

The OOSimL

Object Oriented Simulation Language: Generating C++ Code

José M. Garrido
email: jgarrido@kennesaw.edu
Department of Computer Science
College of Computing and Software Engineering
Kennesaw State University

Updated November 2016

Abstract

The Object Oriented Simulation Language (OOSimL) was designed mainly for research and education. The original language only generated Java code. This paper presents several features of the OOSimL simulation language that generates C++ code and includes an example of a simulation model developed with OOSimL translator.

The semantics of OOSimL is informally defined to be a combined Java and C++ semantics. However, there are several features in the C++ version of the OOSimL language, one of them is the ability to separate class specifications from class implementations. New keywords have been added to maintain full compatibility with C++. The OOSimL language translator is an executable program that is implemented as a one-pass language processor and that generates C++ source code. The generated code can be integrated conveniently with any C++ library, on Linux and Windows.

The OOSimL language development is part of the OOPsim project, and has had partial NSF CPATH support. One of the goals of this project is to develop new tools and approaches for modeling and simulation in computer science, software engineering, information technology, and related disciplines.

Keywords: Object Orientation, Discrete Event Simulation, Process Interaction, Language Translation, Specification, Implementation, Object Oriented Programming.

1 Introduction

The OOSimL simulation language supports the conceptual framework of the process style of discrete-event simulation that can facilitate the application of

modeling and simulation of large and complex systems.

The OOSimL language syntax is defined at a higher level than C++ and Java and it is an enhancement to the standard pseudo-code used in program and algorithm design. OOSimL has retained the combined semantics of Java and C++. The language design was influenced by Simula [1], Eiffel [12], Ada [3], Java, and C++ [13].

The language syntax includes assertions for improving program correctness. The language can also help avoid most (or all) of the C++ pitfalls identified by Cay S. Horstmann. The language design was influenced by Simula, the Demos simulation package, Eiffel, Java, and C++.

An object-oriented simulation programming language has several advantages of compared to an integrated icon-based simulation tool; some of these are: the flexibility and power of the simulation language versus the ease of use of the integrated simulation tool.

The OOSimL language also supports further research in: simulation languages, developing Simulation Modeling Specification Languages for relevant family of systems, higher-level DEVS support, advanced Cellular and Component Simulation modeling, the next generation of Distributed and Parallel Simulation Languages, and other related areas of research. The language supports the application of simulation in industry; it provided a viable and short transition to commercial simulation (programming) languages such as Simscript III, which is no longer marketed.

The OOSimL simulation language is part of the OOPsim project, which began in 1996. Several software tools have been developed and several books ([5],[10], and [11]) have been written that explain the simulation concepts and the application of the simulation approach supported. The development of the OOSimL language was partially funded by an NSF CPATH grant.

The complete software, which includes the compiler/translators (for Java and C++), runtime libraries and several examples, can be downloaded free of cost for education and research purposes from the following Web page:

<http://ksuweb.kennesaw.edu/~jgarrido/psim.html>

The rest of this paper include a short description of the OOSimL language, an overview of the its features, and an example of a simulation model developed with OOSimL.

2 The C++ Translator

The C++ OOSimL language translator is implemented as a one-pass language processor and that generates C++ source code. The generated code can be integrated conveniently with any C++ library. The translator itself is an executable program implemented in C++ that has been compiled (and linked) on Linux and Windows, using the GNU C++ compiler and associated software.

The language supports the reuse of C++ components. The run-time support of this simulation language is an extension of the Psim3 (a collection of

C++ classes) simulation package used with simulation models. The general translation (to C++ or Java code) of a OOSimL program is shown in Figure 1.

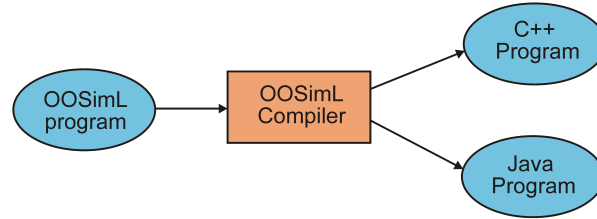


Figure 1: OOSimL Translation

After the OOSimL translation, a program needs to be compiled with the standard Java or C++ compilers (and linked) before execution. Eclipse is used in all development of OOSimL programs. This tool is one of the most widely known IDEs for program development, and is freely available. Kevin Sealy [8] [9] developed a plug-in for Eclipse that helps in the editing, translation, and compilation of OOSimL programs for generating Java code. For generating C++ code, the Codeblocks IDE is recommended.

