possible to use the **C-Sim** library within any ANSI **C++** computational environment, the only limitation is that programs of processes must be **C++** independent functions (i.e. they can't be member functions of classes) so it is not possible to construct process-like classes (like e.g. threads in Java). **C++** utilization is useful especially when we need to construct a GUI for a simulation model (e.g. in order to visualize an experiment run).

[4] replacing the strings in source codes can cause "side effects"

Chapter 7

Demonstration examples

7.1 Model of a Queuing Network

This example demonstrates C-Sim library utilization in the traditional area of discrete-time simulation. The modeled open queuing network is composed of two servers with FIFO queues of transactions waiting for service. There are two sources of new transactions and two input points for them. There are also two output points – see figure 7.1.

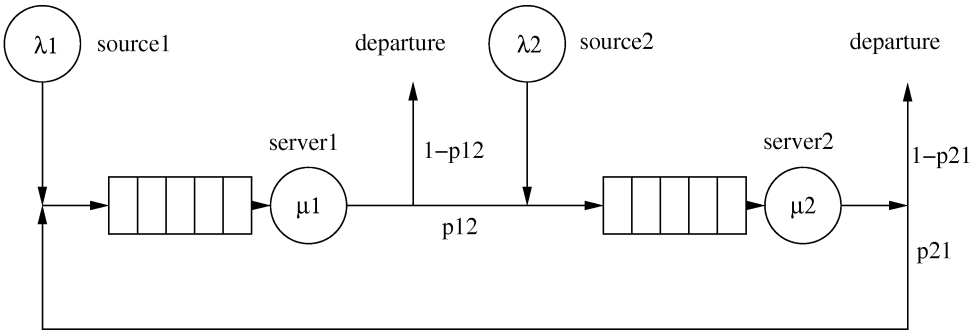

Figure 7.1: Queuing network

Interarrival time in input streams of transactions and service time of servers are exponentially distributed. Thus the corresponding parameters of the simulation model are as follows:

λ_1 – mean frequency of the first input stream (and the parameter of the exponential distribution of interarrival time),

λ_2 – mean frequency of the second input stream,

μ_1 – parameter of the exponential distribution of the first server service time (or the mean conditional frequency of services),

μ_2 – parameter of the exponential distribution of the second server service time.

Additional parameters are branching probabilities which describe passing of transactions through the network:

p_{12} – probability of transactions passing from node 1 to node 2 after being served in the node 1 (and with complementary value $1 - p_{12}$ transactions depart the network),

p_{21} – probability of transactions passing from node 2 to node 1 after being served in the node 2 (and with complementary value $1 - p_{21}$ transactions depart the network),

The output information of the model should be the response time T_q, i.e., the mean time which a transaction spends inside the network.

7.1.1 Object types

Transaction

There are two possible ways of transaction modeling. The "active" principle means to take the transaction as a process-like object. In the "passive" principle (used in this example) a transaction is handled as an object without its own "activity" and is created, destroyed and inserted into queues by other processes. That is why the type `TRANSACTION` is derived from the standard `CSIM_LINK` type. An additional attribute of `TRANSACTION` is the time of the transaction generation.

```
typedef struct {
  csim_d_link;
  double t_input;     /* "birth" time of transaction */
} TRANSACTION;
```

Queue

The queue can be derived from the supported `CSIM_HEAD` type which implements a two-way circular list. The one additional attribute of `QUEUE` type is the binding (i.e. a pointer) to the object–server which is taking transactions out of the queue:

```
typedef struct {
  csim_d_head;
  CSIM_PROCESS *server;  /* pointer to server */
} QUEUE;
```

Source

The source of transactions is evidently an "active" object, which generates a Poisson stream of transactions. It means that the type `SOURCE` must be derived from the supported type `CSIM_PROCESS`. Added attributes are λ–parameter of the Poisson stream and binding to the queue where generated transactions are stored:

```
typedef struct {
  csim_d_process;
  float lambda;  /* parameter of the pdf of interarrival time */
  QUEUE *queue;  /* pointer to output queue */
} SOURCE;
```

The type SOURCE describes the local data record of an instance of the process. Furthermore it is necessary to define a program of process' "life". Due to the fact that the program should be the same for all instances of source-like objects, it has to be constructed with the property of reentrancy.

```
csim_program(SOURCE, SOURCE_PROG)
  TRANSACTION *p_poin;
  /* temporary  pointer to a generated transaction */

  for (;;) {
    p_poin = csim_new_link (TRANSACTION);
    p_poin->t_input = csim_time();
    csim_into ((CSIM_LINK *) p_poin, (CSIM_HEAD *) my.queue );
    if (csim_idle (my.queue->server)) {
      csim_activate_at (my.queue->server, csim_time());
    }
    csim_hold(csim_negexp(rng, my.lambda));  /* interarrival time */
  }
csim_end_program
```

Server

The last (and the most complex) required type is SERVER. This type describes the data of a server–like object. Server introduces evidently an "activity", so we derived its data type from the CSIM_PROCESS. The meaning of several added attributes is explained in comments.

```
typedef struct {
  csim_d_process;
  float mi;          /* rate of service, exponential pdf */
  float p;           /* prob. of departure after the service */
  QUEUE *in_queue;   /* pointer to an input queue */
  QUEUE *out_queue;  /* pointer to an output queue */
  CSIM_UWORD cnt;    /* counter of serviced transactions */
  double stq;        /* sum of responses of finished transactions */
} SERVER;
```

7.1.2 Global data items

The model parameters and pointers to stable objects are declared as global data:

```
QUEUE *queue1,*queue2;    /* pointers to queue 1 and 2 */
SERVER *server1,*server2; /* pointers to server processes 1 and 2 */
SOURCE *source1,*source2; /* pointers to source processes 1 and 2 */

float mi1     = 1.0;      /* service rate of server 1 */
float mi2     = 1.0;      /* service rate of server 2 */
float lambda1 = 0.4;      /* arrival rate from source 1  */
float lambda2 = 0.4;      /* arrival rate from source 2  */
float p12 = 0.5;           /* prob. of trns. passing from 1 to 2 */
float p21 = 0.5;           /* prob. of trns. passing from 2 to 1 */
CSIM_RNG_STATE *rng; /* pointer to an object of random numbers generator  */
```

7.1.3 Model initialization

In the initialization function of the model, stable objects are constructed and source processes are planned.

```
/* Function for the model initialization */
CSIM_BOOLEAN init (void) {
 /* csim's data initialization */
  if (csim_init_mem()) return TRUE;
      /* TRUE means a run-time error within the C-Sim function */
 /* stable objects creating and initializing */
  rng = csim_new_rng(INIT_SEED);
  if (rng == NULL) return TRUE;
  queue1  = csim_new_head (QUEUE);
  if (queue1 == NULL) return TRUE;
       ...

  server1 = csim_new_process(SERVER, SERVER_PROG);
  if (server1 == NULL) return TRUE;
  ini_server (server1, mi1, p12, queue1, queue2);

  /* initial activation of processes */
  if (csim_activate_at ((CSIM_PROCESS *) source1, 0.0))
    return TRUE;
      ...
  return FALSE;
}
```

7.1.4 Main program

First, in the main program the initialization of dynamic memory and of the model is performed. Then the main simulation loop is executed using `csim_step()` function that executes one activity of the first process in the calendar. The final condition of the loop execution is derived from a number of passed transactions.

```
int main(void)
{
  CSIM_ERROR er;
  long int trns_cnt = 0;

  if (init()) {
    csim_error(&er);
    printf("Init: %s\n", csim_error_msg(er.error_code));
    (void) getchar();
    csim_clear_mem();
    return (0);
  }

  printf ("OQN - open queuing network model \n");
  printf ("Number of passing transactions (100 000 should pass): \n");
  while(trns_cnt < 100000) {
    if (csim_step() == FAILURE) {
      csim_error(&er);
      printf("Loop: %s\n", csim_error_msg(er.error_code));
      (void) getchar();
      csim_clear_mem();
      return (0);
    }
    trns_cnt = server1->cnt + server2->cnt;
    printf ("%li\r", trns_cnt);
  }
```

Note: You can try to modify the example - at first to use more nodes of the `SERVER` type connected within a network (here you should carefully estimate model parameters to get stationary behavior of the model). The second recommended modification is to use more complicated type of server - e.g. with its output branching to several other nodes.

7.2 M/M/1 queuing system

Within the previous example (queuing network, two nodes), all the objects (including processes) that is the model composed from, are stable. It means that their lifetime is the same as the lifetime of the simulation program.

