Abstract

C-Sim is a programming tool for simulation of discrete processes using a method of pseudo-parallel processes. It is an extension of C language obtained by including SIMULA-like macros and functions. C-Sim uses a special C-functions and C-macros library. The typical application area of C-Sim is functional validation of distributed, parallel and fault-tolerant systems and programs.

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 originates from the programming language SIMULA [SIM92] 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. The version 5.0 of C-Sim library has been based on comments from users of the older version C-Sim 4.1 [WWW1, CSIM99, CSIM02].

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. Improvements brought by the new version:

- better readability and safety of source code,
- implementation of POSIX threads based mechanism of pseudo-parallel processes and modification of the C-Sim kernel to allow the use of both mechanisms,
- division of C-Sim kernel into several modules,
- addition of new features.

2. Internal structure

The C-Sim library is divided to three layers as seen in Fig.1. The lowest layer contains the definitions of simple data types (e.g. CSIM_BOOLEAN, CSIM_BYTE) and type CSIM_TIME. 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. The header file csim_kr.h is provided in two versions, one for long-jump based implementation and one for POSIX threads based implementation of process switching. Library user may choose desired implementation at compile time by defining the macro KR_PATH on the compiler command line.

** Ing., Department of Computer Science, FAV, West Bohemian Univ., Univerzitní 22, 306 14 Plzen.
The second layer is defined by files `csim.c` and `csim.h`, which contain definitions of data types, macros and functions for creating a simulation model and for controlling simulation flow. As a part of this layer, the functions for dynamic memory control and for object checking are also defined.

![C-Sim module layers](image)

The third layer contains only optional modules, which are:
- random number generator,
- error code to error message conversion,
- console debug tools,
- message passing,
- integer semaphores.

3. Supported object types and user types derivation

As 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`. Hence, it is possible to create list of list heads (i.e. list of lists). This way the user obtains a tool for creating complicated structures of objects. The hierarchy of C-Sim supported object types is depicted in Fig.2.
The C-Sim library provides a possibility to extend the standard object types by own data items (in the object-oriented programming the attributes). Deriving user modifications of the basic supported types is implemented with the aid of macros.

![Fig.2: Object types hierarchy in C-Sim](image)

4. 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 and then started again after a change of its parameters. The memory allocated during the first simulation run must be released properly. If the library user were given the possibility of an uncontrolled manipulation with the dynamic memory, a simple repetition of the simulation run would not be possible. In C-Sim, all dynamic memory is accessed through the predefined type CSIM_DYN_MEM. Because of this, the library is able to free all allocated memory at any time.

5. Implementation of pseudo-parallel processes

The applied method of discrete simulation is based on the use of pseudo-parallel processes. The process programs are also called cooperating routines, shortly coroutines. 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 realizes one simulation step, is constantly being called. The main loop itself is a coroutine and this coroutine creates and destroys all the other coroutines as needed by the simulation.

5.1. 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. These functions allow storing the process context (e.g. contents of processor's registers) into a predefined structure and later return to the stored process status. They provide means for switching the process context without using any nonstandard operations, which improves the portability, security and readability of the library code. A macro that contains a reactivation point is defined as follows:

```c
#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) using the functions `setjmp` and `longjmp`, cannot ensure that the values of its parameters, local (automatic) variables and return address are unchanged\(^1\).

All these values are stored on stack and a possible jump to another part of the program code can overwrite them. This has several important consequences. Functions that perform context switch do not have the stability of local data guaranteed, and should therefore use global memory to store their intermediate state. With return address the problem is even worse – there is no safe way to restore the return address once it has been overwritten, instead a jump operation to another known address must be performed (this is handled in predefined macros in the C-Sim library).

### 5.2. Implementation based on POSIX threads

The second method of implementation of the coroutines is based on the use of threads. To retain maximum portability of the code, POSIX threads were chosen as an interface to threads – IEEE standard 1003.1 [WWW2].

A thread is created for each process of the pseudo-parallel processes method. For the implementation however, it is 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.

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

