Using Specification and Description Language to define and implement discrete simulation models

Pau Fonseca i Casas
Universitat Politècnica de Catalunya
Jordi Girona 1-3
08034, Barcelona, Catalunya, SPAIN
(+34)934017732
pau@fib.upc.edu

Abstract

The formal languages become important tools since they allow the complete understanding of the model and help in its implementation. However only a few simulation tools allow an automatic execution of a simulation model based in a formalization of the system.

Specification and Description Language is a modern object oriented graphical formal language that allows the definition of distributed systems. It has focused on the modeling of reactive, state/event driven systems, and has been standardized by the International Telecommunications Union (ITU) in the Z.100. Since it is a graphical formalism simplifies the understanding of the model.

In this paper we show how we can use Specification and Description Language to represent a discrete simulation model. We propose a solution, implemented in SDLPS, regarding how to manage the time in Specification and Description Language. Also, we show how SDLPS infrastructure allows a distribute simulation of the models.

Keywords: SDL, Simulation, Formalisms.

1. INTRODUCTION

The construction of a simulation model sometimes lacks in the formalization process needed to understand the model before any implementation. This model relations and hypotheses understanding helps in the implementation process and in the communication between the different personnel involved in the model construction. Also the formalization of a system can be considered a product itself [1]. Not only this representation of the model is useful for communication purposes, but also simplifies the validation process. As Sargent states [2], "Computerized model verification ensures that the computer programming and implementation of the conceptual model are correct. The major factor affecting verification is whether a simulation language or a higher level programming language such as FORTRAN, C, or C++ is used. The use of a special-purpose simulation language generally will result in having fewer errors than if a general-purpose simulation language is used, and using a general-purpose simulation language will generally result in having fewer errors than if a general purpose higher level programming language is used."

Some tools have been implemented in order to execute the model from its representation. As an example we can cite simulation environments, like ATOM [3], CoSmOs [4] or CD++ [5],[6] that allows the simulation execution from a representation of a model based on DEVS formalism. The proposed infrastructure allows the definition (and execution) of a simulation model following the Specification and Description Language (SDL). Since SDL allows the definition of distributed systems the resulting model can be executed over different computers without any modification of the model definition.

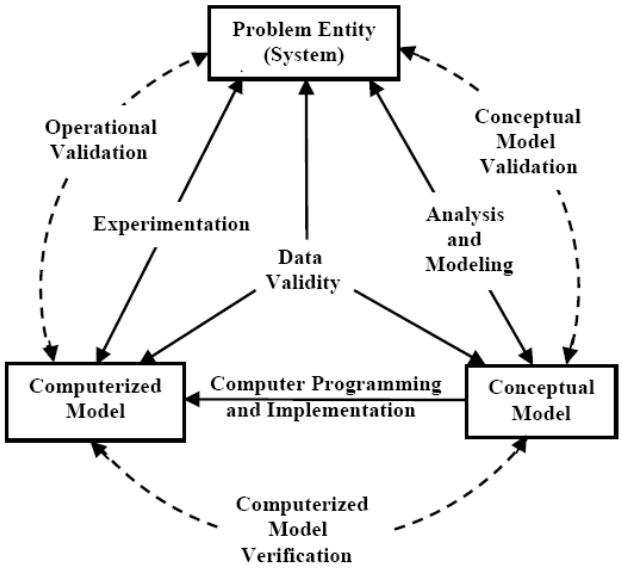


Figure 1. Simplified version of the modeling process **[2]**.

The infrastructure is implemented in C++ and the models are represented using SDL (through XML files).

2. SPECIFICATION AND DESCRIPTION LANGUAGE

SDL is the acronym of Specification and Description Language; an object-oriented, formal language defined by the International Telecommunication Union – Telecommunication Standardization Sector (ITU–T) (formerly *Comité Consultatif International Télégraphique et Téléphonique* [CCITT]) as Recommendation Z.100 [7]. The language is designed to specify complex, event-driven, real-time, interactive applications involving many concurrent activities using discrete signals to enable communication [8], [7].

SDL is a powerful and modern language widely used in different areas, not only in simulation area. It has been standardized by the International Telecommunications Union (ITU) in the Z.100, and can be used easily in combination with UML. The definition of the model is based on different components:

Structure: system, blocks, processes and processes hierarchy.

Behavior: defined through the different processes.

Data: based on Abstract Data Types (ADT).

Communication: signals, with the parameters and channels that the signals use to travel.

Inheritances: to describe the relationships between, and specialization of, the model elements.

The language has 4 levels (i) System, (ii) Blocks, (iii) Processes and (iv) Procedures, as we can see in the next figure.

