Simulation

*Simulation: Transactions of the Society for Modeling and Simulation International*
2015, Vol. 91(3) 249–264
© 2015 The Author(s)
DOI: 10.1177/0037549715571623
sim.sagepub.com
SAGE SCS

# Transforming classic Discrete Event System Specification models to Specification and Description Language

**Pau Fonseca i Casas**

## Abstract

Discrete Event System Specification (DEVS) is one of the main widely used formal languages to represent simulation models, while Specification and Description Language (SDL) is a graphical ITU-T standard language, commonly used in telecommunication and engineering areas. In this paper, we present an algorithm, and a simulation infrastructure that implements this algorithm, to transform a simulation model represented using the DEVS formalism to the SDL standard language. The algorithm can be viewed as a mechanism to represent graphically DEVS models. In addition, because of the transformation, one can use SDL tools in order to implement DEVS models. To implement the algorithm, we propose an Extensible Markup Language representation for the DEVS and SDL models. For practical application of the algorithm, it is implemented in a simulation infrastructure named the Specification and Description Language Parallel Simulator that allows defining the models with both formalisms.

## Keywords

Discrete Event System Specification, Specification and Description Language, simulation, formal language

## 1. Introduction

The growing complexity of simulation models requires a formal system that aids in model specification, particularly when the individuals building the model come from different areas of expertise. The main mechanisms for working with models that are specified using different formalisms[1] are (i) meta-formalism, (ii) common formalism and (iii) co-simulation.

Several alternatives to represent Discrete Event System Specification (DEVS)[2] graphically exist, such as Traoré[3] and Ighoroje et al.,[4] who present a graphical notation for DEVS named DEVS-driven Modeling Language (DDML). A similar approach is presented by Song and Kim[5] with the DEVS diagram. Kidisyuk and Wainer[6] and Bonaventura et al.[7] present CD++ Modeler, an application that permits defining DEVS models graphically based on an Eclipse platform. In Wainer and Liu[8] several other platforms based on CD++ are introduced. Other infrastructures, such as CoSMoS,[9] allow the definition of cellular automaton structures following DEVS notation in a graphical manner.[10] Villalba et al.[11] present an implementation based on Modelica for DEVS-GRAPH.[12] The problem of writing models based on DEVS was discussed in the framework of the DEVS standardization group (see Wainer et al.,[13] Ighoroje et al.,[14] Sarjoughian and Chen[15] and Mittal and Martín[16]). None of these alternatives uses a standard graphical language, such as Specification and Description Language (SDL),[17] to define the models. This makes a transformation between both formalisms interesting, but if a graphical and standard representation for DEVS models appears, this transformation is also interesting due to the capability to combine both formalisms in a single model. In addition, we understand that the graphical representation of DEVS models using SDL provides some interesting features to DEVS. Firstly, a new and complete graphical representation of the models, through SDL/GR (Specification and Description Language Graphical Representation), the graphical representation of SDL models. Secondly, a textual description of a system can be obtained, through SDL/PR (Specification and Description Language Phrase Representation), the textual representation of SDL models. Thirdly, because SDL is a language

Universitat Politècnica de Catalunya, Spain

**Corresponding author:**
Pau Fonseca i Casas, Universitat Politècnica de Catalunya, C\Jordi Girona, 1-3, Ed. C5 Campus Nord, Barcelona, 08034, Spain.
Email: pau@fib.upc.edu

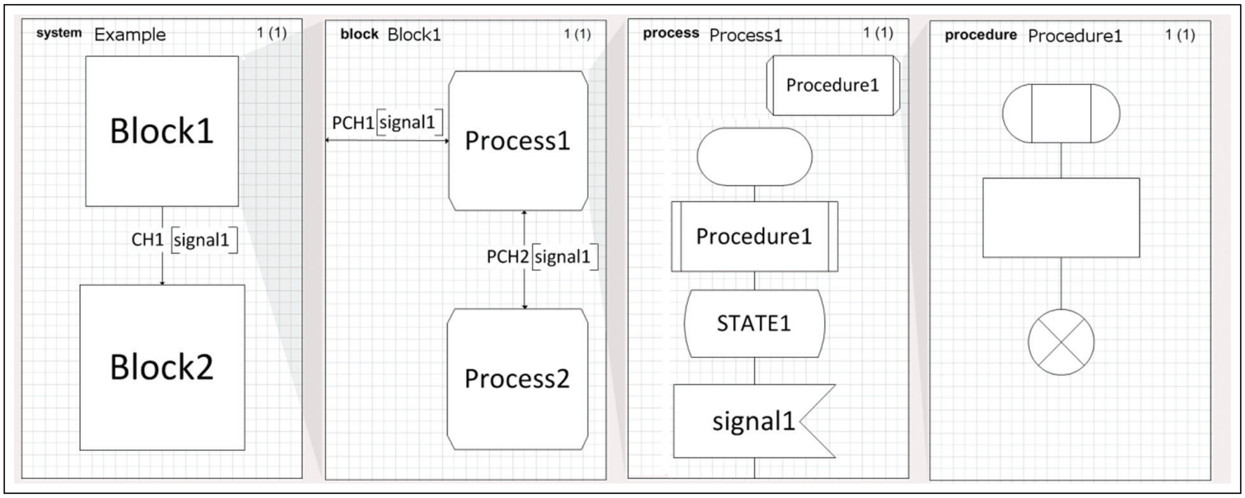


**Figure 1.** Specification and Description Language structure.

of the ITU-T (International Telecommunication Union – Telecommunication Standardization Sector), this transformation provides a standard representation for DEVS models with a complete, well-known and precise grammar; also, it is understood by numerous tools.[18–21] In addition, SDL can be combined with Unified Modeling Language (UML), allowing the formalization of an entire DSS (Decision Support System; see recommendation Z.109[22]). Finally, the DEVS models can benefit from the validation SDL tools that currently exist in combination with TTCN (Testing and Test Control Notation).[23] Regarding this, it is also interesting to note that some efforts have been done to combine DEVS with UML[24] and SysML[25] in order to validate the proposed system.