This demonstration example shows a model of M/M/1 queuing system. Passing transactions are treated as dynamically created (and detached) processes. Parameters of the queuing system model are as follows:

- *lambda*: rate of arrivals (i.e. the parameter of exponential pdf of interarrival time, set value 1.0),
- *mi*: rate of service (i.e. the parameter of exponential pdf of service time, set value 0.8),

- *one server*,
- *FIFO queue, unlimited length.*

The aim of the model construction is to estimate the mean response time (i.e. the mean time of the transactions passing through the queuing system). It is possible to change the number of passing transactions and to follow a precision of the experimental result by comparison with its theoretical value ($T_q = 5.0$ for the chosen values of parameters).

The data record of the process that models a transaction passing through the system is as follows:

```
typedef struct {
  csim_d_process;         /* derived from CSIM_PROCESS  */
  double t_input;            /* transaction's "birth time" */
} TRANSACTION;
```

The model contains one process that generates transactions (stable object, name `source`) and one list (type `CSIM_HEAD`, name `queue`) that contains all the processes of the TRANSACTION type (including the served one). Common program for these processes:

```
csim_program(TRANSACTION, TRNS_PROG)
  my.t_input = csim_time();
  if (csim_empty(queue))   /* test of queue */
    csim_into ((CSIM_LINK*)csim_current(), queue);
    /* process inserts itself into the queue, nonblocking call */
  else
    csim_wait(queue);      /* blocking call - like passivate() */
  csim_hold(csim_negexp(rng, mi));       /* models the service */
  csim_out ((CSIM_LINK*) csim_current());/* process departs the queue */
  if (!csim_empty(queue))  /* awakes next proces in queue - if any */
    csim_activate_at ((CSIM_PROCESS*) csim_first(queue), csim_time());
    ... /* statistics */
csim_end_process           /* deletes the process data record */
```

Clearly, this kind of the model organization (i.e. transactions treated as processes) can be less effective due to the overhead with dynamically created and detached processes. Moreover to debug a model with dynamic processes is much worse than in a case where processes are stable.

7.3 Model of Shared Resource Utilization

This example presents a model of a simple parallel algorithm executed at a shared–memory multiprocessor. Several processes (with the same program) periodically utilize a block of shared data. They use a semaphore with conventional `lock` and `unlock` operations to synchronize access to the data. The synchronized part of the program executed with all the processes is being denoted as *critical section.*

In this example we demonstrate the use of the Semaphore module and furthermore the creation of two mutually independent sequences of pseudo–random numbers, the first sequence for the modeled parallel algorithm itself, the second for modeling the duration of computations.

The type (named `WORKER`) of processes working with the critical section has to be derived from the supported type `CSIM_PROCESS`:

```
typedef struct {
  csim_d_process;
  float mi;       /* 1/mi is the mean time spent inside crit. sec.     */
  float lambda;   /* 1/lambda is the mean time spent out of crit. sec. */
  CSIM_SEMAPHORE *p_sem;     /* pointer to synchronizing semaphore     */
  double rslt;    /* local result of one iteration of computation      */
} WORKER;
```

The local computation part of worker's program generates a local result (emulated with a generated random number) and the critical section computes the contribution of the local result to the global one. The program code follows:

```
csim_program(WORKER, WORKER_PROG)
  for (;;) {  /* infinite loop */
    /* local part of iteration */
    /* a computation - here a random number generation */
    my.rslt = csim_negexp (comp_rng, 0.1);
    /* simulation of a time of local computation */
    csim_hold (csim_negexp(time_rng, my.lambda));

    /* critical section */
    csim_lock_sem(my.p_sem);      /* locking the section */
      /* computation of global "sliding" average using local result  */
      rslt = 0.95*rslt + 0.05*my.rslt;
      cs_count ++;   /* increment of glob. count of iterations */
      /* csim_hold() emulates time spent inside crit. section */
      csim_hold (csim_negexp(time_rng, my.mi));
    csim_unlock_sem(my.p_sem);
  }
csim_end_program
```

The initialization of the model is executed in the `main()` function in this way:

```
  p_sem  = csim_new_semaphore(1);  /* 1 means "open" */
  time_rng = csim_new_rng(INIT_SEED);
  comp_rng = csim_new_rng(INIT_SEED+1);

  for (i=0; i<N; i++) {
    p_workers[i] = csim_new_process (WORKER, WORKER_PROG);
    ini_worker(p_workers[i], mi, lambda, p_sem);
```

```
    csim_activate_at ((CSIM_PROCESS *)p_workers[i],
      csim_negexp(time_rng, lambda));
}
```

This example demonstrates that with a single **C-Sim** based model it is possible:

- to validate a deadlock-free control flow of model-implemented parallel algorithm (with a good reliability that depends on a length of testing - here on the number of algorithm's modeled cycles),
- to validate the time-independence of the result (i.e. deterministic behavior of the implemented program),
- to validate the logical correctness of results (here the computed mean value of generated random numbers should not oscillate too much),
- to determine the chosen performance parameters of the algorithm (for a chosen model of its dynamic behavior and optionally for a dynamic model of its operational environment).

Note 1: The used modeling methodology replaces parallel computation of several threads with serialized (interleaved) pseudo-parallel computation of several discrete-time simulation processes. Using this example it is possible to validate that the replacement works well. We can compare the measured (i.e. obtained from simulation model) frequency of periodical computation with the theoretical value obtained from mathematical model (stochastic Petri net, Markov model) that assumes real concurrency - see the results printed as the program output.

Note 1: Experimentally validated programs of processes can be separated from the simulation program and in a straightforward way adapted for a real computational environment (with semaphore operations in its interface, e.g. POSIX threads). It has apparently no great sense for this simple case, but we can start the experimentation with more sophisticated synchronization schemes what requires more semaphore instances and a danger of deadlock is more significant.

7.4 Distributed election algorithm

This example shows a model that serves as an experimental validation of t-resilient asynchronous distributed election with fault injection before the voting. Asynchronous means, that there is no upper limit for a message delivery and/or processing time.

There is n processes, t of them can be out-of-function (but the others don't know who isn't alive). The processes try to elect one leader (king) among them. Processes use asynchronous message-passing kind of interaction, i.e. the `send()` operation is nonblocking in the contradiction to the `receive()` operation. The election is started by a group of k processes from n ($k > t$), i.e. k processes send a message "to start the election", but not necessarily at the same time.

More detailed description of the used algorithm is contained in the file `kings.ps` within the **C-Sim** 5.1 standard release. Every process acts as a state machine, every incoming (processed) message can be a cause for a state-space transition. The (common) program for all processes that take part in the election (denoted as "fighting kings") is relatively simple:

```
csim_program(P_DATA,P_PROG)
 /* initialization of local data */
 my.state_p = slave;      /* state */
 my.master = 0;           /* master's id - when "defeated" */
 my.waiting = 0;          /* id to wait a response from */
 my.next = my.id;         /* next id to send message */

 /* infinite loop - election can be repeated */
 while (TRUE) {
   /*  process waits for a message */
   csim_receive_msg(my.msg, CSIM_ANY_MSG_TYPE, NULL);
   /* hold() simulates the random time of a message processing
      (and the message delivery as well) */
   csim_hold (csim_negexp(rng, lambda));
   /*  state-space transition function is called according to
       the message type and the actual state of the process */
   (* switch_table [my.msg->m_type] [my.state_p])((P_DATA*)csim_current());
 }
csim_end_program
```

Table that contains pointers of functions that perform the state-space transitions is as follows:

```
/* state transition table */
void (* switch_table[7][7])(P_DATA *) = {
       /* search battle defeated  relay  waiting  leader  slave*/
/*join_1*/ {  a1,     a1,     a1,     a4,    sus,    del,    del },
/*join_2*/ {  a2,     a2,     a3,     a3,     a3,    del,    del },
/*accept*/ { acc,     a6,    del,    del,     a8,    del,    del },
/*reject*/ {  a5,     a7,    del,    del,     a9,    del,    del },
/*leader*/ { ldr,    ldr,    ldr,    ldr,    ldr,    del,    del },
/*wakeup*/ { wak,    del,    del,    del,    del,    del,    del },
/*reset */ { res,    res,    res,    res,    res,    res,    res }};
```

Rows of the table correspond to the type of message and columns of the table means possible states of a process during the election. Within the table there are names of functions that perform a transition. All these functions are of the same type - they have one parameter that points to the data record of a process which is the function operating with.

