Definition of Virtual Reality Simulation Models Using Specification and Description Language Diagrams

Pau Fonseca i Casas[1], Xavier Pi[2], Josep Casanovas[1], and Jordi Jové[3]

[1] Universitat Politècnica de Catalunya, Barcelona-Tech, Barcelona, Spain
{pau,josepk}@fib.upc.edu
[2] Sinetric systems, Barcelona, Spain
xpi@sinetric.com
[3] Arbora-Ausonia, Barcelona, Spain
jove.j@arbora-ausonia.com

Abstract. A full representation of a simulation model encompasses the behavior of the elements that define the model, the definition of the probability distributions that define the delays of the events that control the model, the experimental framework needed for execution, and the graphical representation of certain model elements. This paper aims to use specification and description language to achieve a full model representation by adding two extensions to the language, which allows for a complete and unambiguous definition of a discrete simulation model that is similar to a common discrete operations research simulation tool.

Keywords: Formal Languages, Specification and Description Language, SDL, Operations Research, Discrete Simulation, Virtual Reality.

1 Introduction

A discrete simulation model can be described using formal languages that allow a clear separation between the definition of the model and its implementation. However, for discrete simulation (for operations research), the use of formal languages is desirable but not common. Many of the important discrete simulation tools do not work with a formal language and often are based on proprietary syntax and tools. This proprietary representation of the simulation model often presents a challenging problem for transforming the model to a different implementation.

This paper aims to use a formal language, the Specification and Description Language [1], to achieve a complete representation of a simulation model. This representation encompasses the behavior and structure of the model as well as the graphical representation of the model execution, which simplifies the model validation as suggested in [2]. We developed two simple extensions to the 2000 version of the language (SDL-2000), which allow for a complete definition of a

F. Khendek et al. (Eds.): SDL 2013, LNCS 7916, pp. 258–274, 2013.
© Springer-Verlag Berlin Heidelberg 2013

discrete simulation model that is similar to common discrete operations research simulation tools; however, we use an unambiguous graphical and standard formal language to improve the model description and reuse, while maintaining the benefits of a formal language as suggested in [3]. To achieve this objective, it is necessary to define the behavior of the elements that define the model: the characterization of the probability distributions that define the delays of the different events that control the model, the experimental framework for execution, and the structures for model representation.

Operations research simulation tools apply different paradigms to represent the real world. The discrete-event paradigm includes classical languages, such as GPSS/H [4,5] and SLAM II [6]. These tools have features that are similar to Simprocess [7], Arena [8], Simio [9], and Simul8 [10]. The paradigms that are usually represented with these tools include process-interaction or event-scheduling [11,12]. Other simulation tools do not exactly follow these paradigms. For example, Witness [13] constructs processes using push/pull rules for the different elements in the model. This type of software often allows for **if... then... else rules** for the definition of resources and attributes and allows for the use of dynamic link libraries (DLLs) to use specialized code defined with C++, C# or Visual Basic.

These software tools always allow for a complete definition of the model behavior, structure, time and representation.

Similar to the current paper, there are certain programming libraries and infrastructures that allow for defining a simulation model following a formal language. The tools related to DEVS [14] and PetriNets formalisms [15,16] often represent an excellent alternative to define and implement a simulation model with the previously mentioned simulation tools. An interesting example of extending Petri Nets can be reviewed on [17] . For the DEVS tools and infrastructures, a non-exhaustive list can be reviewed in [18]. For Petri Nets, a similar list is available in [19].

CD++ [20] and DEVSJAVA [21] are examples of DEVS infrastructures. CD++ is mainly a toolkit for Discrete-Event modeling and simulation and the environment is based on the DEVS (Discrete-Event systems Specifications) formalism. Currently, a plug-in exists that allows for using a graphical interface with the Eclipse platform. Figure 1 includes the CD++ plug-in on the eclipse platform.

One of the strengths of DEVS is that it supports the transfer of models from one concrete tool to another (due to the use of XML). Several attempts have been made to define a standard XML representation for DEVS with a complete and common XML schema [22,23].

Specification and Description Language also has different tools that allow for the implementation of a simulation model [24,25,26]. These tools allow the generation of code from the model representation. In this study, we use *Specification and Description Language Parallel Simulator*, SDLPS [23,27].

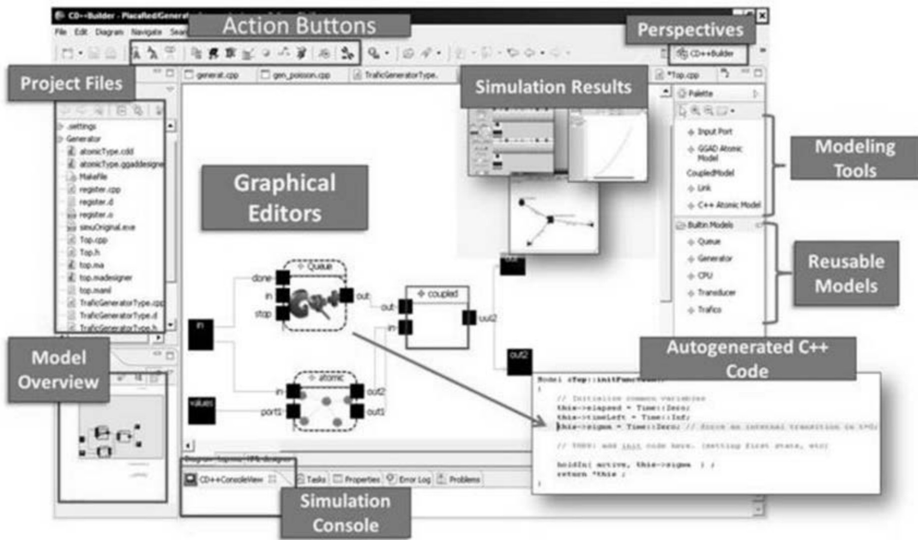

Fig. 1. CD++ infrastructure on the Eclipse Platform (source: `http://cell-devs.sce.carleton.ca/mediawiki/index.php/` Screenshots)

2 Our System

This study describes a small part of a system from an ongoing project with the Arbora-Ausonia enterprise. The main elements of this sub-system include a conveyor and robot.