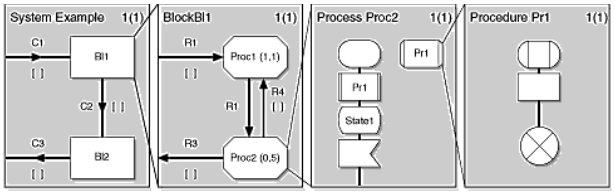


Figure 2. SDL levels **[9]**.

2.1 SDL system diagrams

System diagrams represent all of the objects that make up a model and the communication channels between them. A system is the outermost agent that communicates with the environment. The next figure shows a system containing three blocks [12].

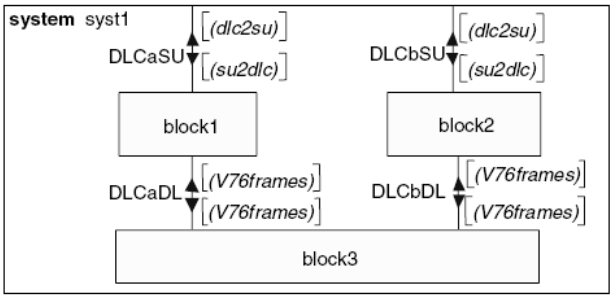


Figure 3. System diagram **[8]**.

2.2 SDL Blocks diagrams

The next stage in SDL specification is the construction of a blocks diagram for each of the different block defined in the system diagram.

The following is the blocks diagram for the block1 and block3 elements defined in Figure 3:

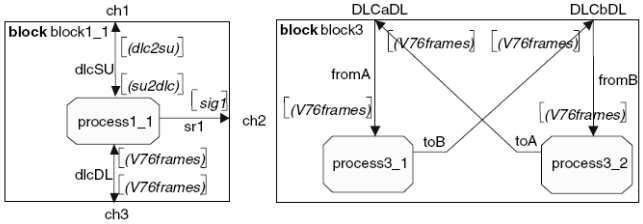


Figure 4. SDL Block diagram **[8]**.

Each rectangle represents an object. The lines that join the objects are the communication channels (bidirectional or unidirectional communication elements). The channels are joined to the objects through ports. Ports are very important elements for implementing and reusing objects, since they ensure the independence of the different objects. An object only knows its own ports, which are the doors through which it communicates with its environment. An object only knows that it sends and receives events using a specific port.

Each block has a name specified by **block** keyword. The blocks diagram contains a number of processes and may also possibly contain other blocks (but not mixed with processes). Processes communicate via **signal routes**, which connect to other processes or to **channels** external to the block

2.3 SDL processes

The processes describe more specifically the behavior of the block. Each one of the processes of the block has one or more states. For each one of the states of a process, SDL describe how it behaves if different events occur. An object may react differently to an event depending on the port that sends it. The process is basically specified using graphical elements that describe operations or decisions.

Table 1. Some important SDL process elements.

	**Start.** Allows defining the first operations to be executed that conducts to the initial state of a process.
	**State.** A state element contains the name of a state. All diagrams start and end with **state** elements. One process can start with the **start** element.
	Input. These elements describe the kind of signals that can be received depending on

	the state and the numbers of the ports that these events travel through. All branches of a specific **state** start with an **Input** element, since an object changes its state only after a new signal is received.
	Create. This element allows the creation of an object.
	Task. This element allows the definition of assignments, assignments attempts or the interpretation of informal texts.
	Procedure call. These elements perform actions that do not generate delays in the model (delays are modeled through the event processing time parameterization).
	Output. These elements describe the kind of signal to be sent and the port used. Other attributes of the event can also be detailed (priority, execution time, etc.).
	Decision. These elements describe bifurcations. Their behavior depends on how the related question is answered.

Table 1 shows the elements used in the SDL processes diagrams implemented in the system. The next figure shows an example of a SDL process.

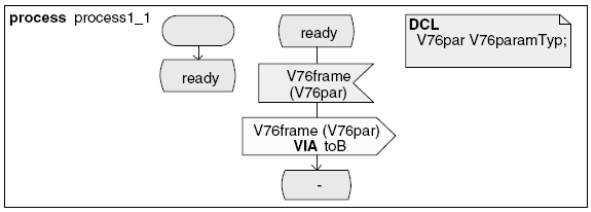


Figure 5. SDL process diagram**[8]**.

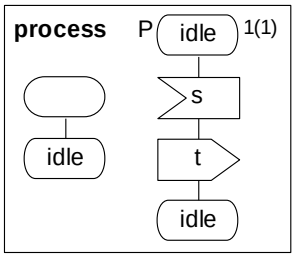
2.4 SDL procedures

The last level of the SDL method is the description of the different procedures that appear in the SDL diagrams. These diagrams help describe and specify the model by detailing its most important aspects at the needed level, depending on the target of the specification requirements.