The main program enables to choose how many election trials should be done. Every trial runs with different initial setting of random numbers generator (i.e. the setting that was the previous trial finished with), so the time sequences of messages processing within a trial are generally different and the algorithm is tested properly (a deadlock condition is tested at the

end of every trial). But the program still behaves deterministically (at the beginning of the first trial the random number generator is initialized the same way), what means that we are able to repeat the computation and analyze a trouble that occurs - say in the election trial number 327.

To demonstrate the possibilities of the deterministically serialized parallel computation, the main program enables to step its main loop manually (Enter key). One step means one process single activity (performed at single point of the model-time). Moreover all the messages are displayed together with the state of process that makes their processing.

Note 1: Instead of the demonstrated simple console-aimed user interface of the simulation program it is possible to construct arbitrarily complex graphical user interface (including a kind of the model activity visualization). For this purpose it is possible to utilize any ANSI **C++** graphic library.

Note 2: Here we didn't use a precise model of the communication system. A delay of a message delivery is (logically) composed with a delay of the message processing using one exponentially distributed random number (see `hold()` pseudo-statement in the program above). It is possible to construct more precise object-oriented model of communication (possibly including active objects, i.e. processes), to distinguish various reasons for delays (i.e. to use more hold() statements), to use other random numbers distributions, etc. Then some chosen performance parameters of the algorithm (including a system that the algorithm is executed on) can be investigated as well.

Note 3: Here we used **C-Sim**'s module that implements asynchronous message passing operations. When the semantics of interaction operations doesn't fit the case to be investigated, it is possible to construct a module that exports another set of operations (possibly including their internal delay) according to the patterns given in the Semaphore or Message passing modules.

7.5 Open queuing network parallelized using PVM tool

This is a version of the standard example (see file `oqn.c`) that is adapted to run under **PVM** using so called farmer-workers model.

Queuing network consists of two serving nodes (FIFO queue and one-channel server with exponential pdf of service time) and from two transaction sources with exponential pdf of interarrival time. The aim of model construction is to estimate the mean response time T_q (i.e. the mean time that transactions need to pass through the network).

The file `oqn_farmer.c` contains the executable program of farmer-type process (here the word process is used in the sense of running program, i.e. not a discrete-time executed pseudo-parallel simulation process). Only one process should run this program and no **C-Sim** utilities are used here. The farmer-type program at first reads (from keyboard) the number of transactions to be passed through the modeled network. Then it creates worker processes and repeatedly (message `CMD_CONTINUE`) sends a number of transactions that a worker should pass through its instance of the modeled system. It registers an overall number of processed transactions and when the requested number was reached, it computes an overall statistics,

prints the results and releases workers.

The second file `oqn_worker.c` contains the program of worker-type process. An arbitrary number of workers can run this program at the same time. The worker-type program repeatedly receives a number of transactions that should pass through the modeled system. At first it runs from the model time value `csim_time=0.0` and the modeled queuing network is empty. At the second, etc., it runs from the previously reached values of model time and network's state. It returns (message `CMD_DATA`) partial result - how many transactions it processed together (i.e. from the beginning of computation) and the value of `Tq` that it reached.

Random number generators of worker processes are set differently at the beginning (using PVM's task ID), so every worker yields statistically independent results that can be composed (in the farmer process) to get the final statistics. The shadow side of this method is nondeterministic computation of the simulation model due to the fact, that tasks ID are different for different runs of computation. So at first we should debug the simulation model properly using its nonparallelized version (here `oqn.c`).

When a worker suffers a csim run-time error, it returns error code instead of partial result. Farmer deletes the failed worker from the stuff and continues its work. At least one worker should stay alive to complete all the computation.

Structure of used messages is described in the file `oqn_pvm.h` together with symbolic constants used as parameters of the model computation.

Note: This example mainly demonstrates the possibility to use both the (**C**-language aimed) **C-Sim** and **PVM** interfaces together within one application. Moreover it shows a way how to seedup the computation of a category of simulation applications using standard parallelization model (i.e. farmer-workers) and a standard tool for parallel computation (here **PVM**, but e.g. **MPI** can be used similarly).

Bibliography

[1] J. Hlavička, S. Racek, P. Herout (1999): C-Sim v.4.1, Research Report CD-99-09, http://www.c-sim.zcu.cz

[2] J. Kačer (2001): J-Sim, Java Based Tool for Discrete Simulations, Diploma thesis