Fig. 2. An example of a roller conveyor similar to that modeled in the project (Source Wikipedia)

For confidentiality, we only show the behavior of one of the more common elements in industry, a roller conveyor belt (see Fig. 2). However, for our purposes this is enough to explain the system. Although it is a common element, the belt is

complex enough to justify the use of the extensions to SDL-2000. Additionally, a conveyor includes the complex behavior of boxes that continuously travel the length of the unit and requires minimization of the number of events (or signals).

2.1 Used SDL-2000 Extensions

We define models using the Specification and Description Language, but to fully define a simulation model, as was performed with certain tools presented, it is not enough to only allow for the definition of the model structure (that is defined for the different elements in the model) and behavior (that is defined for each element). A complete definition of the behavior (including the time) and graphical representation of the simulation model are required.

2.2 Time and Priority Management

For a discrete simulator, a complete definition of the behavior of a model is needed to describe the time related to the execution of each event that manages its evolution. Usually each type of event has a specific probability distribution, which determines when the event is executed. For an event scheduling simulator, the engine manages the time for all the events and decides where and when these events must be sent (to other simulation elements, which are the agents in a Specification and Description Language model).

SDL-2000 has two main structures for time management, *Timers* and *Delaying Channels* [1].

Delaying Channels of SDL-2000 were not acceptable for representing the delays in a simulation model because in SDL-2000 there were no existing mechanisms to define the required time to reach the destination with these channels. The *Delaying Chanel* represents a delay in the transmission of the signal, but the probability distribution of this delay cannot be defined. The other mechanism, *Timers*, is inadequate, because each different instance of a signal that can travel in parallel requires the definition of a new *Timer*. For example, if we need to send a signal to represent the arrival of new entities to a machine, when a new arrival is sent to this machine, the *Timer* is reprogrammed; thus, the signal has not arrived to its final destination and is reprogrammed. Only one instance of the signal represented by the *Timer* can travel through the system. Additionally, *Timers* cannot represent the priorities. This represents a strong limitation in order to perform a more readable representation of a dynamic system where the delays and priorities must be completely defined.

Specification and Description Language time management has been studied by several groups [28,29]. Specifically, [29] presented an extension that defines three kinds of transitions; (i) eager (consumed without delay), (ii) lazy (not urgent) and (iii) delayable (an enabling condition depending on time). For a discrete simulator, all the transitions can be considered delayable because all the transitions have a defined time. An eager transition is equivalent to a delayable transition with the temporal condition set to now=x [29].

Considering these issues, we implemented extensions of SDL-2000 in SDLPS. In SDLPS, all the signals carry the parameters with the following elements: (i) *ExecutionTime* or *delay*, the time when the event must be executed. (ii) *Priority*, the priority of the event, which is used to eliminate simultaneity of events. (iii) *CreationTime*, the time when the event is created. (iv) *Id*, an identifier of the event. (v) *Time*, the clock reading for the process (represents the time related to the last event that was processed by the process). (vi) *Destination*, the final destination (process PId) of the signal.

The parameter *event* delays or sorts (by priority) the different signals. When a *signal* is received, SDLPS uses the *event* parameter to manage the time and the priorities of the signal. In SDLPS, we can use extension elements to define this parameter related to the signal, as shown in Fig. 3. Not all the parameters of the *event* structure have to be defined: only those required to fully define the behavior of the model.

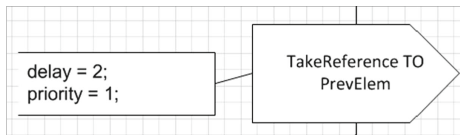

Fig. 3. A *delayable* signal. This *signal* requires 2 time units to reach its destination. Additionally, priorities can be defined to avoid ambiguity when two signals reach the destination at the same time.

Summarizing this section, to manage time we add to the language:

1. All signals can have a time delay: each signal instance output has a parameter that defines the time needed to travel to its final destination (i.e., a *delay* or the value of the *ExecutionTime* value minus the current time) and an input queue schedule parameter defining the priorities with respect to the other signal instances that arrive at the destination at the same time (i.e., the *ExecutionTime* value). A signal instance is therefore only available in the destination input port when the current time is greater or equal than the *ExecutionTime*.
2. The signals in the input port are scanned in the following order to determine if there is an enabled signal: *ExecutionTime* and priority. Signal *priority* determines the signal that is processed first. If two signals with the same *ExecutionTime* and *priority* exist, the implementation decides which signal is executed first (the model does not specify this scenario).

These proposed extensions are included in the SDL-2010 release of the standard [1].

2.3 To Represent the Model

To represent the model, we assume that all SDL-2010 *agents* (*system*, *block* and *process*) can be represented. Thus, the *agent* that represents the conveyor has a real position on the model graphical representation (or layout) and also has a file that describes its shape. This representation suggests that the definition of an *agent* can also have information related to its representation (information regarding the visual behavior of the *agents* in a 2D or 3D environment). Our current implementation provides this representation in an XML file that describes the initial position of the *agents* and a file that describes its shape (see Fig. 4).

```
<Agent name='BCinta2_PCinta2'>
        <state name='ROLLING'>
                <mesh scale='1'>default.obj</mesh>
                <pos x='0' y='0' z='0' />
                <rot x='0' y='0' z='0' />
        </state>
</Agent>
```

Fig. 4. XML representation of an agent with the extensions used to represent the agent in a virtual reality environment. For agent *BCinta2_PCinta2* in the state of *ROLLING, default.obj* represents the agent and the initial position and orientation is (0,0,0).