To know more about SDL the recommendation Z.100 [7] can be consulted, also a lot of information can be reviewed in the www.sdl-forum.org website or in [10], [11] or [8], among other sources.

2.5 SDLP-PR

A no graphical SDL exists (SDL/PR). SDL/PR is not used in this paper. The power of the two SDL representations is equivalent [7]. In SDLPS we use a XML representation of SDL. We are using this instead SDL/PR because it is easiest to manage, transform and represent XML instead the plain text file that defines SDLP-PR. Also XML allows defining special tags that are not part of the model, useful to define representation model parameters (position of the blocks in the layout, as example).




```
process P;
  start;
  nextstate idle;
  state idle;
    input s;
      output t;
      nextstate idle;
  endstate idle;
endprocess P;
```

Figure 6. This figure shows the relation between the no graphical SDL (SDL/GR) and the graphical SDL (SDL/PR).

3. TIME MANAGEMENT IN SDLPS, DELAYING SIGNALS

Different paradigms exists to implement a simulation engine; the three more widely used are, (i) event scheduling, (ii) activity scanning and (iii) process interaction [12],[13], [14]. SDLPS uses an event scheduling simulation engine; however this is transparent to the user, since all the models are defined using SDL language.

In a discrete simulator, to completely define the behavior of a model is needed to describe the time related to the execution of each one of the different events that manage its evolution. Usually each kind of event owns its specific probability distribution, which manages when this event must be executed. In an event scheduling simulator, the engine manages the time of all the events, and decides where and when all those events must be send (to other simulation elements, agents in a SDL model).

SDL have two main structures to manage time, **Timers** and **Delaying Channels** [7]. The problem regarding how to manage time in SDL has been studied for several authors [15], [16]. Specifically in [16] is presented an extension that defines three kinds of transitions, (i) eager, (ii) lazy and (iii) delayable. From a point of view of a discrete simulator, all the transitions can be considered delayable, since all the transitions have a time defined (remark that an eager transition is equivalent to a delayable transition with the temporal condition set to now=x [16]).

In SDLPS all the signals carry the parameter defined in the structure represented in the Figure 7. The elements are: (i) *ExecutionTime*, representing the time when the event must be executed. (ii) *Priority*, the priority of the event, used to break a possible simultaneity of events. (iii) *CreationTime*, representing the time when the event is created. (iv) *Id*, an identifier of the event. (v) *Time*, the clock of the process. (vi) *Destination*, the final destination (process PId) of the signal.

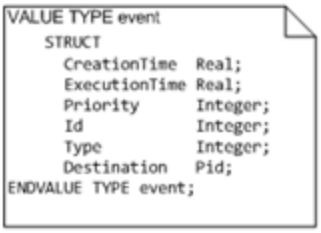


Figure 7. Structure related to the SDLPS signals.

Parameter *event* is needed by SDLPS engine in order to delay or sort by priority the different signals. When a signal is received SDLPS use its *event* parameter to manage the time and the priorities of the signal. In SDLPS context we can use extension elements to define this parameter related to the signal, as we can see in Figure 8. Not all the parameters of *event* structure must be defined, only those needed to fully define the behavior of the model.

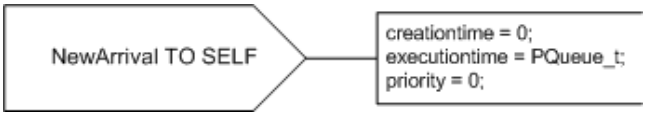


Figure 8. Defining the delay, and other parameters, of the signal using SDL time extensions.

These extensions are now under discussion on the ITU-T Study Group 17 (http://www.itu.int/ITU-T/studygroups/com17/index.asp) to be included in the next release of the standard.

4. SDL FORMALIZATION OF A SIMULATION MODEL

As an example we formalize a GG2 model (two servers and a single queue). The first level (Figure 9) represents the interaction that users can do with the model. In that case there is no interaction between the model and the environment. Going inside the GG2 block we can see its inner structure (Figure 10), two servers and a single queue).

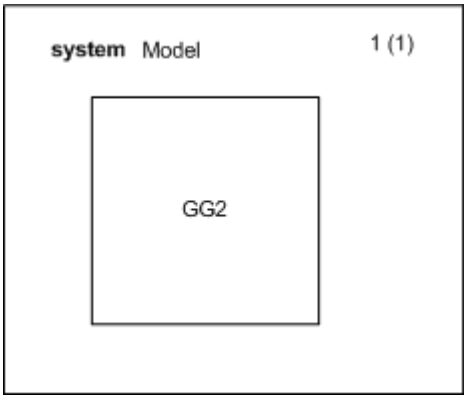


Figure 9. GG2 model system diagram. The GG2 model shows no interaction with the environment.