The objective of this paper is to present an algorithm to transform DEVS formalization to SDL. This transformation simplifies the work of a multidisciplinary team, since we can use SDL as a graphical representation for atomic DEVS models. In addition, we present an infrastructure that implements the algorithm. It is outside the scope of this paper, but worthy of mention that this transformation can be done thanks to the new features we propose to the language that have been added to the new version of the language, SDL-2010 approved in 2012, specifically ‘‘In SDL-2010, it is possible to specify the delay between output of signal and the signal being available for consumption in the destination input port.’’[17]

This paper is structured as follows. Sections 2 and 3 describe the SDL and DEVS formalisms. If the reader is familiar with these languages, we recommend reviewing only Section 2.2 to understand the SDL version used in this paper. Section 4 proposes XML representations for DEVS and SDL, which are required for implementation. Section 5 presents an illustrative example of the algorithm that transforms DEVS to the SDL-2010 formalism. Sections 6 and 7 present the algorithm and its implementation on the SDLPS (Specification and Description Language Parallel Simulator) infrastructure. Finally, concluding remarks are given in Section 8.

## 2. Specification and Description Language

SDL is an object-oriented, formal language that was defined by the ITU-T (formerly Comité Consultatif International Télégraphique et Téléphonique [CCITT]) as Recommendation Z.100.[17] The language was designed to specify complex, event-driven, real-time, interactive applications that involve many concurrent activities that communicate using discrete signals.[26,27]

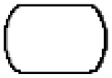
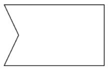
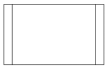
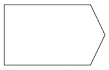
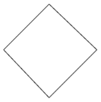
The definition of the model is based on the following components:

- **structure**: system, blocks, processes, procedures and the hierarchy of processes;
- **communication**: signals, including the parameters and channels that the signals use to travel;
- **behavior**: defined through the processes;
- **data**: based on Abstract Data Types (ADTs);
- **inheritances**: describe the relationships between and the specializations of the model elements.

The language has four levels: (i) system; (ii) blocks; (iii) processes; and (iv) procedures. The hierarchical decomposition of SDL is shown in Figure 1.

A SYSTEM diagram represents 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. An AGENT, in SDL

**Table 1.** Some Specification and Description Language process elements.

| | |
|---|---|
|  | **State.** A state element contains the name of a state. All diagrams start and end with state elements. |
|  | **Event reception.** These elements describe the type of events that can be received depending on the state and the number of ports through which the event traveled. Because an object changes its state only after a new event is received, all branches of a specific state start with an event-reception element. The symmetric representation of this element is also allowed. |
|  | **Procedure.** These elements perform actions, encapsulating some part of the model behavior. |
|  | **Send event.** These elements describe the type of event to be sent and the port to be used. Other attributes of the event can also be detailed, e.g., priority, execution time, etc. The symmetric representation of this element is also allowed. |
|  | **Decision point**. These elements describe bifurcations. Their behavior depends on how the related question is answered. |

terminology, can be the SYSTEM, a BLOCK or a PROCESS.

Each rectangle represents an AGENT. The lines that join the objects are the communication channels. Both bidirectional and unidirectional channels are allowed in SDL. The communication channels are joined to the objects through ports. Because ports ensure the independence of objects, they are very important elements for implementing and reusing 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.

The BLOCK diagrams contain a number of PROCESS agents and might contain other BLOCKs. Processes communicate via signal routes, which connect to other processes or to channels external to the BLOCK.

### 2.1 SDL behavior representation

In SDL, the behavior of each state of a PROCESS to each of the SIGNALS that reach the AGENT is described. A PROCESS might react differently to a SIGNAL depending on the port through which it was sent, according to the last version of the standard. The PROCESS is specified using graphical elements that describe operations or decisions. Table 1 describes some of the most important elements.

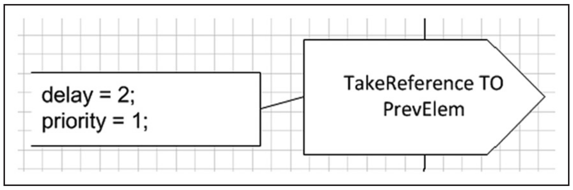


**Figure 2.** SDL-2010 delayable SIGNAL. Note that the SIGNAL requires two units of time to reach its destination. A priority is defined to break the ambiguity that exists when two signals reach the destination at the same time.

The last level of SDL is the description of the different procedures that appear in the SDL diagrams. The structure and elements of a procedure diagram are similar to process diagrams. For more information regarding SDL, the reader is referred to the Telecommunication standardization sector of ITU,[28] SDL Tutorial[29] and Reed.[26]

### 2.2 Working with time in SDL

The scheduled execution time of each model event must be defined in a discrete simulation model. Often, each type of event has a probability distribution that determines when the type of event must be executed. In addition, the priority, in relation to other events scheduled for the same time, must also be defined. Based on our previous work, we proposed two extensions to SDL that simplify the definition of delays and priorities in models. After several years of discussions with the ITU-T committee, the extensions have been accepted and are included in SDL-2010 (published at the end of 2012). In this paper, we assume that we are working with the SDL-2010 standard. This extension simplifies the proposed transformation algorithm we see next.

Figure 2 shows a process diagram that represents the behavior of a server using the extension to define the delay related to the service time. In a simulation model represented using SDL, the events are represented by SIGNAL elements.

As is shown, a *Delay* or *Priority* is added to a SIGNAL (model event) using a text extension symbol.

## 3. Discrete Event System Specification formalism

DEVS was originally designed to formalize simulation and modeling problems.[2,12,30] Proposed in the 1970s by mathematician Bernard Zeigler, DEVS was designed as a general-purpose formalism for representing Discrete Event Systems (DESs). Because it can represent all formalisms, it is one of the most general formalisms for DES analysis.[31] The specification is based on a mathematical description of two elements: a dynamic system and a simulation model (the DEVS model). The structure of a dynamic system defined using DEVS is based on a tuple, *S*, comprising elements defined as follows:

$$
\begin{aligned}
&S = \; < T, X, Y, \Omega, Q, \Delta, \Lambda > \text{, where}\\
&T : \text{time base}\\
&X : \text{set of input values}\\
&Y : \text{set of output values}\\
&\Omega : \text{set of allow able input segments;}\, w < t_1, t_2 >\\
&\quad \rightarrow X, \text{over} T\\
&Q : \text{set of state values}\\
&\Delta : Q \times \Omega \rightarrow Q \text{ global transition function}\\
&\Lambda : Q \times X \rightarrow Y \text{output function}
\end{aligned}
$$

The input segments are the set of valid input values for the model. An input trajectory and an output trajectory are defined in a set of valid input segments.