In this case in the shape for the agent BCinta2_PCinta2 (a process) is in a file (default.obj) and its initial position is 0,0,0 with no rotation.

To represent the model with a 3D (or 2D) animation we define a library that can be accessed during the execution time using a *Procedure Call*. In Fig. 5 there is an example of this *Procedure Call* called *AnimTo*.

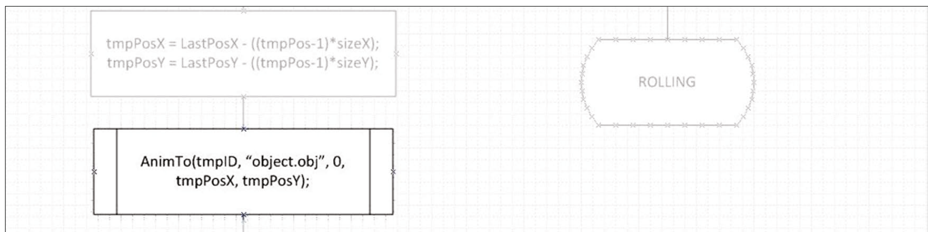

Fig. 5. The *AnimTo* procedure allows for defining the animations to represent the model in a 3D (or 2D) environment

When the simulator executes the procedure it creates a representation in the 3D environment. The parameters of this call are described as follows:

AnimTo(ID, meshPath, delay, x, y, z)

In this case, the element is identified by the *ID* identity and moves to (x, y, z) coordinates at the current *execution time* plus the time *delay*. The shape is defined by a mesh in a WaveFront OBJ format that is stored in the *meshPath* path parameter.

A sequential execution of this procedure creates an animation between the specified points. For the conveyor case, this procedure is used to represent the elements that the conveyor transports. Thus, to create an animation from the beginning to the end of the conveyor we must execute:

AnimTo(ID, meshPath, 0, x0, y0, z0)
AnimTo(ID, meshPath, delay, x, y, z)

Where (x0, y0, z0) is the initial position, (x, y, z) is the last position and *delay* is the time to move the distance of the conveyor.

2.4 Conveyor Model

It is not our intention to perform a complete description of the model. Instead, we detail the behavior of the most complex element (with representation), the conveyor. The conveyor has different parameters including the speed of the boxes, defined by the rotation speed of the rollers. Additionally, we can define the number of boxes that can be carried (this parameter depends on the size of each box). All the parameters that can be configured by the user are represented on the SDLPS diagrams using DCL statements. The behavior of the conveyor is described as follows.

Elements can be added to the beginning of the conveyor. The conveyor continuously moves the elements to the end of its structure. Because this conveyor is composed of different cylinders that move the elements (not by a continuous belt), when the first elements reaches the end of the conveyor (but cannot leave because the next element is blocked), the other elements can continue moving. This relationship implies that the conveyor behaves as a buffer that can store elements until *MaxElems* boxes is reached (defined in the declarations).

Because we are planning to use SDLPS to interpret and generate the model, the notation of the declarations and the code for the diagrams is written in ANSI C language to simplify the DLL needed to perform the execution of the system. This coding implies that the language we are using is similar to SDL-RT [26] or C-language binding as in SDL-2010 [1]. We define following terminology to describe the diagrams.

Entity: the entities are the elements that move through the system using different facilities to define different processes. Each entity “travels” through different agents that perform operations on them.

Process: despite having an agent titled process in SDL-2010, a theoretical process in operations research represents the set of operations that must be performed to the entities. The definition of these operations is based on the SDL-2010 process agents.

Event: An event occurs in the simulation model and implies the modification of select state variables of the model. In our approach, the events are represented using SDL-2010 *signals*.

With these considerations for the model definition, there are two main events (signals), *NewReference* (used to indicate that a new element reaches and attempts to enter another element) and *TakeReference* (the other element, i.e., agent, attempts to take one of the elements that was completed in the process). From these two signals, we can define a model that is similar to a PUSH/PULL paradigm for a process interaction engine and similar to the simulation tools that were described previously.

For the conveyor, we only consider the ROLLING state. In this state, we can receive events, including *NewReference*, *Full*, *Reroll*, *TakeReference*, *Unblock* and *Roll*. The behavior of these elements is described in Fig. 6, Fig. 7 and Fig. 8.

Figure 6 shows the declarations that contain the elements of the conveyor that can be modified to define different scenarios. For example, the variable *double MaxElems=10* can be changed to test the difference between using a short (*MaxElems=5*) or a long (*MaxElems=20*) conveyor.

3 The Implementation and Execution of the Model

The model was implemented with SDLPS software, developed in the InLab FIB of the Polytechnic University of Catalonia [23,27]. This tool uses SDL-RT (the code for tasks is defined using C language) and the the extensions to SDL-2000. Regarding the infrastructure, SDLPS was built with C++ and C languages. The model code (written in C for the tasks and procedures of the SDL-2000 blocks) are used through a DLL. The SDL-XML model is generated with a plug-in on Microsoft Visio®. This coding implies that the model can be mainly validated and verified by reviewing the graphic diagrams on Microsoft Visio®. This property dramatically simplifies the interaction between the different parts involved in the project.

SDL-2010 does not define the ordering of events when two events with the same *ExecutionTime* have the same priority. This situation must be defined in the model or by the simulation engine. SDLPS eliminates this ambiguity by storing these events in a FIFO queue.

From Fig. 9, the simulator shows the model diagrams that will be simulated. The system uses an XML representation of the model that is obtained from Microsoft Visio® with the SanDriLa© plug-in.

Because we use this infrastructure, no specific implementation was performed for this project, simplifying the verification process required for simulation projects [30].

Figure 10 shows the steps that are simplified (red in the online paper/ grey in the printed paper) by this methodology that are needed for a simulation.