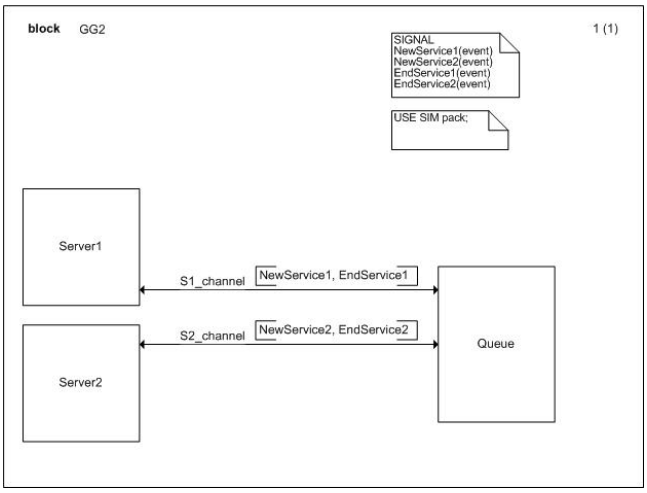


Figure 10. GG2 model blocks diagram. This diagram shows the inner structure of the model, two queues and a server.

The structure and the behavior for the server are represented in the next two figures (Figure 11 and Figure 12).

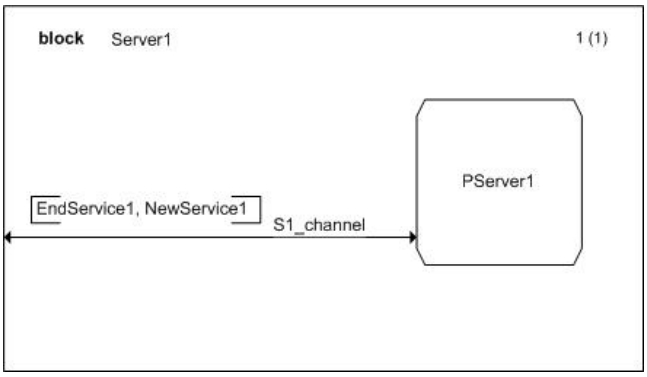


Figure 11. Server1 block processes diagram.

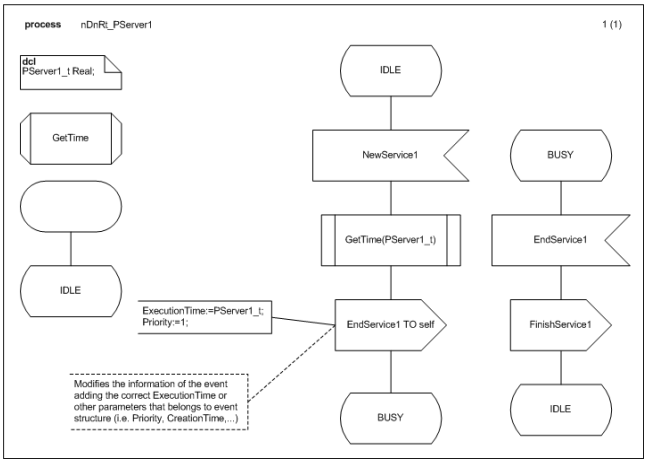


Figure 12. PServer1 process

In server process diagram (Figure 12) the start operation defines the initial state (IDLE). Two states are defined (IDLE and BUSY). The events that modify the state of the server are *NewService* (from IDLE to BUSY) and *EndService* (from BUSY to IDLE).

The last level of the SDL formalism allows the definition of the procedures of the model. As an example the SDL representation for the procedure *GetServiceTime* is:

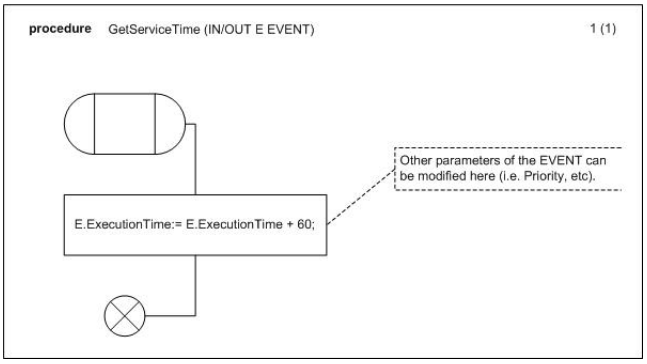


Figure 13. Procedure *GetServiceTime*, considering service time constant of 60 time units.