3 The OOSIML Language

The general OOSimL syntax is briefly explained in [4] and is fully described in [5], [6], and [7]. The notation of the language is very high-level and similar to conventional pseudo-code syntax and standard object oriented statements that are much easier to read, understand, and modify than the syntax of standard object-oriented programming languages such as C++ and Java. The language features include statements for the timing, scheduling of processes, further manipulation of interacting processes, queues, resources, random number generation, and statistics.

Because Java and C++ programs present differences in their overall structure, when using OOSimL a programmer has to structure the program depending if the translator is to generate Java or C++ code.

For the version of OOSimL for generating C++ code, the following list includes the sequence of the program sections and the corresponding keyword with and brief explanation.

- **forward** references, which are references class declarations before they are defined.
- **global declarations**, which are used to declare global variables and normally be written outside of any class definition. The data items declared are accessible by any function. These global data items must be used with extreme caution because they cause difficulties in debugging and may cause synchronization problems with threads.

- **specifications**, which include the class definitions without function bodies. Only the function headers are defined. This is an enhanced feature of the language as it allows separation of class specification from class implementations.
- **implementations**, which only include the complete implementation of the functions specified in the classes (member functions) and non-member functions.

Important additional keywords used in the C++ version of OOSimL.

- **body**, which is used in the implementation of a class. For example,

```
class body Rec_engine as process is
```

- **ref**, which has two related uses.
 - To maintain compatibility with the OOSimL original version, the translator declares all object reference variables as pointer variables in C++, by default. The keyword **ref** is used to override this feature. For example,

```
define ref objTime of class Timer
```

- The keyword **ref** is also used for parameter declarations (maps to & in C++) to declare the parameter in pass-by-reference mode. For example,

```
function calc_alt parameters ref mydocq of class Tide
```

- **pointer**, which is used to declare pointer variables or parameters (maps to * in C++). The first declaration in the following example includes a definition of a pointer variable named *alpha* of type double; the second declaration defines a pointer variable parameter, *ttime*, of class *Timer*, to be passed by reference:

```
define pointer alpha of type double
```

```
parameters ref pointer ttime of class Timer
```

- **def**, which is used for referencing static members of a class (for use of scope resolution :: in C++). For example, the statement in the following line invokes function *mytide* of class *Cport*, and assigns the return value to variable *x*.

```
set x = call mytide def Cport
```

- **in**, which is used for referencing a member of an object that has been declared with the **ref** keyword. This referencing uses the dot notation in C++. For example, in the following statements, object variable *dateobj* is declared with the **ref** keyword. The second statement invokes function *xval* of an object referenced by *dateobj*.

```
define dateobj ref of class Tval
...
set y = call xval in dateobj
```

- **tracewrite**, which is used to write a text line to the *trace* output file.

```
tracewrite sname,
    " shipped 5 units from prod line"
```

- **statwrite**, which is used to write a text line to the statistics output file.

```
statwrite "Number of automobiles shipped: ", ship_count
```

- **file write**, which is used for output stream processing in output file manipulation. For example:

```
write file myoutfile "X = ", x, " y: ", y
```

- **file read**, which is used for input stream processing in input file manipulation. For example:

```
read file myinfile z
```

When using C++ as the target code in the development of a OOSimL model, the GUI and other graphical programming features of the model can be provided by using OpenGL, which is a standard and powerful framework available on Unix, Linux, and Windows. For simple models, only basic GUIs are normally used and GLUT is a good library.

4 Modeling Approach

With OOSimL, a model is composed of active or passive objects, each having attributes and behavior. Active objects have a life of their own and are called *entities* or *processes*. Passive objects only exhibit behavior when requested by another object. A dynamic system is modeled as a set of interacting processes. A process is an active object in the model and implemented as a thread object. One of the processes is the main process, which the starts the simulation.

The output of a simulation run will normally generate two output text files: one with the complete simulation trace, and the other one with the summary

statistics and performance metrics. The other types of output will usually be the various GUI boxes and graphical animations that the model implementation included.

For the generation of C++ code, the following sequence of parts defines the overall structure of simulation models.

- Specification of classes for passive objects needed by processes.
- Specification of classes for processes (active objects).
- Starting of a simulation run that will execute for a predetermined simulation period.
- Definition of queues that are used in the simulation models.
- Definition of resource pools and other passive objects.
- Invoke generation of random numbers, implicitly using a probability distribution.

For more detailed information on OOSimL, refer to [5]. The reference manual of the language is found in the OOPsim Web page.