The obtained results from the model emulation trace can be presented in Microsoft Excel® or SDLPSEye, which is capable of representing information in a 3D environment (for the representation events described in the previous section). All agents can have unique representations, and due to the time extensions, we can determine the movement of the simulation entities.

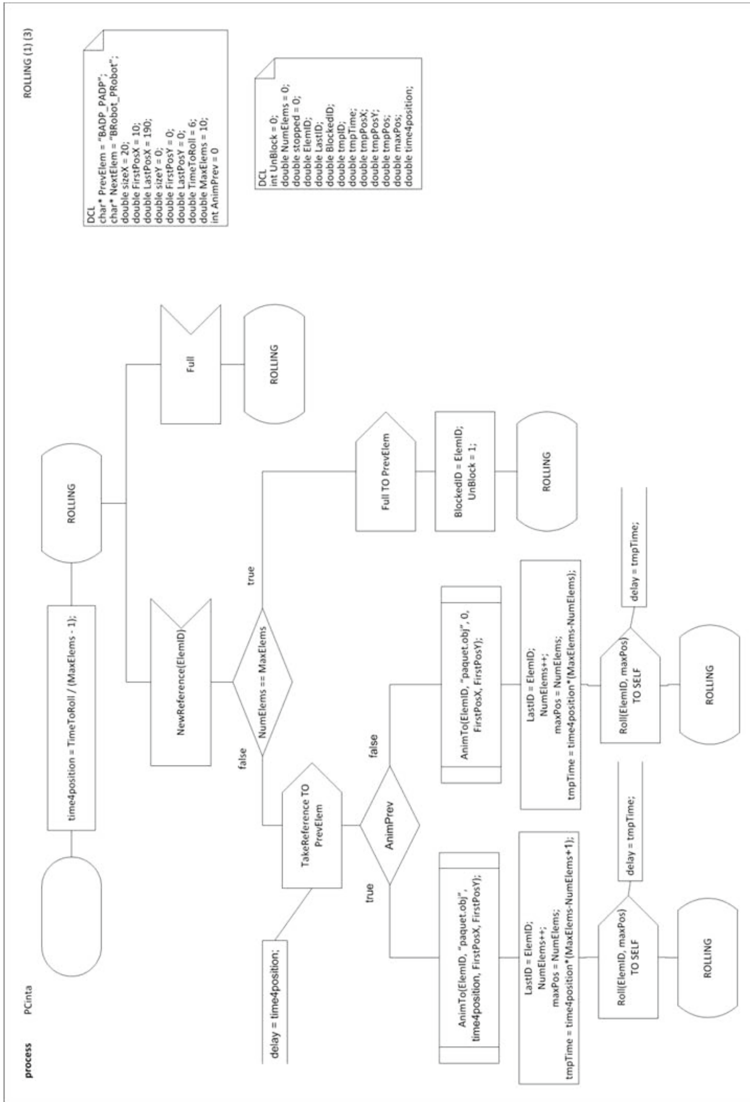

Fig. 6. PCinta for the ROLLING state (1/3). This figure shows how the process is instantiated. The time4position variable is defined from the START symbol and is the time that is required for each of the pieces to reach a specific position on the conveyor. From the "rolling" state, we can receive a *NewReference* signal. We then analyze the number of elements that are in the conveyor. If this number is equal to the conveyor capacity, the conveyor sends the "Full" signal to the previous agent. Otherwise, the conveyor processes the element (sends the *TakeReference* signal to the previous agent). Then, the program determines if the animation must be completed from the beginning or if this element is connected to another conveyor *AnimPrev* value. In this example, the *AnimPrev* value is 0, such that the previous element is not a conveyor.