As we see in this section, SDL formalization of a simulation model is divided in different diagrams. One of the main advantages of the SDL language is that we don't need to show the complete specification to all the specialists that are working in the model construction. For instance in a large industry, the main process of the industry can be represented by the system block (and its inner blocks) showing the main elements of the model and its relations. To understand the behavior of a specific element, we can go further, to the process diagrams and the procedures diagrams that show its complete definition.

5. XML REPRESENTATION OF THE MODEL, SDL/XML

The XML markup language is used to represent the model. This representation is named SDL/XML. Although a no-graphical version of the SDL language exists (SDL/PR), the use of XML simplifies the management of the language structures and its transformation and manipulation in the SDLPS infrastructure. Also SDL/XML allows adding information about the graphical representation of the different simulation elements (represented by SDL agents).

System block is represented by the <*system*> element. This is the root element of the SDL/XML file. Inside this element we find the channels and the system blocks that can be defined in this block. In our example no channels are defined at this level, see Figure 9. For all the blocks different channels can be defined. The channels allow the communication between the different elements that can be executed in different computers. The XML code representing the channels is shown in the next lines. Note that each channel have a name and a *start* and *end* attribute; dual="yes" means that the channel is bidirectional. All the channels describe the kind of events that can travel through it. At the moment SDL/XML do not describe if the event is related to the input or output.

```
<channels>
  <channel name="MainS1" start="BlockServer1"
end="BlockQueue" dual="yes">
    <!--The events that use the channel.-->
    <event name="FinishService1"></event>
    <event name="NewService1"></event>
  </channel>
  <channel name="MainS2" start="BlockServer2"
end="BlockQueue" dual="yes">
    <!--The events that use the channel.-->
    <event name="FinishService2"></event>
    <event name="NewService2"></event>
  </channel>
</channels>
```

Figure 14. SDL/XML definition for the channels.

The <*block*> XML element allows the complete description of the Block SDL element. As we can see in the next code a block can contain a process. Each process can define variables, <*DCL*> element, and procedures, <*procedures*> element. The main elements of the process are related with the process definition. Each process starts in a state and defines the different operations. The <*start*> element defines the initialization operations.

```
<block id="2" name="Server1" implementation="" IP="192.168.1.5"
portRead="8687">
  <channels>
    <channel name="S1Ch" start="BlockServer1" end="PServer1"
dual="yes">
      <!--The events that use the channel.-->
      <event name="FinishService1"></event>
      <event name="NewService1"></event>
    </channel>
  </channels>
  <process id="1" name="PServer1" implementation=""
IP="192.168.1.5" portRead="8687">
    <!—Process variable declarations.-->
    <DCLS>
      <DCL name="PServer1_t" type="double" value=""></DCL>
    </DCLS>
    <!--Procedures definition.-->
    <procedures>
      <procedure id="1" name="DelayTimeSrv1" implementation="">
        <params>
          <param name="TimeSrv1_t" type="double" defvalue=""
ref="yes"></param>
        </params>
        <body>
          <task id="1" name="">TimeSrv1_t=60;</task>
        </body>
      </procedure>
    </procedures>
```

```
<!--Process operations definition.-->
  <start>
    <setstate id="1" name="IDLE"></setstate>
  </start>
  <state name="IDLE">
    <input id="1" name="NewService1"></input>
    <procedurecall id="2" name="DelayTimeSrv1">
      <param name="TimeSrv1_t" value="PServer1_t"></param>
    </procedurecall>
    <output id="3" name="EndService" self="yes" via="">
      <param name="delay" value="PServer1_t"></param>
      <param name="priority" value="0"></param>
    </output>
    <setstate id="4" name="BUSY"></setstate>
  </state>
  <state name="BUSY">
    <input id="1" name="EndService"></input>
    <output id="2" name="FinishService1" self="" via="S1Ch">
      <param name="delay" value="0"></param>
      <param name="priority" value="0"></param>
    </output>
    <setstate id="3" name="IDLE"></setstate>
  </state>
 </process>
</block>
```

Figure 15. SDL/XML definition for the blocks.

This XML code defines the *Server1* block defined in the Figure 11 and his process *PServer1* defined in the Figure 12.

All the elements can be hardcoded using the attribute *implemented*. If *implemented* is set, the definition of the element is ignored, and the events are received by this piece of code (C++ code, program or DLL). This allows the reuse of legacy simulation models or the implementation of specific parts of the models that we do not what to represent using SDL. *IP* attribute and *port* attribute must be defined in order to specify where this block or process is running.