An initial state value (in the set of state values *Q*) corresponds to the initial state of the input segment.

The event streams are defined by:

- **$\omega < t_0, t_n >$ ➔ $AU\{Ø\}$**, segment over continuous time.
- $\omega$ is an event segment if a finite set of points, $t_1$, $t_2$, …, $t_n \in \; < t_0, t_n >$, exists such that $\omega(t_i) = a_i \in A$ for $I = 1,\ldots,n-1$, and $\omega(t) = Ø$ for each $t \in \; < t_0, t_n >$.

The transition function uses these elements as arguments and calculates the new state value for the end time of the input segment.

Through the output function, the system output reflects the system state. The following is a definition of a model in DEVS:

$$
\begin{aligned}
&M = \; < X, S, Y, \delta_{int}, \delta_{ext}, \lambda, ta > \text{where,}\\
&X : \text{set of input values}\\
&S : \text{set of states}\\
&Y : \text{set of output values}\\
&\delta_{int} : \text{internal transition function;}\, \delta_{int} : S \rightarrow S\\
&\delta_{ext} : \text{external transition function;}\, \delta_{ext} \rightarrow Q \times X \rightarrow S\\
&Q = \{(s,e) | s \in S, 0 \leqslant e \leqslant ta(s)\} : \text{set of total states}\\
&e : \text{time from the last transition}\\
&\lambda : S \rightarrow Y : \text{output function}\\
&ta : \text{time advance function}\\
&ta : S \rightarrow R_0^+
\end{aligned}
$$

An interesting feature of DEVS is the concept of internal and external transitions.

When an internal transition is executed, no external event is processed (external to the model). That is, if the system reaches state *s* at time *t*, the system remains in state *s* for a time defined by *ta*(*s*), during which no external events are received. At time *e*, equal to *ta*(*s*), the system generates an output event, $\lambda(s)$, and changes its state to $s' = \delta_{int}(s)$, defined by the internal transition of state *s*.

An external transition is the processing of an event that comes from outside the model. As in the previous example, the system reaches state *s*. Before the system clock reaches *ta*(*s*) time, an external event with value *x* occurs. The system is in state (*s,e*), where $e \leqslant ta(s)$. In this case, the system changes its state to *s'*, $s' = \delta_{ext}(s,e,x)$, but no output event is generated.

With this information, we classify states based on *ta* function definitions:

- if *ta*(*s*) is 0, *s* is a *transitory* state;
- if $ta(s) = \infty$, *s* is a *passive* state.

More information regarding DEVS formalism can be obtained in Zeigler et al.,[2,12] and Fonseca i Casas.[32]

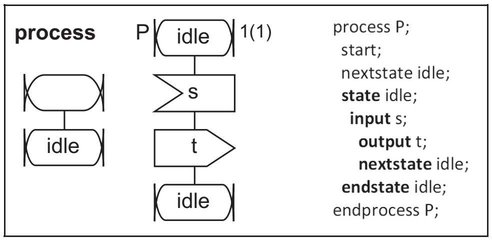
## 4. Extensible Markup Language representation of Specification and Description Language and Discrete Event System Specification

The DEVS and SDL models must be represented in a textual form to code an algorithm that automatically transforms the model from DEVS to SDL. We chose an Extensible Markup Language (XML) representation. There is not a standard representation of DEVS or SDL using XML. DEVS is not yet a standard, and SDL uses SDL/PR as a textual standard. The following details our proposed representations.

### 4.1 XML representation of an SDL simulation model

Figure 3 depicts an example of a non-graphical SDL representation, SDL/PR.[28]

We use SDL/PR as a basis to represent SDL models using XML. Because it is easier to extend and parse an XML file than a textual one, we use an XML representation. From this point forward, this version of SDL is referred to as SDL-XML.



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

**Figure 3.** A non-graphical Specification and Description Language representation. In this example, the initial STATE is IDLE.

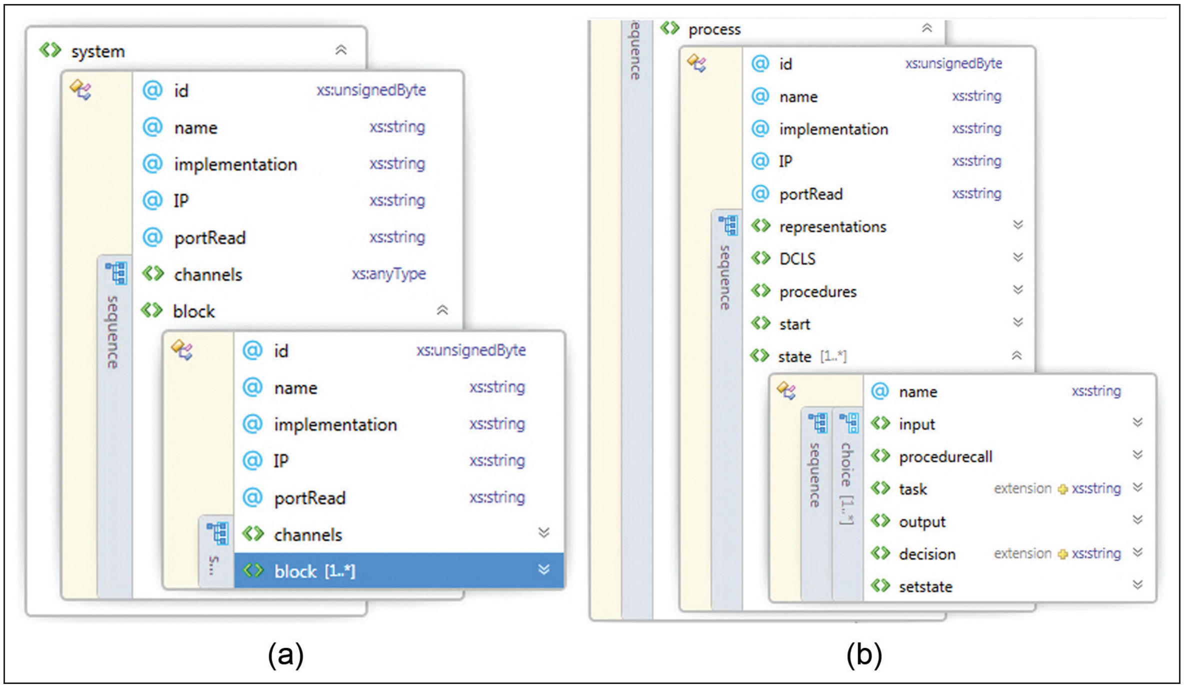


**Figure 4.** XML Schema Documentation. The *system* view in (a) shows all the constitutive elements of a Specification and Description Language (SDL) model, while (b) shows a process *type* that represent a SDL PROCESS.