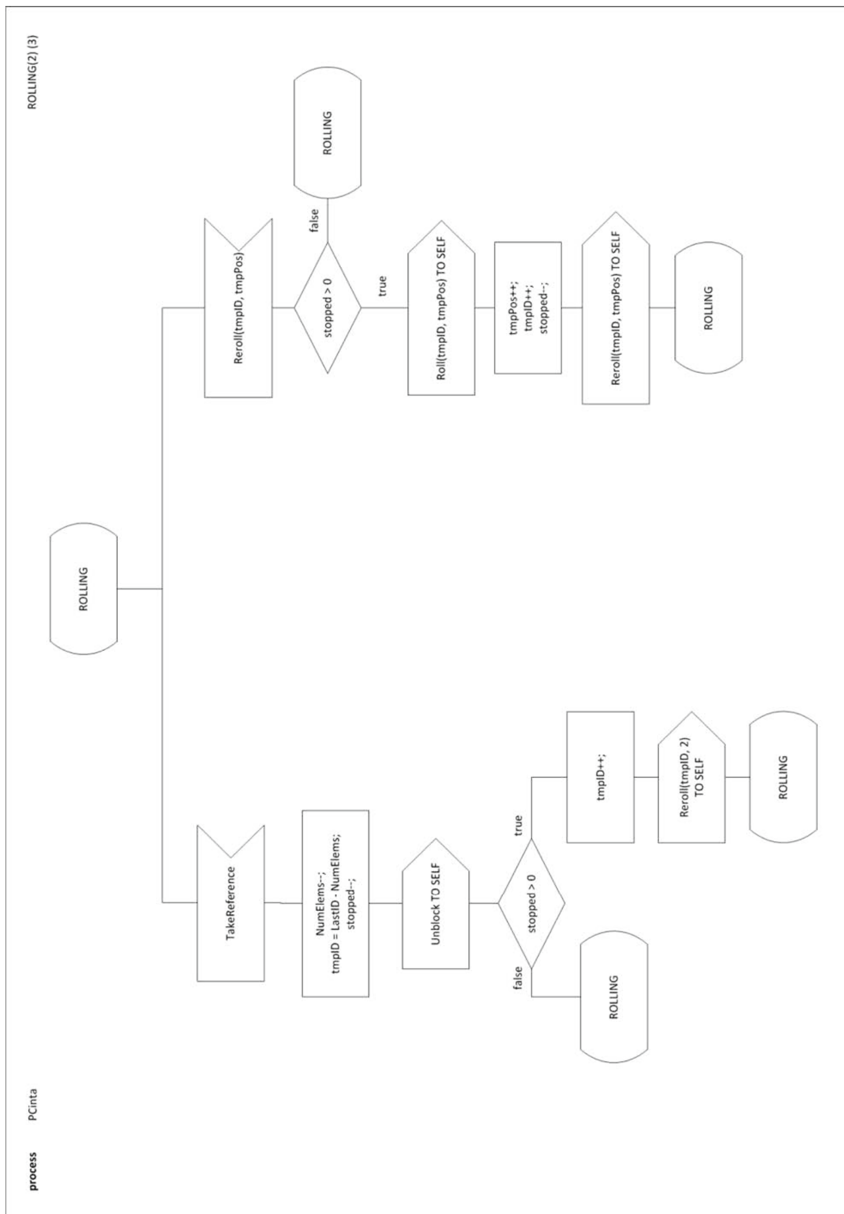

Fig. 7. PCinta for the ROLLING state (2/3). When a conveyor receives a *TakeReference* signal, the *NewReference* is accepted by the next agent. Thus, after receiving this signal, we can decrease the number of elements of our agent and unblock the previous agent (in case that agent was blocked by our agent). We also can observe the behavior of the conveyor when it receives a *Reroll* signal. This signal is sent by the agent (self-signal) and is used to start the motion of the conveyor after a blocking occurs.

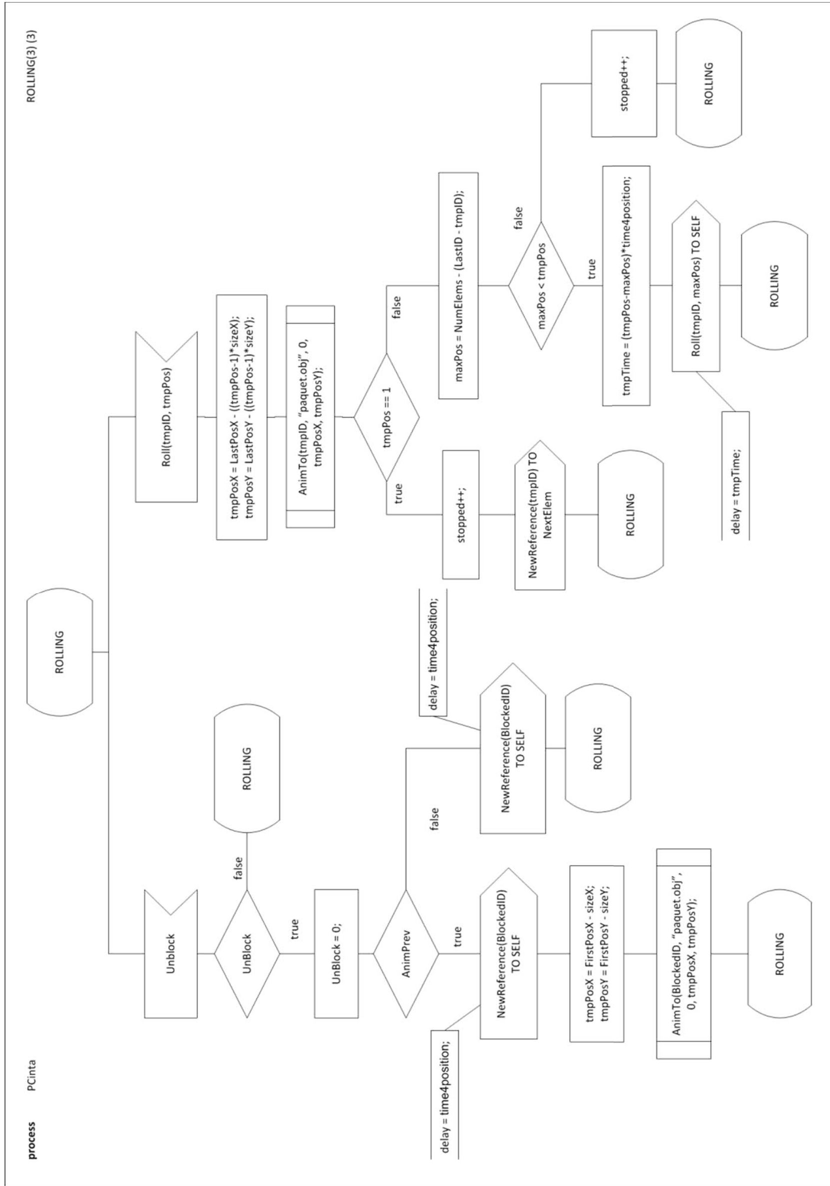

Fig. 8. PCinta for the ROLLING state (3/3). A conveyor receives an *Unblock* signal when we need to unblock the previous agent. To unblock the previous agent, we simulate receiving a *NewReference* signal. We also observe the formalization of the *Roll* signal. This signal is used to simulate the traveling time of the elements along the conveyor. When an element attempts to travel from the current position to position number 5, we send a *Roll* signal to the self conveyor with a delay, which represents the time for traveling from one position to the next.