[3] WWW: DECthreads Interface Documentation, http://www.tru64unix.compaq.com

[4] WWW: Simula Language, http://www.simula.com

[5] WWW: PVM http://www.csm.ornl.gov/pvm/

Appendix A

C-Sim Library Interface

```
/*****************************************************************************
            University of West Bohemia, DCS, Pilsen, Czech Republic
                                (c) copyright
                                  03.04.2003

                               C-Sim version 5.1

                                  file csim.h

                          header file of C-Sim library
                    (all exported functions and global variables)

*****************************************************************************/

#ifndef CSIM_H
#define CSIM_H 1

#ifndef KR_PATH
#define KR_PATH "kr_jmp/csim_kr.h"
#endif

#include <stdlib.h>
#include "csim_dt.h"

#include KR_PATH
#include "csim_tm.h"


/*****************************************************************************
                                    used macros
*****************************************************************************/

/*
 *  Use following macros like this:
 *
 *  typedef struct my_link {
 *    csim_d_link;
```

```
 *      ...
 *    user attributes;
 *      ...
 *  } MY_LINK;
 */

/*
 * Class for block of dynamic memory
 * name       - used by the debug function - user can use this to identify
 *              object
 * type       - object type
 *              P ... process
 *              L ... csim_link
 *              H ... csim_head
 *              M ... dynamic memory block
 * check      - pointer used for checks
 * p_head_mem - pointer to head of list of dynamically created objects
 * p_suc_mem  - pointer to successor in the list
 * p_pred_mem - pointer to predecessor in the list
 * destructor - pointer to object's destructor
 * view       - function for viewing object's data
 */

/*
 * Macro for derivation from the CSIM_DYN_MEM type
 * class of dynamically generated objects which are
 * inserted into one list
 */

#define csim_d_dyn_mem               \
  char                 *name;        \
  char                 type;         \
  char                 *check;       \
  struct csim_dyn_mem  *p_head_mem;  \
  struct csim_dyn_mem  *p_suc_mem;   \
  struct csim_dyn_mem  *p_pred_mem;  \
  CSIM_DESTRUCTOR      destructor;   \
  CSIM_VIEW            view

/*
 * Class for objects insertable into list
 * suc        - successor in the list
 * pred       - predecessor in the list
 * head       - head of the list
 * enter_time - time of the first insertion to the queue
 */

/*
 * Macro for derivation from the CSIM_LINK type objects insertable
 * into the CSIM_HEAD list
 */
#define csim_d_link                  \
  csim_d_dyn_mem;                    \
```

```
  struct csim_link     *suc;         \
  struct csim_link     *pred;        \
  struct csim_head     *head;        \
  CSIM_TIME            enter_time

/*
 * Class for list head objects, also insertable into list
 * p_first     - pointer to the first item in the list
 * p_last      - pointer to the last item in the list
 * lq          - actual list length
 * arrival_cnt - number of items inserted into the list during the time
 *               statistics was turned on
 * last_time   - time of the last change of list length
 * sum_tw      - sum of the item's waiting times
 * sum_lw      - sum of the actual list lengths weight by the time their last
 * sum_time    - total time the statistics was on (used to count Lw)
 * statistics  - enables/disables the statistics
 *             - if the global variable statistics is TRUE, the statistics
 *               are evaluated always
 *             - if the global variable statistics is FALSE, the statistics
 *               are only evaluated when this attribute is TRUE
 */

/*
 * Macro for derivation from the CSIM_HEAD type the head of the list
 */
#define csim_d_head                  \
  csim_d_link;                       \
  CSIM_LINK       *p_first;          \
  CSIM_LINK       *p_last;           \
  CSIM_UWORD      lq;                \
  CSIM_ULONG      arrival_cnt;       \
  CSIM_TIME       last_time;         \
  CSIM_TIME       sum_tw;            \
  CSIM_TIME       sum_lw;            \
  CSIM_TIME       sum_time;          \
  CSIM_BOOLEAN    statistics

/*
 * Class for process objects
 * program          - pointer to program of the process. It is run after the
 *                    process was started (e.g. is in state NEW_...)
 * suc_in_sqs       - pointer to the next process in scheduler
 * pred_in_sqs      - pointer to previous process in scheduler
 * state            - state of the process
 * evtime           - time of the process's activation. It determines position
 *                    of the process in scheduler
 * last_time        - the time when the process was last inserted into
 *                    scheduler
 * non_passive_time - sum of time intervals when the process was scheduled
 * passive_time     - sum of time intervals when the process was passive
 * statistics       - enables/disables the statistics
 *                  - if the global variable statistics is TRUE, the statistics
```

```
 *                    are evaluated always
 *                  - if the global variable statistics is FALSE, the statistics
 *                    are only evaluated when this attribute is TRUE
 */

/*
 * Macro for derivation from the CSIM_PROCESS type object of one process
 */
#define csim_d_process                    \
  csim_d_link;                            \
  csim_process_kr;                        \
  CSIM_PROC_PROGRAM    program;           \
  struct csim_process  *suc_in_sqs;       \
  struct csim_process  *pred_in_sqs;      \
  CSIM_PROC_STATE      state;             \
  CSIM_TIME            evtime;            \
  CSIM_TIME            last_time;         \
  CSIM_TIME            non_passive_time;  \
  CSIM_TIME            passive_time;      \
  CSIM_BOOLEAN         statistics

/*
 * The macros with the same function as above, but with better readability
 *
 * Use like this:
 * csim_derived_link(my_link)
 *     ...
 *   user attributes;
 *     ...
 * csim_end_derived(MY_LINK);
 */


/*
 * !!!!!!!!!!!!!!!!!!!!!!!! WARNING !!!!!!!!!!!!!!!!!!!!!
 *
 *  The parameter TYPE_NAME (macros derived_xyz(TYPE_NAME)
 *  and csim_end_derived(TYPE_NAME)) are the names of one and
 *  the same structure. Please use
 *
 *  lower case (like type_name) when using macro derived_xyz
 *  it is used by the structure for selfreferencing
 *  upper case (like TYPE_NAME) when using macro csim_end_derived
 *  it is used as structure's name
 *
 *  Example:                          Expands to:
 *  csim_derived_link(my_link)        typedef struct my_link {
 *      ...                             csim_d_link;
 *    my_attributes;                    my_attributes;
 *    struct my_link *p_my_link;        struct my_link *p_my_link;
 *      ...                               ...
 *  csim_end_derived(MY_LINK)         } MY_LINK;
 */
```

```
#define csim_derived_dyn_mem(TYPE_NAME) typedef struct TYPE_NAME {            \
                                    csim_d_dyn_mem;

#define csim_derived_link(TYPE_NAME) typedef struct TYPE_NAME {               \
                                  csim_d_link;

#define csim_derived_head(TYPE_NAME) typedef struct TYPE_NAME {               \
                                  csim_d_head;

#define csim_derived_process(TYPE_NAME) typedef struct TYPE_NAME {            \
                                    csim_d_process;

#define csim_end_derived(TYPE_NAME) } TYPE_NAME


/*
 * Macros for dynamic generation of new objects
 *
 * TYPE_NAME - name of the type - CSIM_LINK, MY_LINK, CSIM_HEAD, MY_HEAD ...
 * PROC_PROG - pointer to program of the process
 *
 * Usage:
 * csim_derived_process(my_process)
 *             ...
 * csim_end_derived(MY_PROCESS);
 *
 * void prog_of_process(void) {
 *             ...
 * }
 *
 * void main(void) {
 *   MY_PROCESS *p_proc;
 *
 *   p_proc = new_process(MY_PROCESS, prog_of_process);
 *             ...
 * }
 */

/* Generates new object derived from CSIM_DYN_MEM */
#define csim_new_dyn_mem(TYPE_NAME)                                           \
  (TYPE_NAME *) csim_init_dyn_mem((CSIM_DYN_MEM *) malloc(sizeof(TYPE_NAME)))

/* Generates new object derived from CSIM_LINK */
#define csim_new_link(TYPE_NAME)                                              \
  (TYPE_NAME *) csim_init_link((CSIM_LINK *) malloc(sizeof(TYPE_NAME)))

/* Generates new object derived from CSIM_HEAD */
#define csim_new_head(TYPE_NAME)                                              \
  (TYPE_NAME *) csim_init_head((CSIM_HEAD *) malloc(sizeof(TYPE_NAME)))

/* Generates new object derived from CSIM_PROCESS */
#define csim_new_process(TYPE_NAME, PROC_PROG)                                \
```

```
  (TYPE_NAME *) csim_init_process(                                          \
    (CSIM_PROCESS *) malloc(sizeof(TYPE_NAME)), (PROC_PROG))

/*
 * Macros that initialize the attributes of CSIM_DYN_MEM type
 *
 * CSIM_DYN_MEM *P_DYN_MEM    - pointer to object
 * char *NAME                 - object's name (identification string)
 * CSIM_DESTRUCTOR DESTRUCTOR - pointer to object's destructor
 * CSIM_VIEW VIEW             - pointer to function used to view the object
 *                              (is of type CSIM_VIEW)
 */

#define csim_set_destructor(P_DYN_MEM, DESTRUCTOR)                          \
  (P_DYN_MEM)->destructor = (DESTRUCTOR)

#define csim_set_view(P_DYN_MEM, VIEW) (P_DYN_MEM)->view = (VIEW)

/* Macro used to access object's attributes */
#define my (*p_my)

/* Macro used to generate error code */
#define CSIM_ERR_CODE(func, err) (CSIM_UWORD) ((err << 8) | func)

/*
 * Next four macros are working with active process and are changing it's state:
 *   - csim_hold(del_t)   : changes the state of process to CSIM_PLANNED
 *                          and schedules it to current time plus del_t
 *   - csim_passivate()   : changes the process to state PASSIVE
 *   - csim_wait(p_head)  : changes the process to state PASSIVE and inserts it
 *                          into list p_head
 *   - csim_end_program   : ends program code
 */

#define csim_hold(del_t)                                                    \
do {                                                                        \
  CSIM_PROCESS *_p_process;                                                 \
  CSIM_TIME _csim_dt;                                                       \
  _p_process = csim_current();                                              \
  _csim_dt = (del_t);                                                       \
                                                                            \
  csim_sqs_out(_p_process);                                                 \
  if (csim_time_cmp(_csim_dt, csim_zero_time()) >= 0)                       \