A preliminary proposal to represent an SDL model was presented by Fonseca i Casas.[33] In Figure 4, the system BLOCK and the process *type* (Figure 4(b)) that represent an SDL PROCESS are shown for the first level of the XML Schema Documentation (XSD) that we use to validate the structure of our XML.

## 4.2 XML representation of DEVS models

Some attempts have been made to represent DEVS models using XML. As an example, Risco-Martín et al.[34] presented schema that cannot represent programming logic, loops or if-then-else constructs. Our XML representation for DEVS models allows for the representation of those elements. Because it is an ISO standard, we propose using ANSI C to represent the code contained in a model. The representation of a model in SDL is simplified by using a variant, SDL-RT, which also uses ANSI C.

We follow conventions to represent a DEVS model using XML syntax. Firstly, all of the code required to define a simulation model is defined in the ‘‘values’’ XML section. Secondly, the initial conditions of the model are defined in the XML using a ‘‘value’’ attribute that is related to all of the variables that define the state of an atomic DEVS model. Finally, to represent $\infty$, which is used in the passive states, we use a literal value, ‘‘inf’’.

Some of the more interesting parts of the XML schema used to represent coupled and atomic models are shown in Figure 5.

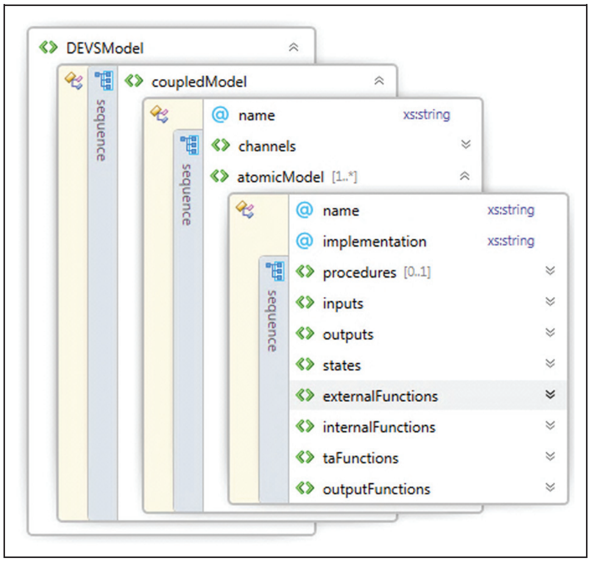


**Figure 5.** Discrete Event System Specification (DEVS) Extensible Markup Language schema. We have different *external*, *internal* and *ta* functions because we can define several transitions in our DEVS model.

## 5. Transformation of a Discrete Event System Specification to Specification and Description Language

Based on a system theory, DEVS is a general formalism in which all other formalisms can be transformed.[31] Because

DEVS can be transformed to SDL, the full power of DEVS can be used. In addition, models can be represented graphically using SDL, which makes them more understandable to audiences that are not familiar with mathematical languages.

In the DEVS formalism, two levels can be used to define a model's behavior: coupled and non-coupled. When coupled models are used, the structure of the entire model can be defined. When non-coupled models are used, the behavior of simple model elements can be specified.

Once the behavior of the basic model elements has been defined, the model structure can be defined. This structure is defined by connecting the various basic elements with a known behavior.

The next section presents the equivalence between the SDL and DEVS formalisms at these two levels.

## 5.1 Coupled DEVS models and SDL

Simulation models can be specified in the DEVS formalism without having to describe the behavior of each model element. The structural relationships between identical elements can be defined. Such models are called coupled models.

In DEVS, there are two main types of coupled models: modular and non-modular. In modular coupling, the interaction of the various model components happens only at their entrances and exits. In non-modular coupling, the interaction occurs throughout the state. The literature shows that it is possible to change from one type of coupling model specification to the other.[30] Therefore, this paper focuses on the relationship between the SDL specification and the DEVS modular formalism. For simplicity, we work with the coupled DEVS model with ports. With this model, a series of input and output ports are described for each type of event that can be processed in the DEVS model. Figure 6 shows an example in which we combine the two models representing a queue and a processor.[2]

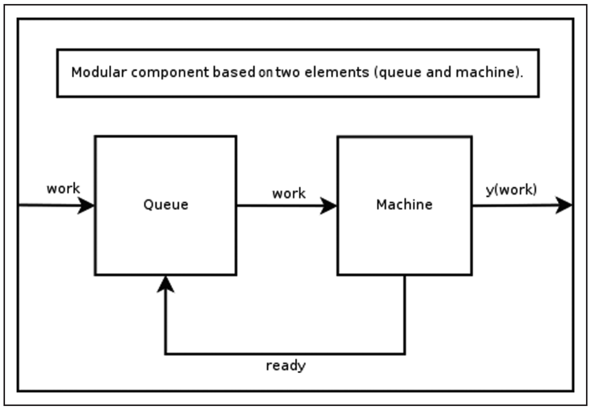

Modular component based on two elements (queue and machine).
work
Queue
work
Machine
y(work)
ready

**Figure 6.** Coupled Discrete Event System Specification model combining two models, one representing a Queue and the other a Machine.

The diagram that represents a coupled model is quite similar to the first level of an SDL specification. Therefore, the transformation problem lies in representing non-coupled models using SDL.

## 5.2 Behavior definition of non-coupled models

From a DEVS system specification, we extract state diagrams that represent internal and external transitions and convert them to SDL diagrams.