5 Overview of OOSimL Statements

The general structure of a class definition in OOSimL is:

```

description
    . . .

class < class_name > is

    private

        data declarations (attributes)

        object reference declarations

        function definitions

    public

        data declarations (attributes)

        object reference declarations

        function definitions

endclass < class_name >

```

The following statement is used to set up a simulation with the title of the simulation project.

```

simulation title < variable_string >

```

The following statement starts a simulation run with the required simulation period.

```
start simulation with < num.variable >
```

The **start** statement starts execution of the main body of the process referenced. This command must be used after creating the referenced process.

```
start object < ref.variable >
start thread < ref.variable >
start self
```

The following statement makes a process wait for the specified time interval.

```
wait < ref.variable > for < time.variable >
wait self for < time.variable >
```

The following statement is similar to statement **wait**. It is used when a process is to carry out an activity for a specified interval of time.

```
hold < ref.variable > for < time.variable >
hold self for < time.variable >
```

The following statement schedules a process, after the specified time interval, or at a specified time instant. At that instant, the process will become the running process.

```
schedule < ref.variable > in < time.variable >
schedule self in < time.variable >
schedule < ref.variable > at < time.variable >
schedule self at < time.variable >
schedule < ref.variable > now
schedule self now
```

The statement **assign priority** is used to access the priority of a process. The general structure of the statement follows.

```
assign priority of < ref.variable > to < prio.var >
```

The statement **fix priority** is used to set the priority of a process. The general structure of the statement follows.

```
fix priority < prio.var > [ to < ref.variable > ]
```

The statement **terminate** will terminate the process referenced, or when used with the keyword **self**, the current process (the one executing the statement).

```
terminate < ref.variable >
terminate self
```

The following statement suspends the referenced process. The process will remain suspended until another process reactivates it.

```
suspend < ref_variable >
suspend self
```

The following statement reactivates the referenced process, which had been suspended before. The process becomes activate again immediately, or in the specified time interval from the current time. The process is then scheduled to carry out normal activities.

```
reactivate < ref_variable > now
reactivate < ref_variable > in < time_var >
resume < ref_variable > now
resume < ref_variable > in < time_var >
```

The **interrupt** statement sends an interrupt signal to the specified process using the indicated interrupt level. The interrupted process is rescheduled immediately so it can take appropriate action (i.e., execute its interrupt handler routine).

```
interrupt < ref_variable > with level < variable >
```

6 A Port Simulation Model

The following example was originally presented in [2] using Simula, then re-designed and implemented in OOSimL in [5] for Java code generation. The model presented here has been redesigned further for C++ code generation and using the additional keywords added to OOSimL.

The model represents a port system with arriving ships and various types of resources. Ships arrive at a port and unload their cargo. Ships request two tugboats and a pier (harbor deck). To leave, the ships request one tugboat. The activities of a ship: docking, unloading, and leaving, have a finite interval of duration. If the resources are not available, ships have to wait.

Condition: ships are only allowed to dock if two tugboats are available and the tide is not low. The tide changes every 13 hours. The low tide lasts for 4 hours. The tide is modeled as a process.

The implementation files of this simulation model in archive file `cport.zip`, this includes the source files, compiled files, and the executable. The following listing shows the first part of the source code in OOSimL of the file `Cport.osl`.

```
import <iomanip>
import "oosiml.h" // psim3
import "res.h"
import "condq.h"
/*
Port System with conditional waiting
File: Cport.osl
```


Ships arrive at a port and unload their cargo.
 Ships request 2 tugboats and a pier (harbor deck).
 To leave the ships request 1 tugboat. The
 activities of a ship, docking, unloading, and
 leaving, have a finite period of duration.
 If the resources are not available, ships have
 to wait.

Condition: Ships are only allowed to dock if
 2 tugboats are available and the tide is not low.
 The tide changes every 13 hours. The low tide
 lasts for 4 hours. The tide is a process.

```

00simL model, J. Garrido
for C++, Jan 2010.
*/
forward references
  class Cport
  class Arrivals
  class Ship
  class Tide
global declarations
variables
  // mean inter-arrival
  define static arr_mean of type double
  define static unloadmean of type double
  define static unloadstdev of type double
  define static numarrivals = 0 of type integer
  define static acc_wait = 0.0 of type double
  define static simperiod of type double
  define static close_arrival of type double
object references
  define static dockq of class Condq
  define static tugs of class Res
  define static piers of class Res
  define static run of class Simulation
  define static carrivals of class Arrivals
  define static cur_tide of class Tide
  define static port of class Cport
  // file for statistics
  define static statf of class ofstream
  // file for trace
  define static tracef of class ofstream
specifications
  description
    Main class: Cport */
class Cport as process is
public
  function initializer parameters portname
    of type string
  description
  function to evaluate condition;
  there must be atleast two tugboats
  available and the tide must not be low
  */
  function tugs_low_tide return type boolean