6. SDLPS ARCHITECTURE

SDLPS is implemented in C++ and intended to allow the distribute execution of different SDL blocks or processes in different machines. Each one of the different blocks implements a port and a set of input and output channels that can be used to communicate with the other model blocks.

In SDLPS each process and block of the model must be assigned to a specific machine with a specific IP and port. In Figure 16 a representation of the architecture is shown. Each one of the different block are used to send the signals to its correct destination. Finally, when a signal is received by a process block the execution of the model begins. Since the code represented by the user (embedded in the tasks or decision SDL blocks) depends on the model, this code must be compiled once the model is defined. This compilation generates *SDLCode.dll*. This DLL, that is the same for all the SDL process (hence equal in all the machines), contains all the methods needed to execute the model obtained from the SDL definition of the model. In the current version of the infrastructure *gcc* compiler is used. SDLPS allows the configuration of the compiler (the location of gcc.exe), compile and link the DLL.

It is important to remark that although SDLPS generates code (SDLCode.DLL) in order to be able to execute the code contained in the task elements, is not a code generator system. SDLPS is a simulator capable to perform the simulation directly using the DLL that represent the task code.

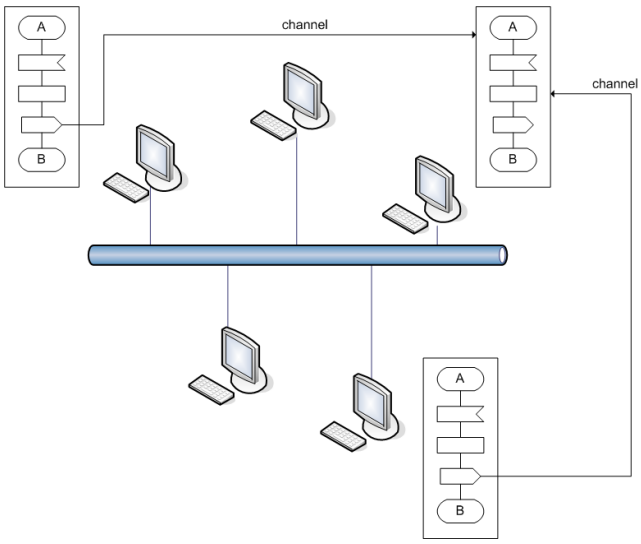


Figure 16. Distributed model architecture. Each process of the model can be executed in a different machine.

As we said previously the processes and the procedures can use native C++ functions defined in the SDLPS environment. These functions can be used to send information of the simulation execution to other environments or to use legacy code of specific simulation models. This is done through the specialization of the class *CSDLOperationTask* that defines the structure for the Task operation.

If we want to use C++ code inside our simulation model, the *implementation* tag of the SDL/XML can define the class that must be used to execute this piece of code. As an example, if we have a class that allows sending information to a remote server, we can use it in the model defining its *implementation* tag as we can see next:

implementation="CSDLOperationProcedureCallReport"

CSDLOperationProcedureCallReport class implements the execute method that defines what to do with the signals received. In this case sends statistical information regarding the signal to a remote client that manages this information.

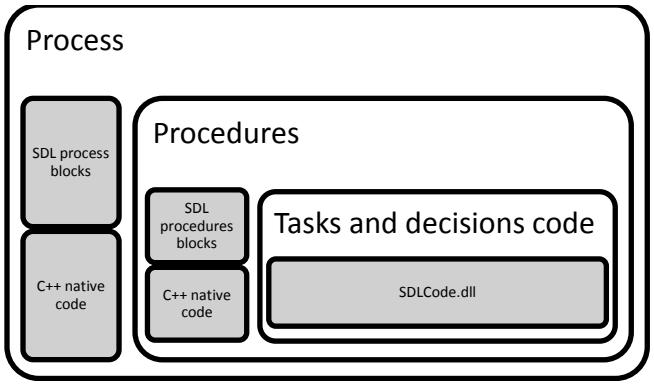


Figure 17. SDLPS process architecture

This approach has two main advantages: (i) the compiler is not needed and (ii) the execution can be faster. However with this approach new programming is needed and also no specification of this piece of code is defined using SDLPS. In Figure 17 the architecture of the process SDLPS environment shown.

The application GUI is shown in Figure 18. SDLPS is intended to capture the events and process it. Other applications can be connected to it in order to allow a representation of the simulation model or statistical acquisition.