We use an example to describe the process and to show the behavior of a generator:[2]

$$\begin{aligned}&DEVS_{period} = (X, Y, S, \delta_{ext}, \delta_{int,} \lambda, \text{ta})\\&X = \{\}\\&Y = \{1\}\\&S = \{''passive'', ''\text{active}''\} \times R^+\\&\delta_{int} = (\text{phase}, \sigma) = (''active'', \text{period})\\&\lambda = (''active'', \sigma) = 1\\&ta = (\text{phase}, \sigma) = \sigma\end{aligned}$$

To transform this DEVS model to an SDL model, we first construct the states diagram. Because the number of states is not finite, we use the transitions to define the states (classes of states).

Only one transition is defined, $\delta_{int}$ = (phase,$\sigma$) = (‘‘active’’, period). In this transition, we reach the state ‘‘active’’, and remain in this state for $\sigma$ time (period time). The starting state for this transition can be ‘‘active’’ $\times R^+$. Although the number of states is infinite, it is only necessary to consider two states (phases): ‘‘passive’’ and ‘‘active’’ (see Figure 7).

The internal transition defined in $\delta_{int}$ is represented by INT (see Figure 8). We can define the SDL diagrams for each of the states. The SDL diagrams represent the transitions defined. In this example is represented the internal transition. For each internal transition it is necessary to define at least two procedure blocks. One procedure defines *ta*, and the second defines the *output function*. The internal event, INT, carries *ta* as a parameter that represents the delay required by the simulation engine to process the event.

As another example, we consider the Binary counter from Zeigler et al.[2] and its SDL equivalent diagram.

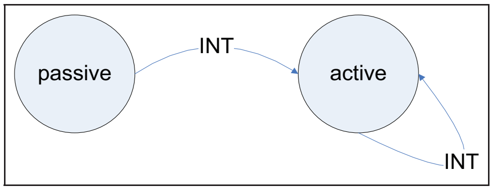


**Figure 7.** States diagram for the period Discrete Event System Specification model.

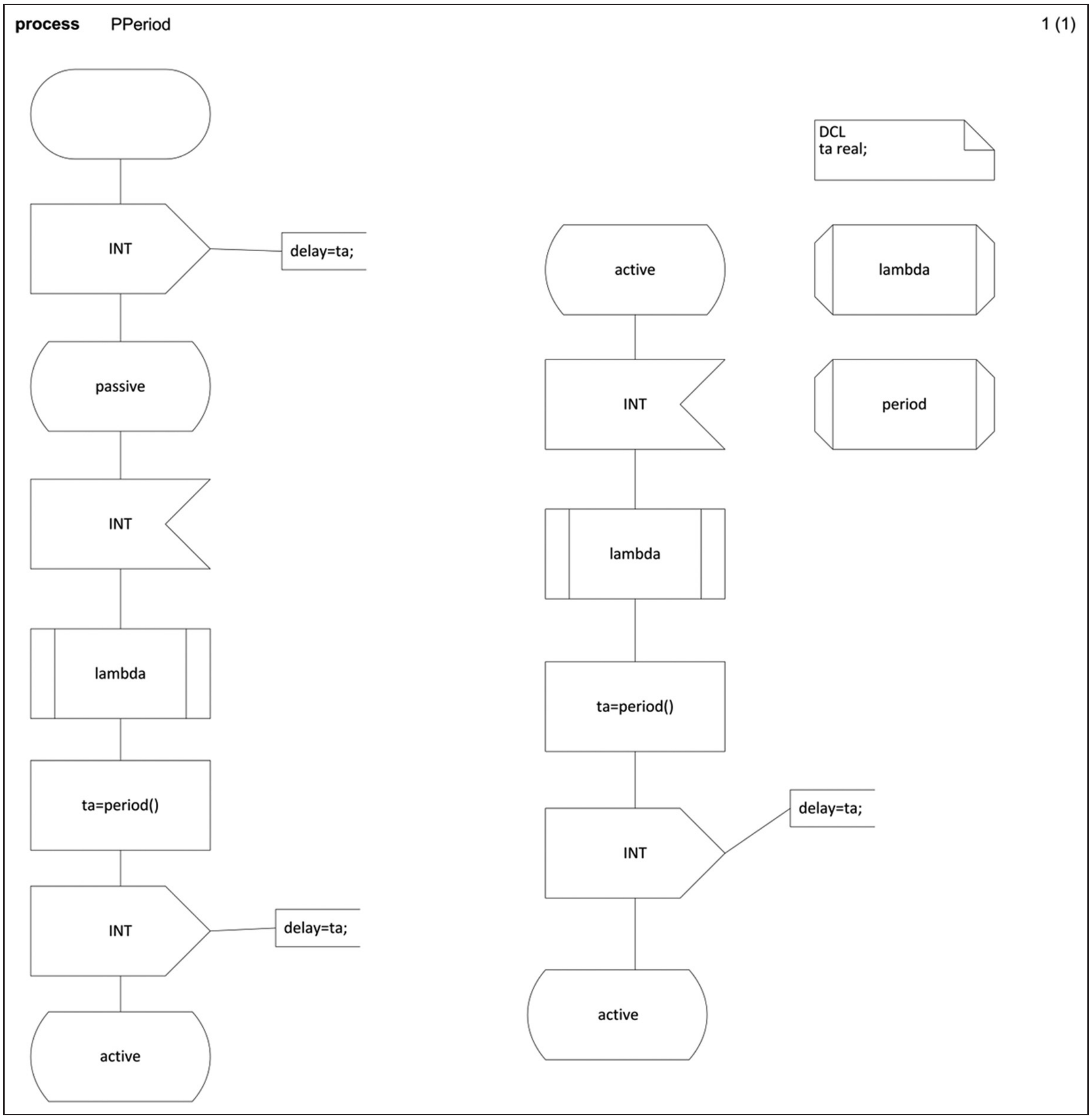


**Figure 8.** Specification and Description Language (SDL) diagram for the period Discrete Event System Specification (DEVS) model. The two states are "active and "passive". The usual action that a DEVS model performs when an element changes from one state to another due to an internal or an external event (SIGNAL in SDL) is described in the PROCESS. Note that the "passive" state can be neglected.

$$
\begin{aligned}
&DEVS_{binary\_counter} = (X, Y, S, \delta_{ext}, \delta_{int,} \lambda, ta)\\
&X = \{0, 1\}\\
&Y = \{1\}\\
&S = \{''passive'', ''\text{active}''\} \times R_0^+ \times \{0, 1\}\\
&\delta_{ext} = (''passive'', \sigma, \text{count}, e, x)\\
&= \begin{cases} if(\text{count} + x < 2)\ then\ (''passive'', \sigma, -\ e, count + x)\\ \qquad\qquad\qquad else(''active'', 0, 0) \end{cases}\\
&\delta_{int} = (\text{phase}, \sigma, \text{count}) = (''passive'', \infty, \text{count})\\
&\lambda = (''active'', \sigma, \text{count}) = 1\\
&ta = (\text{phase}, \sigma, \text{count}) = \sigma
\end{aligned}
$$

Again, the states are not finite, and we use the transitions to define the classes of states that allow the representation of the diagram. The two classes of states can be defined by "passive" and "active" states.