    _p_process->evtime = csim_time_add(_p_process->evtime, _csim_dt);       \
  else                                                                      \
    csim_exception(CSIM_E_HOLD_DEL_T, _p_process, NULL);                    \
  csim_rank(_p_process);                                                    \
  _p_process->state = CSIM_PLANNED;                                         \
                                                                            \
  csim_compute_deactivate_ps(_p_process);                                   \
  _p_process->last_time = csim_time();                                      \
                                                                            \
  csim_switch_processes(_p_process, csim_sqs_point());                      \
```

```
  csim_restore_process;                                                    \
} while (0)

#define csim_passivate()                                                   \
do {                                                                       \
  CSIM_PROCESS *_p_process;                                                \
  _p_process = csim_current();                                             \
                                                                           \
  csim_sqs_out(_p_process);                                                \
  _p_process->state = CSIM_PASSIVE;                                        \
                                                                           \
  csim_compute_deactivate_ps(_p_process);                                  \
  _p_process->last_time = csim_time();                                     \
                                                                           \
  csim_switch_processes(_p_process, csim_sqs_point());                     \
  csim_restore_process;                                                    \
} while (0)

#define csim_wait(p_head)                                                  \
do {                                                                       \
  CSIM_PROCESS *_p_process;                                                \
  CSIM_HEAD *_csim_p_head;                                                 \
  _p_process = csim_current();                                             \
  _csim_p_head = (p_head);                                                 \
                                                                           \
  if (csim_check_head(_csim_p_head) == TRUE) {                             \
    csim_into(((CSIM_LINK *) _p_process), p_head);                         \
                                                                           \
    csim_sqs_out(_p_process);                                              \
    _p_process->state = CSIM_PASSIVE;                                      \
                                                                           \
    csim_compute_deactivate_ps(_p_process);                                \
    _p_process->last_time = csim_time();                                   \
  }                                                                        \
  else {                                                                   \
    csim_exception(CSIM_ERR_CODE(CSIM_E_WAIT_HEAD,                         \
     csim_head_state(_csim_p_head)), _p_process, _csim_p_head);            \
  }                                                                        \
                                                                           \
  csim_switch_processes(_p_process, csim_sqs_point());                     \
  csim_restore_process;                                                    \
} while (0)

#define csim_end_program                                                   \
{                                                                          \
  CSIM_PROCESS *_p_process;                                                \
  _p_process = csim_current();                                             \
                                                                           \
  csim_sqs_out(_p_process);                                                \
  _p_process->state = CSIM_TERMINATED;                                     \
                                                                           \
  csim_compute_deactivate_ps(_p_process);                                  \
  _p_process->last_time = csim_time();                                     \
```

```
                                                                        \
  csim_return_to_step();                                                \
}                                                                       \
}

/* Macros-brackets creating a special kind of functions */
#define csim_program(TYPE_NAME, PROG_NAME)                              \
void *PROG_NAME(void *p_void)                                           \
{                                                                       \
  TYPE_NAME *p_my = (TYPE_NAME *) p_void;

#define csim_end_process                                                \
{                                                                       \
  CSIM_PROCESS *_p_process;                                             \
  _p_process = csim_current();                                          \
                                                                        \
  csim_sqs_out(_p_process);                                             \
  _p_process->state = CSIM_ZOMBIE;                                      \
                                                                        \
  csim_return_to_step();                                                \
}                                                                       \
}

#define csim_begin_destructor(TYPE_NAME, DESTR_NAME)                    \
void DESTR_NAME(void *p_void)                                           \
{                                                                       \
  TYPE_NAME *p_my = (TYPE_NAME *) p_void;

#define csim_end_destructor                                             \
}

#define csim_begin_view(TYPE_NAME, VIEW_NAME)                           \
void VIEW_NAME(void *p_void)                                            \
{                                                                       \
  TYPE_NAME *p_my = (TYPE_NAME *) p_void;

#define csim_end_view                                                   \
}


/*************************************************************************
                          classes of basic objects
*************************************************************************/

/*
 * enumeration type - the states of process
 * CSIM_NEW_PASSIVE - new passive process
 *                    needed to recognize if we will run program
 *                    of process from the beginning or from
 *                    the reactivation point
 * CSIM_NEW_PLANNED - new scheduled process
 *                    needed to recognize if we will run program
 *                    of process from the beginning or from
```

```
 *                    the reactivation point
 * CSIM_PLANNED     - scheduled process run from
 *                    the reactivation point
 * CSIM_ACTIVE      - currently active process
 * CSIM_PASSIVE     - process is waiting to be scheduled
 * CSIM_TERMINATED  - program of process has ended
 * CSIM_ZOMBIE      - program of process has ended and process is disposed
 *                    after return to step
 */

typedef enum {
  CSIM_NEW_PASSIVE, CSIM_NEW_PLANNED, CSIM_PLANNED, CSIM_ACTIVE,
  CSIM_PASSIVE, CSIM_TERMINATED, CSIM_ZOMBIE
} CSIM_PROC_STATE;

/* enumeration type for states of CSIM_DYN_MEM type - used for internal checks
 * CSIM_D_IS_RANK    - object is probably inserted into the list of dynamic
 *                     blocs. We can only be sure that p_suc_mem, p_pred_mem
 *                     and p_head_mem are not NULL.
 * CSIM_D_NOT_RANK   - object is not inserted into the list of dynamic blocs
 * CSIM_D_DEFECTIVE  - object is not in any of the previous states
 * CSIM_D_ILLEGAL    - pointer to object does not correspond to control pointer
 *                     char *check set by object's generation
 * CSIM_D_IS_NULL    - pointer to checked object is NULL
 */

typedef enum {
  CSIM_D_DEFECTIVE, CSIM_D_ILLEGAL, CSIM_D_IS_NULL,
  CSIM_D_IS_RANK, CSIM_D_NOT_RANK
} CSIM_DYN_MEM_CHECK;

/* enumeration type for states of CSIM_LINK object - used for internal checks
 * CSIM_L_IS_RANK    - object is probably inserted into the list of dynamic
 *                     blocs. We can only be sure that suc, pred and head
 *                     are not NULL.
 * CSIM_L_NOT_RANK   - object is not inserted into any list
 * CSIM_L_DEFECTIVE  - object in not in some of previous states
 * CSIM_L_ILLEGAL    - pointer to object does not correspond to control pointer
 *                     char *check set by object's generation
 * CSIM_L_IS_NULL    - pointer to checked object is NULL
 */

typedef enum {
  CSIM_L_DEFECTIVE, CSIM_L_ILLEGAL, CSIM_L_IS_NULL,
  CSIM_L_IS_RANK, CSIM_L_NOT_RANK
} CSIM_LINK_CHECK;

/*
 * enumeration type for states of CSIM_HEAD object - used for internal checks
 * CSIM_H_EMPTY             - the list is empty - p_first and p_last are NULL
 * CSIM_H_NOT_EMPTY         - the list is probably not empty. We can only be sure
 *                      that p_first and p_last are not NULL.
 * CSIM_H_DEFECTIVE       - object is not in any of the previous states
```

```
 * CSIM_H_ILLEGAL - pointer to object does not correspond to control pointer
 *                      char *check set by object's generation
 * CSIM_H_IS_NULL         - pointer to checked object is NULL
 */

typedef enum {
  CSIM_H_DEFECTIVE, CSIM_H_ILLEGAL, CSIM_H_IS_NULL,
  CSIM_H_EMPTY, CSIM_H_NOT_EMPTY
} CSIM_HEAD_CHECK;

/*
 * enumeration type for states of CSIM_LINK object - used for internal checks
 * CSIM_P_IS_RANK    - process is in the scheduler. Sure is only that
 *                     suc_in_sqs and pred_in_sqs are not NULL
 * CSIM_P_NOT_RANK   - object is not in the scheduler - suc_in_sqs and
 *                     pred_in_sqs are NULL
 * CSIM_P_DEFECTIVE  - object is not in any of the previous states
 * CSIM_P_ILLEGAL    - pointer to object does not correspond to control pointer
 *                     char *check set by object's generation
 * CSIM_P_IS_NULL    - pointer to checked object is NULL
 */

typedef enum {
  CSIM_P_DEFECTIVE, CSIM_P_ILLEGAL, CSIM_P_IS_NULL, CSIM_P_IS_TERMINATED,
  CSIM_P_IS_RANK, CSIM_P_NOT_RANK
} CSIM_PROC_CHECK;


typedef void *((*CSIM_PROC_PROGRAM) (void *p_void)); /* program of process    */
typedef void ((*CSIM_DESTRUCTOR) (void *p_void));    /* destructor            */
typedef void ((*CSIM_VIEW) (void *p_void));          /* function used to view */

/*
 * Types of basic objects
 */

typedef struct csim_dyn_mem {
  csim_d_dyn_mem;