```c
#define csim_switch_to_process(process) \  do { \    pthread_mutex_lock(&(csim_sqs_point()->mutex)); \    csim_sqs_point()->thread_run = FALSE; \    pthread_mutex_lock(&(process->mutex)); \    process->thread_run = TRUE; \    pthread_mutex_unlock(&(process->mutex)); \    pthread_cond_signal(&(process->cond_var)); \    while (csim_sqs_point()->thread_run != TRUE) \    pthread_cond_wait(&(csim_sqs_point()->cond_var), \    &(csim_sqs_point()->mutex)); \    pthread_mutex_unlock(&(csim_sqs_point()->mutex)); \  } while (0)
```

### 6. Random number generator module

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

\(^1\) In fact, we can find one exception from this rule. If we use several nested calls of functions, the inmost of which is the first to call critical function, there will be created a compact layer of their return addresses and other data at the bottom of the stack.
function, to create multiple instances of a generator, i.e. it is possible to use any number of independent sequences of random numbers.

The congruent function `csim_rand` returns 32-bit unsigned integers uniformly distributed in the \(<0, \text{CSIM\_RAND\_MAX}>\) interval. Moreover, the module provides functions for the generation of random numbers with other than uniform distribution. These function are based upon various methods of uniform-to-other distribution transformation, i.e. all use the function `csim_rand` internally.

7. Semaphores module

One type of C-Sim application is the verification of parallel programs. Such programs require synchronization objects. This module provides a basic synchronization object – an integer semaphore.

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 here not necessary. The lock operation is defined as a preprocessor macro because it can block a process.

8. Message passing module

This module implements asynchronous inter-process communication using the method of message passing 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. The pointer to their process record may uniquely specify both the receiver and the sender.

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.

9. Comparison of pseudo-parallel processes implementation

The long-jump based implementation sets some restricting rules for a potential library user, when creating programs of process-like objects. The use of `setjmp` and `longjmp` functions causes invalidation of stack, what in turn leads to invalidation of automatic local variables. Other problems (like the destruction of the return address of a function) are handled in macros and functions provided by C-Sim. In contrast the POSIX threads based implementation, because each thread has its own stack, allows an unrestricted use of automatic variables and nesting of function calls.

The POSIX implementation is slower than long-jump implementation because the process switching in POSIX threads is more complex than the long-jump operation (administrative overhead). The slow-down is dependent on the frequency of process switching, i.e. the amount of code executed in one simulation step. When using worst-case benchmark application, the POSIX threads implementation is approximately 50 times slower on Linux and 10 times slower on OSF/1. A real simulation used in benchmarks shows that the thread implementation is slower approximately 15 times on Linux and 3 times on OSF/1.

The implementation based on long-jumps is used when:

- portability is important,
- speed is important,
- there is no need to use automatic variables in programs of processes and nested functions calls.
The implementation based on POSIX threads is used when:
- the target platform provides POSIX threads,
- speed is not a critical issue,
- automatic variables in programs of processes or nested function calls are required.

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

10. Conclusion

All changes to C-Sim were applied with regard to best possible backwards compatibility with the previous library version, although full compatibility could not be retained. For this reason, a conversion utility that translates source codes written for C-Sim version 4.1 to the new library version has been created.

The library is still in development, but its interface is already stable. Currently most effort is given into testing of the new features, for this purpose a relatively large project based on C-Sim ver. 4.1 will be updated to version 5.0 (the EU FIT project [WWW3, TTP01]). In addition, the new library version will be used in subsequent courses on University of West Bohemia.

References


Acknowledgement

This research was in part supported by a grant of the 5th Framework Program Information Societies Technology, IST-1999-10748 Fault injection for Time-Triggered Architecture (FIT). Moreover, it was supported by a grant of Czech Republic Ministry of education, MSM 235200005 Information systems and technologies.