The internal transition only "acts" when the state is ("active", 0, 0), returning "1". Hence, there are two main classes of states: ("active", 0, 0) and ("passive", ∞, count).

The states diagram is shown in Figure 9.

We draw the external and internal transitions and take into account that *ta* is ∞, that is, this internal event must

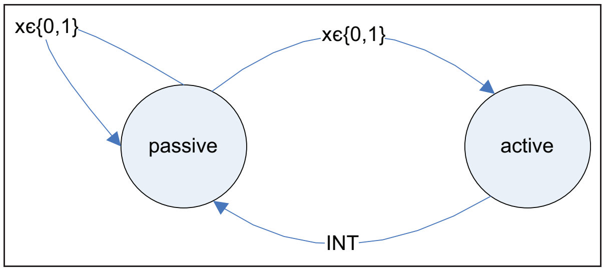


**Figure 9.** States diagram for the binary_counter Discrete Event System Specification model.

not be represented. In SDL, it does not make sense to send an event that will never be processed. The resulting diagram is given in Figure 10.

We detail the external transitions of the *binary_counter* model in a block diagram by defining the events we want to receive.

The semantics is preserved in both cases because the SDL diagrams contain the same states, the travel from one state to other are caused by the same signal (event) and the procedures executed (the λ function) are the same.

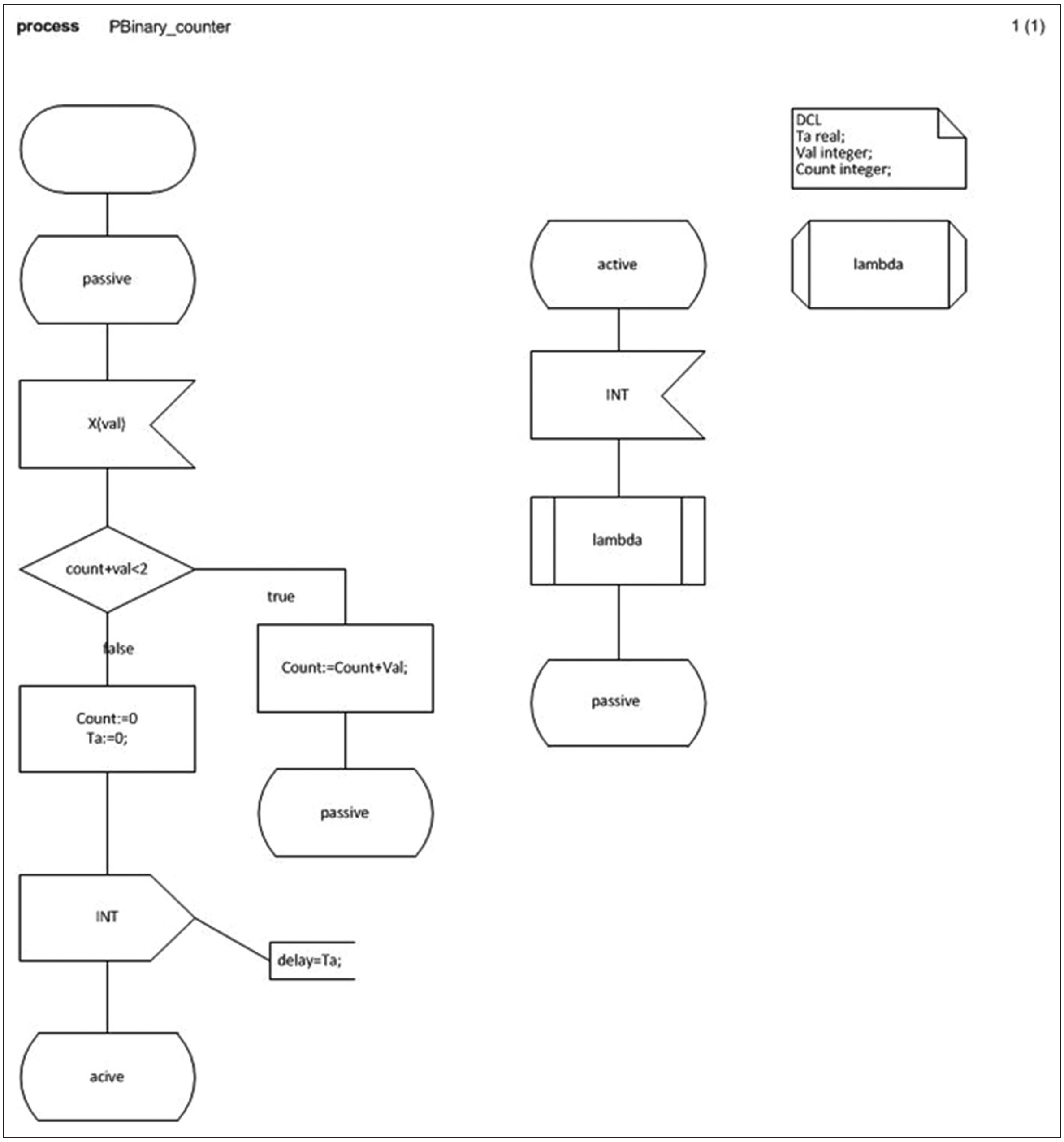


**Figure 10.** Specification and Description Language diagram for the binary_counter Discrete Event System Specification model.