} CSIM_DYN_MEM;

typedef struct csim_link {
  csim_d_link;
} CSIM_LINK;

typedef struct csim_head {
  csim_d_head;
} CSIM_HEAD;

typedef struct csim_process {
  csim_d_process;
} CSIM_PROCESS;

/*
```

```
 * Enumeration type for error code generation
 */

typedef enum {
  CSIM_E_SET_NAME_DYN_MEM = 1,
  CSIM_E_FIRST_HEAD,       /* csim_first()                */
  CSIM_E_LAST_HEAD,        /* csim_last()                 */
  CSIM_E_EMPTY_HEAD,       /* csim_empty()                */
  CSIM_E_CARD_HEAD,        /* csim_cardinal()             */
  CSIM_E_CLEAR_HEAD,       /* csim_clear()                */
  CSIM_E_OUT_LINK,         /* csim_out()                  */
  CSIM_E_INTO_LINK,        /* csim_into()                 */
  CSIM_E_INTO_HEAD,
  CSIM_E_FOLL_WHAT,        /* csim_follow()               */
  CSIM_E_FOLL_WHERE,
  CSIM_E_PREC_WHAT,        /* csim_precede()              */
  CSIM_E_PREC_WHERE,
  CSIM_E_WAIT_HEAD,        /* csim_wait()                 */
  CSIM_E_DISP_DYN_MEM,     /* csim_dispose_dyn_mem()      */
  CSIM_E_DISP_LINK,        /* csim_dispose_link()         */
  CSIM_E_DISP_HEAD,        /* csim_dispose_head()         */
  CSIM_E_DISP_PROC,        /* csim_dispose_process()      */
  CSIM_E_STATE_PROC,       /* csim_state()                */
  CSIM_E_IDLE_PROC,        /* csim_idle()                 */
  CSIM_E_EVTIME_PROC,      /* csim_evtime()               */
  CSIM_E_CANCEL_PROC,      /* csim_cancel()               */
  CSIM_E_ACT_AT_PROC,      /* csim_activate_at()          */
  CSIM_E_ACT_DEL_PROC,     /* csim_activate_delay()       */
  CSIM_E_REACT_AT_PROC,    /* csim_reactivate_at()        */
  CSIM_E_REACT_DEL_PROC,   /* csim_reactivate_delay()     */
  CSIM_E_HSTAT_ON_HEAD,    /* csim_h_stat_on()            */
  CSIM_E_HSTAT_OFF_HEAD,   /* csim_h_stat_off()           */
  CSIM_E_HSTAT_INIT_HEAD,  /* csim_init_h_stat()          */
  CSIM_E_HSTAT_ST_HEAD,    /* csim_h_stat_status()        */
  CSIM_E_HSTAT_HEAD,       /* csim_h_stat()               */
  CSIM_E_PSTAT_ON_PROC,    /* csim_p_stat_on()            */
  CSIM_E_PSTAT_OFF_PROC,   /* csim_h_stat_off()           */
  CSIM_E_PSTAT_INIT_PROC,  /* csim_init_h_stat()          */
  CSIM_E_PSTAT_ST_PROC,    /* csim_h_stat_status()        */
  CSIM_E_PSTAT_PROC,       /* csim_h_stat()               */

  CSIM_E_ERROR_CODE_SEPARATOR,

  CSIM_E_INTO_LINK_IS_IN,
  CSIM_E_FOLL_WHAT_IS_IN, CSIM_E_FOLL_WHERE_NOT_IN,
  CSIM_E_PREC_WHAT_IS_IN, CSIM_E_PREC_WHERE_NOT_IN,
  CSIM_E_HOLD_DEL_T,
  CSIM_E_INIT_PROC_PROG_NULL,
  CSIM_E_DISP_PROC_ACTIVE,
  CSIM_E_STEP_EMPTY,
  CSIM_E_CANCEL_NOT_RANK, CSIM_E_CANCEL_IS_ACTIVE,
  CSIM_E_ACT_AT_T, CSIM_E_ACT_AT_IS_RANK, CSIM_E_ACT_AT_IS_ACTIVE,
  CSIM_E_ACT_DEL_T, CSIM_E_ACT_DEL_IS_RANK, CSIM_E_ACT_DEL_IS_ACTIVE,
```

```
  CSIM_E_REACT_AT_T, CSIM_E_REACT_AT_IS_ACTIVE,
  CSIM_E_REACT_DEL_T, CSIM_E_REACT_DEL_IS_ACTIVE,
  CSIM_E_CREATE_PROCESS,

  CSIM_USER_ERROR_CODE
} CSIM_ERROR_CODE;

/*
 * Structure describing the error - filled by function error(...)
 *
 * error_code   - code of the error
 * p_proc_error - pointer to process that was active when
 *                the error occured
 * p_void       - pointer to object that caused the error
 *                (the debug function can be used to view the object)
 *                user can define his own errors - in that case there
 *                can be pointer to user-defined structure
 */

typedef struct {
  CSIM_UWORD      error_code;
  CSIM_PROCESS    *p_proc_error;
  void            *p_void;
} CSIM_ERROR;


/*
 * Initialization functions and functions for dynamic allocation
 */

/* Memory initialization */
extern CSIM_RESULT csim_init_mem(void);

/* Clears the list of dynamic blocks */
extern void csim_clear_mem(void);

/* Initializes the CSIM_DYN_MEM object */
extern CSIM_DYN_MEM *csim_init_dyn_mem(CSIM_DYN_MEM *p_dyn_mem);

/* Initializes the CSIM_LINK object */
extern CSIM_LINK *csim_init_link(CSIM_LINK *p_link);

/* Initializes the CSIM_HEAD object */
extern CSIM_HEAD *csim_init_head(CSIM_HEAD *p_head);

/* Initializes the CSIM_PROCESS object */
extern CSIM_PROCESS *csim_init_process(CSIM_PROCESS *p_process,
                                       CSIM_PROC_PROGRAM proc_prog);

/* Releases the CSIM_DYN_MEM object's memory */
extern CSIM_RESULT csim_dispose_dyn_mem(CSIM_DYN_MEM *p_dyn_mem);

/* Releases the CSIM_LINK object's memory */
```

```
extern CSIM_RESULT csim_dispose_link(CSIM_LINK *p_link);

/* Releases the CSIM_HEAD object's memory */
extern CSIM_RESULT csim_dispose_head(CSIM_HEAD *p_head);

/* Releases the CSIM_PROCESS object's memory */
extern CSIM_RESULT csim_dispose_process(CSIM_PROCESS *p_process);

extern CSIM_RESULT csim_set_name(CSIM_DYN_MEM *p_dyn_mem, char *name);

/*
 * Functions that support the CSIM_LINK object
 */

/* Removes the item prom the list */
extern CSIM_RESULT csim_out(CSIM_LINK *p_link);

/* Inserts the item into the list on the last place */
extern CSIM_RESULT csim_into(CSIM_LINK *p_link,
                      CSIM_HEAD *p_head);

/* Inserts the item into the list as successor of another item */
extern CSIM_RESULT csim_follow(CSIM_LINK *p_link_what,
                               CSIM_LINK *p_link_where);

/* Inserts the item into the list as predecessor of another item */
extern CSIM_RESULT csim_precede(CSIM_LINK *p_link_what,
                                CSIM_LINK *p_link_where);

/*
 * Functions that support the CSIM_HEAD object
 */

/* Returns the first item in the list */
extern CSIM_LINK *csim_first(CSIM_HEAD *p_head);

/* Returns the last item in the list */
extern CSIM_LINK *csim_last(CSIM_HEAD *p_head);

/* Returns TRUE if the list is empty */
extern CSIM_BOOLEAN csim_empty(CSIM_HEAD *p_head);

/* Returns actual list length */
extern CSIM_WORD csim_cardinal(CSIM_HEAD *p_head);

/* Empties the list */
extern CSIM_RESULT csim_clear(CSIM_HEAD *p_head);

/*
 * Functions that support process and simulation control
 */

/* Schedules process on time t */
```

```
extern CSIM_RESULT csim_activate_at(CSIM_PROCESS *p_process,
                                    CSIM_TIME t);

/* Scheduler process on current time plus del_t */
extern CSIM_RESULT csim_activate_delay(CSIM_PROCESS *p_process,
                                       CSIM_TIME del_t);

/* Rechedules process on time t */
extern CSIM_RESULT csim_reactivate_at(CSIM_PROCESS *p_process,
                                      CSIM_TIME t);

/* Reschedules process on current time plus del_t */
extern CSIM_RESULT csim_reactivate_delay(CSIM_PROCESS *p_process,
                                         CSIM_TIME del_t);

/* Removes process from scheduler */
extern CSIM_RESULT csim_cancel(CSIM_PROCESS *p_process);

/* Returns the state of process */
extern CSIM_PROC_STATE csim_state(CSIM_PROCESS *p_process);

/* Tests the process state - returns TRUE if the process is passive */
extern CSIM_BOOLEAN csim_idle(CSIM_PROCESS *p_process);

/* Returns the process scheduled time */
extern CSIM_TIME csim_evtime(CSIM_PROCESS *p_process);

/* Returns pointer to the second process in scheduler */
extern CSIM_PROCESS *csim_next_proc(void);

/* Realizes one step of simulation alghoritm */
extern CSIM_RESULT csim_step(void);

/* Returns actual model time */
extern CSIM_TIME csim_time(void);

/* Returns pointer to currently active */
extern CSIM_PROCESS *csim_current(void);

/* Returns pointer to the head of the scheduler's list */
extern CSIM_PROCESS *csim_sqs_point(void);

/* Returns pointer to the head of the heap */
extern CSIM_DYN_MEM *csim_mem_point(void);

/*
 * Check functions
 */

/* Returns control state of the CSIM_DYN_MEM object */
extern CSIM_DYN_MEM_CHECK csim_dyn_mem_state(CSIM_DYN_MEM *p_dyn_mem);

/* Returns control state of the CSIM_LINK object */
```

```
extern CSIM_LINK_CHECK csim_link_state(CSIM_LINK *p_link);

/* Returns control state of the CSIM_HEAD object */
extern CSIM_HEAD_CHECK csim_head_state(CSIM_HEAD *p_head);

/* Returns control state of the CSIM_PROCESS object */
extern CSIM_PROC_CHECK csim_process_state(CSIM_PROCESS *p_process);

/* Checks the CSIM_DYN_MEM object */
extern CSIM_BOOLEAN csim_check_dyn_mem(CSIM_DYN_MEM *p_dyn_mem);

/* Checks the CSIM_PROCESS object */
extern CSIM_BOOLEAN csim_check_process(CSIM_PROCESS *p_process);

/* Checks the CSIM_HEAD object */
extern CSIM_BOOLEAN csim_check_head(CSIM_HEAD *p_head);

/* Checks the CSIM_LINK object */
extern CSIM_BOOLEAN csim_check_link(CSIM_LINK *p_link);

/*
 * Statistics functions
 */

/* Turns the global statistics on */
extern void csim_stat_on(void);

/* Turns the global statistics off */
extern void csim_stat_off(void);

/* Returns the state of global statistics - TRUE if on */
extern CSIM_BOOLEAN csim_stat_status(void);

/* Turns the local statistics of the object CSIM_HEAD on */
extern CSIM_RESULT csim_h_stat_on(CSIM_HEAD *p_head);

/* Turns the local statistics of the object CSIM_HEAD off */
extern CSIM_RESULT csim_h_stat_off(CSIM_HEAD *p_head);

/* Initiates the statistics variables */
extern CSIM_RESULT csim_init_h_stat(CSIM_HEAD *p_head);

/* Returns state of the local statistics - TRUE if on */
extern CSIM_BOOLEAN csim_h_stat_status(CSIM_HEAD *p_head);

/* Returns statistics results */
extern CSIM_RESULT csim_h_stat(CSIM_HEAD *p_head,
                               double *p_Lw,
                               double *p_Tw);

/* Turns the local statistics of the CSIM_PROCESS object on */
extern CSIM_RESULT csim_p_stat_on(CSIM_PROCESS *p_process);
```

```
/* Turns the local statistics of the CSIM_PROCESS object off */
extern CSIM_RESULT csim_p_stat_off(CSIM_PROCESS *p_process);

/* Initiates the statistics variables */
extern CSIM_RESULT csim_init_p_stat(CSIM_PROCESS *p_process);

/* Returns state of the local statistics - TRUE if on */
extern CSIM_BOOLEAN csim_p_stat_status(CSIM_PROCESS *p_process);

/* Returns statistics results */
extern CSIM_RESULT csim_p_stat(CSIM_PROCESS *p_process,
                               double *p_Ts,
                               double *p_Tr);

/*
 * Error functions
 */

/* error indication */
extern CSIM_BOOLEAN csim_error_status(void);

/* sets the error structure with actual error and returns it */
extern void csim_error(CSIM_ERROR *p_error);

/*
 * Functions used in macros, DO NOT USE DIRECTLY!
 */

/* Inserts process into scheduler on place determined by the value of evtime */
extern void csim_rank(CSIM_PROCESS *p_process);

/* Removes process from calendar */
extern void csim_sqs_out(CSIM_PROCESS *p_process);

/* compute process statistics */
extern void csim_compute_deactivate_ps(CSIM_PROCESS *p_process);

/* sets the error and ends the simulation step */
extern void csim_exception(CSIM_UWORD error_code,
                           CSIM_PROCESS *p_proc,
                           void *p_void);

/*
 * Global variables
 */

/* counter of simulation steps */
extern unsigned long csim_step_number;

#endif
```