```

```

function Main_body
endclass Cport
description
This class represents the behavior of the
tide object, which changes state from
low tide to high tide and from high to
low. The tide changes every 13 hours
(time units). The low tide lasts for
4 hours. The tide is modeled as a process.
*/
class Tide as process is
private
variables
    define lowtide of type boolean
    define tidename of type string
public
function initializer parameters tname
    of type string
function get_lowtide return type boolean
function Main_body
endclass Tide
class Arrivals as process is
private
variables
    define ss of type string
    define aunloadmean of type double
    define unloadstd of type double
object references
    define next of class NegExp // Random gen
    define unload of class Normal // Random gen
public
function initializer parameters
    arrname of type string, arr_mean
    of type double,
    umean of type double, unlstd
    of type double
function Main_body
endclass Arrivals
description
Activity of a ship object:
1. conditional wait for tugboats and
low tide; 2. dock; 3. unload;
4. undock; 5. leave (terminate)
Assumptions: 1. dock interval is constant;
2. undock interval takes about 65
*/
class Ship as process is
private
variables
    // ship start time of wait
    define startw of type double
    define unload of type double
    define simclock of type double
    define shipname of type string
object references
    define static ctide of class Tide
public

```

```

variables
    define shipnum of type integer // ship number
function initializer parameters shname
    of type string, unloadper
    of type double
function Main_body
endclass Ship
//
implementations
// class implementation follows
class body Cport as process is
function initializer parameters
    portname of type string
super using portname is

```

The complete implementation including the output files generated by a sample simulation run is found in the OOPsim Web page.

7 Conclusion

The OOSimL language was designed for developing discrete-event simulation models. The simulation principles and concepts used with the OOSimL language and the simulation model development are explained in detail elsewhere. The C++ version of OOSimL provides a valuable type of flexibility and expressive power to C++ developers. The language statements have been described in this paper, together with a simulation model designed for translation to C++.

Development of simulation models in OOSimL is carried out with an IDE such as Codeblocks. The OOSimL compiler (translator), run-time libraries, several sets of documentation, and collection of examples can be freely downloaded from the Psim web page. OOSimL is part of the OOPsim project and its development has been partially supported by an NSF CPATH grant.

References

- [1] Birtwistle, Graham M., Ole-Johan Dahl, Bjoern Myhrhaug, and Kristen Nygaard. *SIMULA Begin*. Petrocelli/Charter, New York, 1975.
- [2] Birtwistle, Graham M. *DEMOS: Discrete Event Modelling on SIMULA*. MacMillan Press, London, 1979.
- [3] Nell, Dale, John A. McCormick, and Chip Weems. *Programming and Problem Solving with Ada95*. Second Ed. Jones and Bartlett, Sudbury, MA, 2006.
- [4] Garrido, José M. “OOSimL: An Object Oriented Discrete-Event Simulation Language for Computing Education” In *HSC2009 The Huntsville Simulation Conference*. Huntsville, Alabama. October 27-29, 2009.

- [5] Garrido, José M. *Object-Oriented Simulation: A Modeling and Programming Perspective*. Springer, New York, 2009.
- [6] Garrido, José M. *The OOSimL Language: Part 1 Programming Aspects*. Department of Computer Science and Information Systems. Kennesaw State University. Updated April 2009.
- [7] Garrido, José M. *The OOSimL Language: Part 2 Simulation Aspects*. Department of Computer Science and Information Systems. Kennesaw State University. Updated April 2009.
- [8] Garrido, José M. and Kevin Sealy. *Using Eclipse with OOSimL: For Java Code*. The OOPSim Project. Department of Computer Science and Information Systems. Kennesaw State University. December 2009.
- [9] Garrido, José M. and Kevin Sealy. *Using Eclipse with OOSimL: For C++ Code*. The OOPSim Project. Department of Computer Science and Information Systems. Kennesaw State University. December 2009.
- [10] Garrido, José M. *Object-Oriented Discrete-Event Simulation with Java*. Kluwer Academic / Plenum Publishers. New York, 2001.
- [11] Garrido, José M. *Practical Process Simulation Simulation Using Object Oriented Techniques and C++*. Artech House. Boston, 1999.
- [12] Meyer, Bertrand. *Eiffel: The Language*. Prentice Hall International, Hemel Hemstead, 1992.
- [13] Savitch, Walter. *Problem Solving with C++: The Object of Programming*. Fourth Edition. Addison Wesley Pearson, 2003.