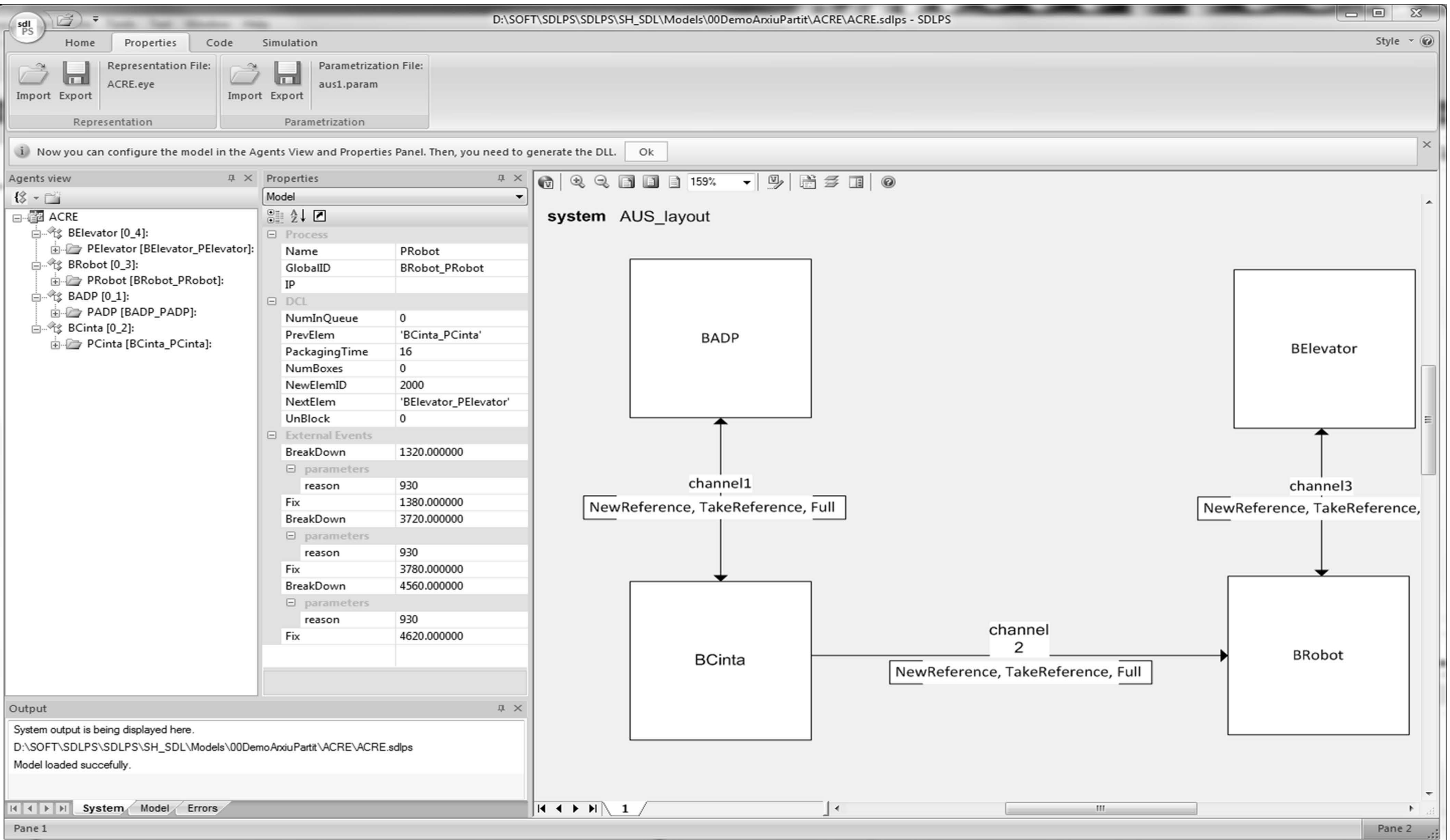

Fig. 9. SDLPS interface with the model

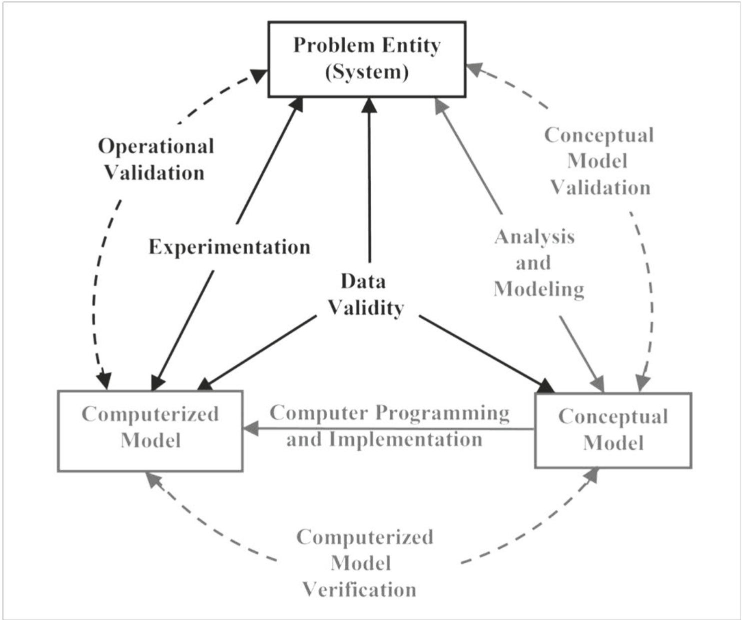

Fig. 10. Simplified version of the modeling process [30]

3.1 The Model Eyes: SDLPSEye

The representation of the model is stored in a trace file. This file contains the representation events from all the simulations. The events can be generated by the *AnimTo* procedure or by changing any process of the model. For the latter, the event must be completed with a representation of every possible state of the given process. Figure 11 is an example with a defined state representation of every agent. In this case, the agent *BRobot_PRobot* has 3 states (IDLE, BLOCKED and WORKING). Figure 12 is an example of a representation of an event sequence. In this case, there are two representation events types, which include *EYE_SetState* and *EYE_AnimTo.*

Figure 13 and Fig. 14 represent the model that was obtained from the description of the SDL diagrams. The boxes are the *mesh* elements (the representation of the agent) defined on the extensions.