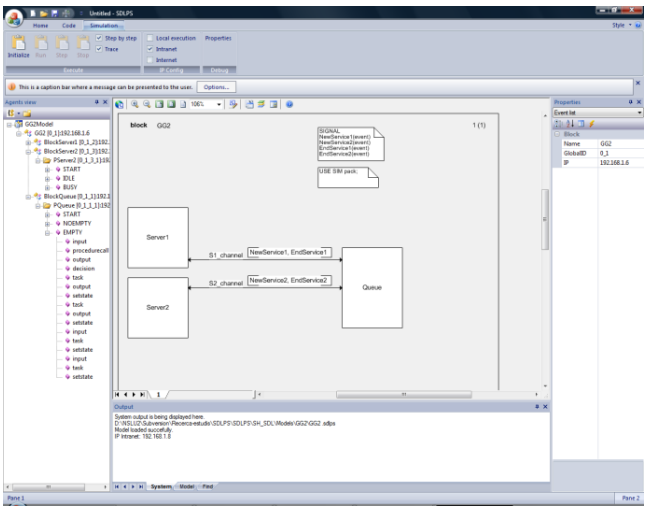


Figure 18. SDLPS GUI.

Since SDLPS allows the execution of the SDL model in a distributed environment is needed to implement some time management mechanism. The proposed mechanism, which is implemented in SDLPS, uses a conservative approach, described in the next section.

6.1 Time management

The main objective of SDLPS is to allow the simulation of a model from an SDL specification; the second objective is to perform a distributed simulation of this model. To allow this a conservative approach for a distribute simulation model has been implemented. Each one of the different channels that connect the elements of the model implements an event list. The element (process or a block, or other computer program if have a specific implementation) takes the event that have the smallest timestamp in all the incoming channels. This method can be reviewed in [17]. The problem is that some cases can cause a deadlock. One of the common approaches to avoid the deadlock in a conservative algorithm is to send null events to other elements [17]. In our approach all the SDLPS's instances send the events to a local *CSDLEngine* that manages the local time of each sub-model. All the different *CSDLEngines* have the main objective of maintain the knowledge of the time of all the channels of the model. With this knowledge we know the events with the smallest timestamp that are safe to be processed, avoiding the deadlocks. Looking more in detail the proposed algorithm, three different scenarios have been detected.

First, no events exist in any of the different channels of a *CSDLAgent* (a *CSDLProcess* or *CSDLBlock*). In that case is needed to inform to the local *CSDLEngine* that no events exist in the object. Local *CSDLEngine* informs all the other *CSDLEngine* of the distributed model.

Second, the channel with smallest timestamp has events. In that case it is safe to process the events.

An Third, the channel with smallest timestamp does not have events, but other channels have events. In this case, the *CSDLEngine* decides if this event (the first event of the channel that do not have the smallest timestamp) is safe or not to be processed, since the *CSDLEngine* stores what is the time of the event with the smallest timestamp.

Some different approaches exist to manage the problem stated in the third case. Some of the approaches to break this deadlock use some knowledge of the model [17]. In the SDLPS system we use a conservative approach, meaning that we wait until *CSDLEngine* assures that the smaller timestamp to be processed in the local agent is one who belongs to one channel with events. The proposed conservative algorithm can be changed for other algorithms thanks the modular development of the tool.

7. CONCLUSIONS AND FUTURE WORK

This paper presents an infrastructure capable to perform a simulation of a model represented using Specification and Description Language. Also solution to manage time in SDL is proposed, adding *event* structure to all SDL signals.

Since the program needs to manage the Specification and Description Language model representation, a XML representation of SDL is used. We use XML instead SDL/PR because XML simplifies the manipulation of the model representation in SDLPS. SDL/XML representation allows the definition of elements implemented using a DLL or C++ classes, allowing the use of legacy simulation

models or other elements that we don't want to represent in the specification of the model.

This infrastructure allows a distributed simulation of the different elements defined in SDL. SDLPS manages the time and the resources needed to execute the simulation. The user only must describe the behavior of the model following SDL, without the need of think if the execution will be local or shared over different computers. In this first release of SDLPS not all the structures are implemented. Specifically in SDLPS we do not have an implementation of Timers, and the events cannot carry other parameters than simple types (structures are not allowed yet, with the exception of *event* structure reviewed in this paper). In future releases of SDLPS we plan to add fully compliance to SDL 2000 and the future release of the standard SDL.

This methodology and infrastructure has been used during several years successfully. As an example of the application of this methodology we can mention the simulation of the Almirall Prodesfarma enterprise [18], or the simulation of the Barcelona international Airport. More recently and using the infrastructure we can mention, in the environmental area, the wildfire [19] or slap avalanches [20] modeling.