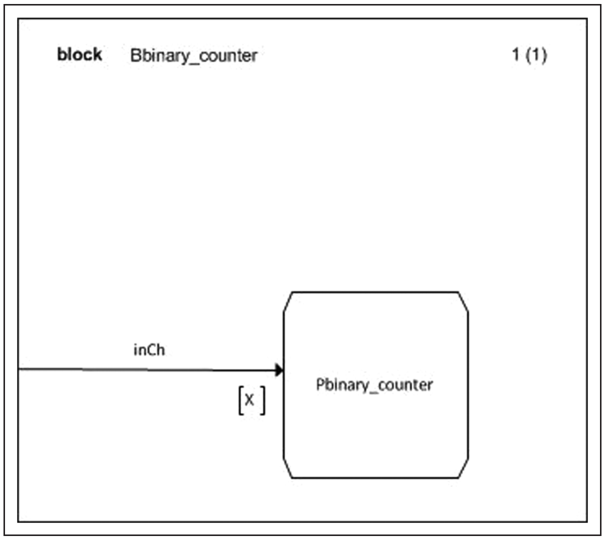


**Figure 11.** Block diagram for the binary_counter model. The output channel that represents the output function of the Discrete Event System Specification model is not represented in the diagram.

## 6. The algorithm (DEVStoSDL)

The proposed algorithm used to represent a DEVS atomic model using SDL diagrams is detailed in this section.

1. Represent the **external signals** that are received by the process through the definitions of external transitions, $\delta_{ext}$, in a **block** diagram.

In the period example, no external transition existed. In the *binary_counter* example, the block diagram is represented in Figure 11. Note that this defines the communication channels and events (SDL signals) of the model. All coupled models can be mapped to an equivalent SDL BLOCK. This allows representing the hierarchical decomposition of coupled DEVS models. Finally, to represent the atomic model we can use a SDL PROCESS. This PROCESS, like the atomic DEVS models, represents the behavior of the model, while the hierarchical decomposition of the model, represented on the BLOCKS diagram, represents the model structure.

2. For the **process** diagram, define the **states**, $S$ based on the transitions, $\delta_{ext}$ and $\delta_{int}$, that define the set of states.

The states diagram can always be represented because the number of states can be finite or infinite:

a. in the finite case, the state diagram is based on this finite set, and thus can be represented;
b. in the infinite, the states of the state diagram are based on the finite transitions defined on the DEVS model, and thus can be represented.

3. Draw the **external transitions** in the **process** diagram.

The transitions can own several conditions, as is the case of the external transition for the binary_counter DEVS model example:

$$\delta_{ext} = (''\text{passive}'', \sigma, \text{count}, e, x)$$
$$\begin{cases} \mathit{if}(\text{count} + x < 2)\text{then}(''\text{passive}'', \sigma, -e, \text{count} + x) \\ \qquad\qquad \text{else}(''\text{active}'', 0, 0) \end{cases}.$$

Hence, it is necessary to decompose the external (or the internal functions) in several branches on the SDL equivalent diagram, starting from a DECISION element that defines the starting point of the branches.

Also, note that for the external transition function $\delta_{ext} : Q \times X \rightarrow S$, the total state set $Q$ is defined as $Q = \{(s, t_e) | s \in S, t_e \in (T \cap [0, t_a(s)])\}$, representing the set of total states and $t_e$ is the elapsed time since the last event. On SDL, if necessary due to the external function calculus, we can store the elapsed time in a PROCESS variable every time an event (SIGNAL) is processed. SDL implements a **now()** method that allows obtaining the simulation clock for the PROCESS. This can be represented, as is proposed in the algorithm, in the PROCEDURE that encapsulates all the variables modifications. We need to follow the next algorithm for each external transition defined on the model.

```
for each condition of $\delta_{ext}$ {
    Represent the variables modified in a procedure
(one for each branch). Here if needed we can store $t_e$
($t_e$ = now();).
    If the final state for a branch of the $\delta_{ext}$ is a state
with an internal transition defined {
    DrawInternalTransitionOutputSignal
    }
    }
```

4. Draw the **internal transitions** in a **process** diagram.

For the internal transitions, the procedure is the same that for the external transition, but adding the output function. We need to follow the next algorithm for each internal transition defined on the model.

```
for each condition of $\delta_{int}$ {
    Represent the variables modified in a procedure
(one for each branch)
    Represent the output function in a procedure.
    If the final state for a branch of the $\delta_{int}$ is a state
with an internal transition defined {
    DrawInternalTransitionOutputSignal
    }
    }
```

The method DrawInternalTransitionOutputSignal is

```
Calculate ta in a task.
    if (ta < ∞) {
    Represent the output signal related to the event.
    Represent ta in the delay related to the output signal.
    }
```

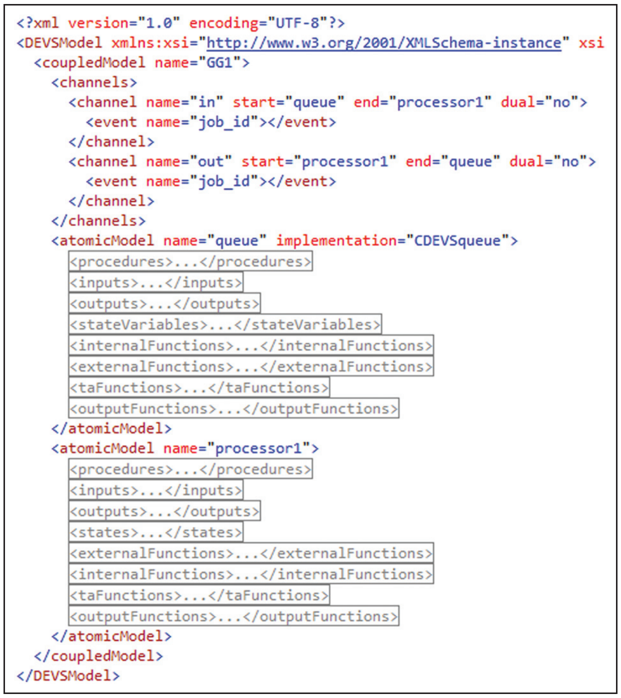
5. Define the **initial state** of the model by executing the **internal transitions** (those that can be executed). This defines the signals (internal) that the PROCESS must send before the PROCESS reaches its initial STATE.