4 Concluding Remarks

Despite that the obtained results from the simulation model can be used for a decision process in industry, the Specification and Description Language becomes an excellent language to fully describe the behavior of the enterprise elements, due the added time and representation capabilities. As we see in the introduction

```
<ModelInfo>
    <ModelName>Ausonia</ModelName>
    <Agents>
        <Agent name='BRobot_PRobot'>
            <state name='IDLE'>
                <mesh scale='1'>box_green.obj</mesh>
                <pos x='420' y='0' z='0'/>
                <rot x='0' y='0' z='0'/>
            </state>
            <state name='BLOCKED'>
                <mesh scale='1'>box_red.obj</mesh>
                <pos x='420' y='0' z='0'/>
                <rot x='0' y='0' z='0'/>
            </state>
            <state name='WORKING'>
                <mesh scale='1'>box_yellow.obj</mesh>
                <pos x='420' y='0' z='0'/>
                <rot x='0' y='0' z='0'/>
            </state>
        </Agent>
        <Agent name='BElevator_PElevator'>
            <state name='RUNNING'>
```


Fig. 11. XML file defining the representation of the model


```
<Events>
    <EYE_SetState xTime='0.000000' agent="BRobot_PRobot" state="IDLE"></EYE_SetState>
    <EYE_SetState xTime='0.000000' agent="BElevator_PElevator" state="RUNNING"></EYE_SetState>
    <EYE_SetState xTime='0.000000' agent="BADP_PADP" state="RUNNING"></EYE_SetState>
    <EYE_SetState xTime='0.000000' agent="BCinta_PCinta" state="ROLLING"></EYE_SetState>
    <EYE_AnimTo xTime='0.000000' agent='1000.000000'>
        <mesh scale='1'>paquet.obj</mesh>
        <pos x='10.000000' y='0.000000' z='0'/>
    </EYE_AnimTo>
    <EYE_AnimTo xTime='0.555556' agent='1001.000000'>
        <mesh scale='1'>paquet.obj</mesh>
        <pos x='10.000000' y='0.000000' z='0'/>
    </EYE_AnimTo>
    <EYE_AnimTo xTime='2.111112' agent='1002.000000'>
        <mesh scale='1'>paquet.obj</mesh>
        <pos x='10.000000' y='0.000000' z='0'/>
    </EYE_AnimTo>
    <EYE_AnimTo xTime='4.777780' agent='1003.000000'>
        <mesh scale='1'>paquet.obj</mesh>
        <pos x='10.000000' y='0.000000' z='0'/>
    </EYE_AnimTo>
    <EYE_AnimTo xTime='5.000004' agent='1001.000000'>
        <mesh scale='1'>paquet.obj</mesh>
        <pos x='170.000000' y='0.000000' z='0'/>
    </EYE_AnimTo>
    <EYE_SetState xTime='5.000004' agent="BRobot_PRobot" state="BUSY"></EYE_SetState>
    <EYE_AnimTo xTime='5.000004' agent='1000.000000'>
        <mesh scale='1'>paquet.obj</mesh>
        <pos x='190.000000' y='0.000000' z='0'/>
    </EYE_AnimTo>
    <EYE_AnimTo xTime='5.666671' agent='1000.000000'>
```


Fig. 12. A trace of the representation, representing the complete behavior of the model

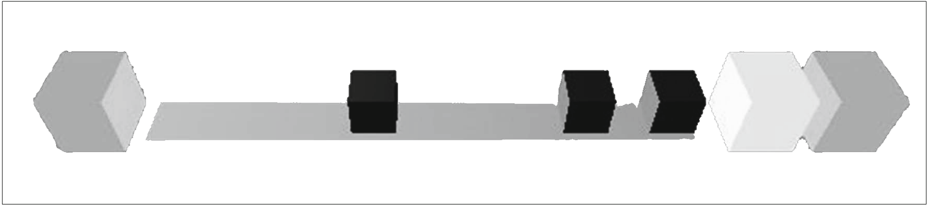

Fig. 13. The conveyor contains three boxes. The two boxes at the end are waiting for service.

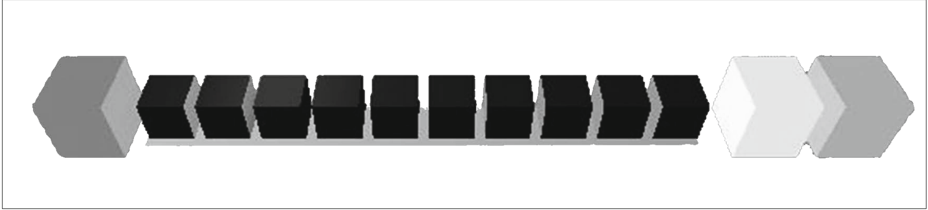

Fig. 14. The conveyor is full and blocking the box generator (in red). In this example, 4 blocks are represented. From left to right, the first, third and fourth elements have the simplest representation, consisting of a color representation of each state. For the conveyor block, a more accurate representation is presented using the *AnimTo* procedure.

there are certain formal languages, programming libraries and infrastructures that allow defining a simulation model. However, any of these languages allows the definition of the model representation. To achieve this it is needed to use proprietary infrastructures and tools. This obviously is far from the objective to achieve a complete and formal representation of a simulation model independent of the tool or infrastructure used to finally perform the implementation.

Thus, once the system is fully described, a client can modify the structure of the model using only Microsoft Visio® SDL-2010 diagrams with the SanDriLa [31] plug-in. Additionally, because the more important parameters of the model are defined in the declarations and can be modified directly in the SDLPS infrastructure, simple modifications (new parameterizations) of the model are not time intensive. Thus, in industry, the managers can validate the accuracy of certain proposed alternatives using common tools such as Microsoft Visio® .