Appendix B

Error Messages Module Interface

```
/*****************************************************************************
            University of West Bohemia, DCS, Pilsen, Czech Republic
                                (c) copyright
                                  27.03.2002

                               C-Sim version 5.1

                                file csim_err.h

                      header file of C-Sim error messages

*****************************************************************************/

#ifndef CSIM_ERR_H
#define CSIM_ERR_H 1

#include "csim.h"

/*
 * Translates the given numerical error code into a human readable text.
 * The function returns pointer to a staticaly allocated string.
 */
extern char *csim_error_msg(CSIM_UWORD code);

#endif
```

Appendix C

Random Number Generator Module Interface

```
/*************************************************************************
            University of West Bohemia, DCS, Pilsen, Czech Republic
                                (c) copyright
                                  27.03.2002

                               C-Sim version 5.1

                                file csim_rng.c

                     header file of random number generator
                          Random number generator TT800
                  by M. Matsumoto, email: matumoto@math.keio.ac.jp
                                1996 Version

*************************************************************************/

#ifndef CSIM_RNG_H
#define CSIM_RNG_H 1

#include "csim.h"

#define CSIM_RAND_MAX 0xFFFFFFFFL         /* maximum value returned by rng */
#define CSIM_RNG_N 25

/*
 * The random generator state is stored within the following structure.
 * Usage example:
 * {
 *   RNG_STATE *rng;
 *   double x;
 *
 *   rng = s_new_rng(INIT_SEED);
 *     ...
 *   x = (double) csim_rand(rng) / (double) CSIM_RAND_MAX;
```

```
 *     ...
 *   dispose_rng(rng);
 * }
 */

/* Holds the current state of a random number generator (RNG). */
csim_derived_dyn_mem(csim_rng_state)
  CSIM_ULONG rng_array[CSIM_RNG_N];
  int index;
csim_end_derived(CSIM_RNG_STATE);

/*
 * Allocates dynamic memory for a new RNG and initializes the structure.
 * Returns pointer to the new dynamic memory.
 */
#define csim_new_rng(seed)                                                    \
  (CSIM_RNG_STATE *) csim_init_rng(                                           \
    (CSIM_RNG_STATE *) malloc(sizeof(CSIM_RNG_STATE)), seed);

/* Frees all resources asociated with specified RNG. */
#define csim_dispose_rng(rng)                                                 \
  csim_dispose_dyn_mem((CSIM_DYN_MEM *) rng);

/*
 * Initializes the specified RNG structure. Must be called explicitly if
 * staticaly allocated RNG is used.
 */
CSIM_RNG_STATE *csim_init_rng(CSIM_RNG_STATE *p_rng,
                              CSIM_ULONG seed);

/* Generates a single random number in the range <0,CSIM_RAND_MAX> */
CSIM_ULONG csim_rand(CSIM_RNG_STATE *p_rng);

/* Generator of exponential distribution */
double csim_negexp(CSIM_RNG_STATE *p_rng,
                   double lambda);

/* Generator of the regular distribution on the interval <a,b> */
double csim_uniform(CSIM_RNG_STATE *p_rng,
                    double a,
                    double b);

/* Returns TRUE with probability p (FALSE with probability 1-p) */
CSIM_BOOLEAN csim_draw(CSIM_RNG_STATE *p_rng,
                       double p);

/* Generator of the normal (Gaussian) distribution */
double csim_gauss(CSIM_RNG_STATE *p_rng,
                  double sigma,
                  double center);

#endif
```

Appendix D

Semaphores Module Interface

```
/*****************************************************************************
            University of West Bohemia, DCS, Pilsen, Czech Republic
                                (c) copyright
                                  27.03.2002

                               C-Sim version 5.1

                                file csim_sem.h

                        header file of C-Sim semaphores

*****************************************************************************/

#ifndef CSIM_SEM_H
#define CSIM_SEM_H 1

#include "csim.h"


/*
 * Structure that contains the state of a semaphore is defined bellow.
 * Usage example:
 * {
 *   CSIM_SEMAPHORE *sem;
 *   sem = s_new_semaphore(1);
 *   lock_sem(sem);
 *     ...
 *   unlock_sem(sem);
 *   dispose_semaphore(sem);
 * }
 */

/* Structure that contains the complete state of a semaphore. */
typedef struct csim_semaphore {
  csim_d_head;
  CSIM_UWORD count;
} CSIM_SEMAPHORE;
```

```
/*
 * Dynamicaly creates (allocates memory) and initializes a semaphore.
 * The cnt argument is the maximum number of simultanous locks on the
 * semaphore (typicaly 1).
 */
#define csim_new_semaphore(cnt)                                          \
  csim_init_semaphore((CSIM_SEMAPHORE *) malloc(sizeof(CSIM_SEMAPHORE)), cnt);

/* Frees all resources asociated with the semaphore. */
#define csim_dispose_semaphore(sem)                                      \
  csim_dispose_head((CSIM_HEAD *) sem);


/*
 * Initializes the given semaphore. This function must be used explicitly
 * if staticaly allocated semaphore is used.
 */
CSIM_SEMAPHORE *csim_init_semaphore(CSIM_SEMAPHORE *p_sem,
                                    CSIM_UWORD cnt);


/*
 * Locks the given semaphore for the current process. If no lock is available,
 * the process sleeps til it may continue.
 */
#define csim_lock_sem(p_sem)                                             \
do {                                                                     \
  CSIM_SEMAPHORE *_p_sem;                                                \
                                                                         \
  _p_sem = (p_sem);                                                      \
  if (_p_sem->count > 0)                                                 \
    (_p_sem->count)--;                                                   \
  else {                                                                 \
    csim_wait((CSIM_HEAD *) _p_sem);                                     \
  }                                                                      \
} while (0)


/*
 * Unlocks the given semaphore. Any locked semaphore must be freed after the
 * protected resource is no longer needed.
 */
void csim_unlock_sem(CSIM_SEMAPHORE *p_sem);

#endif
```