References

[1] Brade, D. (2000). Enhancing modeling and simulation accreditation by structuring verification and validation results. In J. A. Joines, R. R. Barton, K. Kang, & P. A. Fishwick (Ed.), *Winter Simulation Conference.*

[2] Sargent, R. G. (2007). VERIFICATION AND VALIDATION OF SIMULATION MODELS. In S. G. Henderson, B. Biller, M.-H. Hsieh, J. Shortle, J. D. Tew, & R. R. Barton (Ed.), *Proceedings of the 2007 Winter Simulation Conference*. IEEE.

[3] De Lara, J., & Vangheluwe, H. (2002). ATOM³: A Tool for Multi-formalism Modelling and Meta-modelling. *4th International Conference on Enterprise Information Systems ICEIS 2002.*

[4] Sarjoughian, H. S., & Elamvazhuthi, V. (2009). CoSMoS: A Visual Environment for Component-Based Modeling, Experimental Design, and Simulation. *International Conference On Simulation Tools And Techniques.* Rome, Italy.

[5] Wainer, G., & Chen, W. (2003, November). A framework for remote execution and visualization of Cell-DEVS models. *Simulation: Transactions of the Society for Modeling and Simulation International* , 626-647.

[6] Wainer, G. (2002). CD++: a toolkit to develop DEVS models. *Software, Practice and Experience , 32* (3), pp. 1261-1306.

[7] Telecommunication standardization sector of ITU. (1999). *Specification and Description Language (SDL).* Retrieved April 2008, from Series Z: Languages and general software aspects for telecommunication systems.: http://www.itu.int/ITU-T/studygroups/com17/languages/index.html

[8] Doldi, L. (2003). *Validation of Communications Systems with SDL: The Art of SDL Simulation and Reachability Analysis.* John Wiley & Sons, Inc.

[9] IEC. (n.d.). *SDL Tutorial*. Retrieved May 2010, from IEC: http://www.iec.org/online/tutorials/sdl/topic04.html

[10] *SDL Tutorial*. (n.d.). Retrieved January 2009, from IEC International Enginyeriing Consortium: http://www.iec.org/online/tutorials/sdl/

[11] Reed, R. (n.d.). *Re: SDL-News: Request for Help: Initialisation of Pids*. Retrieved April 2009, from SDL- FORUM: http://www.sdl-forum.org/Archives/SDL/0032.html

[12] Law, A. M., & Kelton, W. D. (2000). *Simulation Modeling and Analysis.* McGraw-Hill.

[13] Guasch, A., Piera, M. À., Casanovas, J., & Figueras, J. (2002). *Modelado y simulación.* Barcelona, Catalunya/Spain: Edicions UPC.

[14] Fishman, G. S. (2001). *Discrete-Event Simulation: Modeling, Programming and Analysis.* Berlin: Springer-Verlag.

[15] Bozga, M., Graf, S., Mounier, L., Kerbrat, A., Ober, I., & Vincent, D. (2000). SDL for Real-Time: What Is Missing ? *SAM'2000*. Grenoble, France.

[16] Bozga, M., Graf, S., Mounier, L., Ober, I., Roux, J.-L., & Vincent, D. (2001). Timed Extensions for SDL. *Proceedings of SDL-Forum'01*. Copenhagen, Denmark.

[17] Fujimoto, R. M. (2001). Parallel simulation: parallel and distributed simulation systems. *Winter Simulation Conference*, (pp. 147-157).

[18] Casanovas, J., Perez, W., Montero, J., & Fonseca, P. (1999). Simulation of reception, expedition and picking areas of a pharmaceutical products plant. In J. Fuertes (Ed.), *Emerging Technologies and Factory Automation*, (pp. 321-328). Barcelona, Catalunya, SPAIN.

[19] Fonseca i Casas, P., & Casanovas, J. (2005). Simplifying GIS data use inside discrete event simulation model through m_n-AC cellular automaton. *Proceedings ESS 2005.*

[20] Fonseca i Casas, P., & Rodríguez Fontoba, S. (2007). Using GIS data in a m:n-ACk cellular automaton to perform an avalanche simulation. *Geographical Information Science Research UK Conference 2007*. National University of Ireland Maynooth.

Biography

Pau Fonseca i Casas is a Professor of the Department of Statistics and Operational research of the Technical University of Catalonia. He obtained his degree and master degree in computer engineering on 1999 and his Ph.D. on 2007 from Technical University of Catalonia. He works in the LCFIB (Barcelona informatics school laboratory) developing Simulation projects since 1998, also is member of LogiSim, group dedicated to the research and simulation tools and projects development. His website is http://www-eio.upc.es/~pau/ .