Additionally, this tool can validate the accuracy of several of the proposed solutions. Thus, the diagrams that represent the tacit and explicit knowledge of the industry can be validated, allowing for the representation and validation of these types of knowledge.

We are currently continuing with the project implementation in industry and installing the system for the clients so they can modify and define their own models. The main elements of the system can be predefined with SDL-2010 blocks that implement a library, so that several elements can be reused.

As shown in this study, specification and description language can be used in operations research to fully represent discrete simulation models with temporal extensions and a library to represent the basic operations to render a 3D environment.

Because the validation of the model is performed in the SDL-2010 representation of the model, non-simulation specialists that are experts in system behavior can understand the model's behavior. Thus, all the actors involved in the project can participate in the model validation. Thus, SDL-2010 diagrams can be used to represent the tacit knowledge that expresses the behavior of the complex interactions between several actors in industry.

References

1. International Telecommunication Union: Recommendation Z.100 (12/11) Specification and Description Language – Overview of SDL-2010, http://www.itu.int/rec/T-REC-Z.100
2. Earle, N., Henriksen, J.: Proof Animation – Better Animation For Your Simulation. In: Proceedings of the 25th Conference on Winter Simulation (WSC 1993), pp. 172–178. ACM (1993), http://doi.acm.org/10.1145/256563.256617
3. Brade, D.: Enhancing modeling and simulation accreditation by structuring verification and validation results. In: Proceedings of the 32nd Conference on Winter Simulation (WSC 2000), pp. 840–848. Society for Computer Simulation International (2000)
4. Gordon, G.: The Development of the General Purpose Simulation System (GPSS). ACM SIGPLAN Notices 13(80), 183–198 (1978), http://portal.acm.org/citation.cfm?doid=960118.808382
5. Fonseca i Casas, P., Casanovas, J.: JGPSS, an Open Source GPSS Framework to Teach Simulation. In: Winter Simulation Conference (WSC 2009), pp. 256–267. Winter Simulation Conference (2009)
6. Pritsker, A.: Introduction to simulation and SLAM II. Halsted Press (1986)
7. CACI: Simprocess, http://simprocess.com/
8. Rockwell Automation: Arena Simulation Software, http://www.arenasimulation.com/
9. Simio LLC: Simio forward thinking, http://www.simio.com/index.html
10. Simul8 Corporation: Simul8, http://www.simul8.com/
11. Guasch, A., Piera, M., Casanovas, J., Figueras, J.: Modelado y simulación. Edicions UPC (2002)
12. Law, A., Kelton, W.: Simulation Modeling and Analysis. McGraw-Hill (2000)
13. Lanner: Witness, http://www.lanner.com/en/witness.cfm
14. Zeigler, B.P., Kim, D., Praehofer, H.: DEVS formalism as a framework for advanced distributed simulation. In: Proceedings of the 1st International Workshop on Distributed Interactive Simulation and Real-Time Applications (DIS-RT 1997), pp. 15–21. IEEE Computer Society (1997)
15. Peterson, J.: Petri Net Theory and the Modeling of Systems. Prentice-Hall (1981)
16. Petri, C.: Kommunikation mit Automaten. University of Bonn, Bonn (1962)
17. Liu, R., Kumar, A., van der Aalst, W.: A formal modeling approach for supply chain event management. Decision Support Systems 43(3), 761–778 (2007)
18. Wainer, G.: DEVS tools, http://www.sce.carleton.ca/faculty/wainer/standard/tools.html

19. University of Hamburg: Petri Nets Tool Database, http://www.informatik.uni-hamburg.de/TGI/PetriNets/tools/complete_db.html
20. Wainer, G.: CD++: a toolkit to develop DEVS models. Software, Practice and Experience 32(3), 1261–1306 (2002)
21. Zeigler, B.P., Sarjoughian, H.S.: Creating Distributed Simulation Using Devs M&S Environments. In: Proceedings of the 32nd conference on Winter simulation (WSC 2000), pp. 158–160. Society for Computer Simulation International (2000)
22. Risco-Martín, J.L., Mittal, S., López-Peña, M.A., De la Cruz, J.M.: A W3C XML Schema for DEVS Scenarios. In: Proceedings of the 2007 Spring Simulation Multiconference (SpringSim 2007), vol. 2, pp. 279–286. Society for Computer Simulation International (2007)
23. Fonseca i Casas, P., Casanovas, J.: Towards a SDL-DEVS Simulator. In: The Third International Conference on Advances in System Simulation (SIMUL 2011), pp.188–194. International Academy, Research and Industry Association (2011)
24. Cinderella ApS: Cinderella SDL 1.3, http://www.cinderella.dk
25. IBM: Rational SDL Suite, http://www-947.ibm.com/support/entry/portal/overview/software/rational/rational_sdl_suite
26. Specification & Description Language - Real-Time (2006), http://www.sdl-rt.org/
27. Fonseca i Casas, P.: SDL distributed simulator (poster) In: Proceedings of the 40th Conference on Winter Simulation (WSC 2008). Winter Simulation Conference (2008), http://www-eio.upc.edu/~pau/?q=node/67
28. Bozga, M., Graf, S., Mounier, L., Kerbrat, A., Ober, I., Vincent, D.: SDL for Real-Time: What Is Missing? Interval project publication, http://www-interval.imag.fr/Pub/sam2k.ps
29. Bozga, M., Graf, S., Mounier, L., Ober, I., Roux, J.-L., Vincent, D.: Timed Extensions for SDL. In: Reed, R., Reed, J. (eds.) SDL 2001. LNCS, vol. 2078, pp. 223–240. Springer, Heidelberg (2001)
30. Sargent, R.G.: Verification and validation of simulation models. In: Proceedings of the 39th Conference on Winter Simulation (WSC 2007), pp.124–137. Winter Simulation Conference (2008)
31. SanDriLa Ltd.: www.sandrila.co.uk/visio-sdl/