Appendix E

Message Passing Module Interface

```
/*****************************************************************************
            University of West Bohemia, DCS, Pilsen, Czech Republic
                                (c) copyright
                                  27.03.2002

                               C-Sim version 5.1

                                file csim_msg.h

                    header file of C-Sim message passing system

*****************************************************************************/

#ifndef CSIM_MSG_H
#define CSIM_MSG_H 1

#include "csim.h"

/* Constants used in message transmissison */
#define CSIM_ANY_MSG_RECEIVER  NULL
  /* The message can be received by any csim_process                    */
#define CSIM_ANY_MSG_SENDER    NULL
  /* The receiver accepts messages from any server                      */
#define CSIM_ANY_MSG_TYPE      0xFFFFFFFFL
  /* This mask ensures the reception of any message type and vice versa */

/* Macro for derivation of user defined message type */
#define csim_d_message         \
  csim_d_link;                 \
  CSIM_ULONG       msg_type;   \
  CSIM_PROCESS     *sender;    \
  CSIM_PROCESS     *receiver

#define csim_derived_message(TYPE_NAME) typedef struct TYPE_NAME {           \
                                          csim_d_message;

/*
```

```
 * Standard message type
 * Use like this:
 *
 * csim_program(CSIM_PROCESS, sender_program)
 * {
 *   CSIM_MESSAGE *msg;
 *
 *   msg = csim_new_message(CSIM_MESSAGE);
 *   csim_send_msg(msg, 1, CSIM_ANY_MSG_RECEIVER);
 * }
 *
 * csim_program(CSIM_PROCESS, receiver_program)
 * {
 *
 *   csim_receive_msg(my.msg, 1, CSIM_ANY_MSG_SENDER);
 *   csim_dispose_message(msg);
 * }
 */
typedef struct csim_message {
  csim_d_message;
} CSIM_MESSAGE;

/* For internal use only - DO NOT USE */
typedef struct csim_waiting_process {
  csim_d_link;
  CSIM_ULONG    msg_type;
  CSIM_PROCESS  *sender;
  CSIM_PROCESS  *receiver;
  CSIM_MESSAGE  **p_msg;
} CSIM_WAITING_PROCESS;

/*
 * Initialization of the Message-passing module. This function should be
 * called exactly once - before other function.
 * Before exiting the function "csim_clear_msg()" should be called.
 */
extern CSIM_RESULT csim_init_msg();

/* Frees all alocated resources by this module. */
extern void csim_clear_msg();

/*
 * Dynamicaly creates (allocates memory) and initializes a new message.
 * The TYPE_NAME argument can be instance of any structure derived using
 * the "csim_d_message" macro.
 */
#define csim_new_message(TYPE_NAME)                                          \
  (TYPE_NAME *) csim_init_message((CSIM_MESSAGE *) malloc(sizeof(TYPE_NAME)));

/*
 * Frees the resources allocated to the specified message. The message must
 * have been allocated dynamicaly using the csim_new_message macro.
 */
```

```
#define csim_dispose_message(msg)                                          \
  csim_dispose_link((CSIM_LINK *) msg);

/*
 * Initializes any user-derived message with the default starting values.
 * Must be called before any use of the message.
 */
extern CSIM_MESSAGE *csim_init_message(CSIM_MESSAGE *msg);

/*
 * Sends the specified message "msg" to the "receiver" process. The message
 * will be received only if the bitwise AND of sender and receiver "msg_type"
 * gives a nonzero result. If the "receiver" argument is CSIM_ANY_MSG_RECEIVER
 * the message is received by the first available process
 * (with appropiate message type).
 */
extern void csim_send_msg(CSIM_MESSAGE *msg,
                          CSIM_ULONG msg_type,
                          CSIM_PROCESS *receiver);

/*
 * Starts the receiving of a message. The call to this macro blocks the process
 * until a message is received. The message is written into the "r_msg"
 * argument, which is pointer to a message. In the long_jump implementation of
 * C-Sim the actual argument cannot be a local variable as the pointer would be
 * destroyed. The "r_sender" parameter specifies the process from which message
 * may be received. If "r_sender" is CSIM_ANY_MSG_SENDER, the message may be
 * received from any process. The "r_msg_type" argument specifies which types
 * of message can be received and is explained in the previous function.
 */
#define csim_receive_msg(r_msg, r_msg_type, r_sender)                      \
do {                                                                       \
  CSIM_ULONG _r_msg_type;                                                  \
  CSIM_PROCESS *_r_sender;                                                 \
  CSIM_BOOLEAN _in_msg_queue = FALSE;                                      \
  CSIM_MESSAGE *_p_m;                                                      \
                                                                           \
  _r_msg_type = (r_msg_type);                                              \
  _r_sender   = (r_sender);                                                \
                                                                           \
  if ((_p_m = (CSIM_MESSAGE *) csim_first(csim_msg_queue_point())) != NULL) { \
    do {                                                                   \
      if (csim_test_msg(_p_m, _r_msg_type, _r_sender, csim_current())) {   \
        csim_out((CSIM_LINK *) _p_m);                                      \
        (CSIM_MESSAGE *)(r_msg) = _p_m;                                    \
        _in_msg_queue = TRUE;                                              \
      }                                                                    \
      else                                                                 \
        _p_m = (CSIM_MESSAGE *) _p_m->suc;                                 \
    } while ((_in_msg_queue == FALSE)                                      \
      && (_p_m != (CSIM_MESSAGE *) csim_first(csim_msg_queue_point())));   \
  }                                                                        \
  if (_in_msg_queue == FALSE) {                                            \
```

```
    CSIM_WAITING_PROCESS *_p_wp;                                                \
                                                                                \
    _p_wp = (CSIM_WAITING_PROCESS *) csim_new_link(CSIM_WAITING_PROCESS);       \
    _p_wp->msg_type = _r_msg_type;                                              \
    _p_wp->sender   = _r_sender;                                                \
    _p_wp->receiver = csim_current();                                           \
    _p_wp->p_msg    = (CSIM_MESSAGE **) &(r_msg);                               \
    csim_into((CSIM_LINK *) _p_wp, csim_wp_queue_point());                      \
    csim_passivate();                                                           \
  }                                                                             \
} while(0)


/* For internal use only - DO NOT USE */
CSIM_BOOLEAN csim_test_msg(CSIM_MESSAGE *msg,
                           CSIM_ULONG msg_type,
                           CSIM_PROCESS *sender,
                           CSIM_PROCESS *receiver);

/* For internal use only - DO NOT USE */
extern CSIM_HEAD *csim_msg_queue_point();
extern CSIM_HEAD *csim_wp_queue_point();

#endif
```

Appendix F

Console Debug Module Interface

```
/*****************************************************************************
            University of West Bohemia, DCS, Pilsen, Czech Republic
                                (c) copyright
                                  27.03.2002

                               C-Sim version 5.1

                                file csim_dgb.h

                    header file of C-Sim console debug rutines

*****************************************************************************/

#ifndef CSIM_DBG_H
#define CSIM_DBG_H 1

#include "csim.h"

/*
 * Basic C-Sim object VIEW functions for console oriented output. Every
 * object uses different function - specified in the 'view' attribute.
 */

/* Shows the CSIM_DYN_MEM object */
void csim_view_dyn_mem(void *p_void);

/* Shows the CSIM_LINK object */
void csim_view_link(void *p_void);

/* Shows the CSIM_HEAD object */
void csim_view_head(void *p_void);

/* Shows the CSIM_PROCESS object */
void csim_view_process(void *p_void);

/* Function for viewing the list of dynamically generated objects */
void csim_debug(CSIM_DYN_MEM *p_dyn);
```

```
#endif
```