### 6.1 SDLPS implementation of the algorithm

We implement the proposed algorithm using the XML representations for the SDL and DEVS models (DEVS-XML and SDL-XML) in the SDLPS[20] infrastructure. This allows us to obtain a new SDL-XML file that represents a DEVS model. In Figure 12, the DEVS model using XML is shown. From this DEVS-XML representation, we can use XML (SDL-XML) to obtain an equivalent model described using SDL. In Figure 13, the SDL-XML representation of the model is shown. The representation contains two processes: *queue* and *processor1*.

SDLPS implements all of these features and simulates models represented in SDL or DEVS languages. SDLPS has been built using C++ and C. The code related to the model is represented using a dynamic-link library (DLL), and the generation of the SDL-XML model is performed automatically using a plug-in in Microsoft Visio®. The main idea of this kind of formal language is to understand and share the model behavior; it is clear that some parts of the model will be easy to understand using code or a mathematical function. This is what happens when in DEVS we define the transitions or when in SDL we define procedures. The PROCEDURES represent the ‘‘last level’’ of a SDL specification, encapsulating a behavior that is not related with the time (it does not modify the simulation clock) . Hence, this is represented by code and, then, when this must be executed it must be encapsulated on a DLL or similar. Although SDL allows a graphical representation of the PROCEDURES, sometimes it is clearer to have a simple C function detailing this .

Figure 14 shows the DEVS GG1 model in SDLPS. Note that the DEVS model is not represented because the Microsoft Visio® plug-in we developed generates SDL-XML from an SDL Microsoft Visio® diagram. However, we cannot regenerate the diagram from an SDL-XML representation.



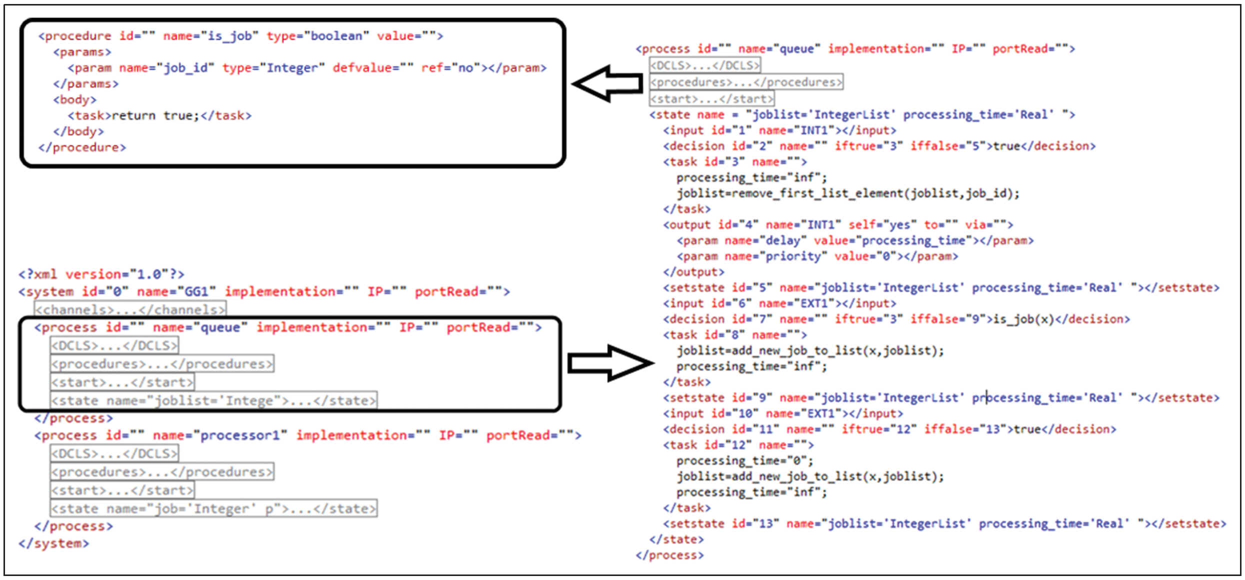
```
<?xml version="1.0" encoding="UTF-8"?>
<DEVSModel xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xsi
  <coupledModel name="GG1">
    <channels>
      <channel name="in" start="queue" end="processor1" dual="no">
        <event name="job_id"></event>
      </channel>
      <channel name="out" start="processor1" end="queue" dual="no">
        <event name="job_id"></event>
      </channel>
    </channels>
    <atomicModel name="queue" implementation="CDEVSqueue">
      <procedures>...</procedures>
      <inputs>...</inputs>
      <outputs>...</outputs>
      <stateVariables>...</stateVariables>
      <internalFunctions>...</internalFunctions>
      <externalFunctions>...</externalFunctions>
      <taFunctions>...</taFunctions>
      <outputFunctions>...</outputFunctions>
    </atomicModel>
    <atomicModel name="processor1">
      <procedures>...</procedures>
      <inputs>...</inputs>
      <outputs>...</outputs>
      <states>...</states>
      <externalFunctions>...</externalFunctions>
      <internalFunctions>...</internalFunctions>
      <taFunctions>...</taFunctions>
      <outputFunctions>...</outputFunctions>
    </atomicModel>
  </coupledModel>
</DEVSModel>
```

**Figure 12.** GG1 Discrete Event System Specification model.

## 7. Discussion

Firstly, it is necessary to remark that this transformation is not bidirectional; here the focus is on the transformation from DEVS to SDL. A transformation from an SDL specification to a DEVS specification will be little more complex due to the more flexible structure of SDL.

Now we analyze the applicability and correctness of this transformation. Reviewing the DEVStoSDL algorithm, we notice that the first step can always be performed; the number of external transitions that can be represented is finite for a DEVS specification. The second step can also be always performed; in the infinite case, we define the states based on the transitions of the DEVS model and the number of transitions is finite, implying that the number of states is finite. The third and fourth steps also can be done always, due to the finite nature of the DEVS transitions; SDL diagrams allow representing bifurcations, using the DECISION element to represent the transitions that are defined by conditionals. The transformation is defined and a SDL diagram depicting the overall structure that follows a DEVS model and all the specific model elements can be represented inside this diagram using variables and DECISION elements. In addition, the initial state can be calculated always. Finally, any existing function on the DEVS model can be represented in SDL PROCEDURES. Because all of the steps can always be applied, the method can always be used to transform DEVS models to SDL.

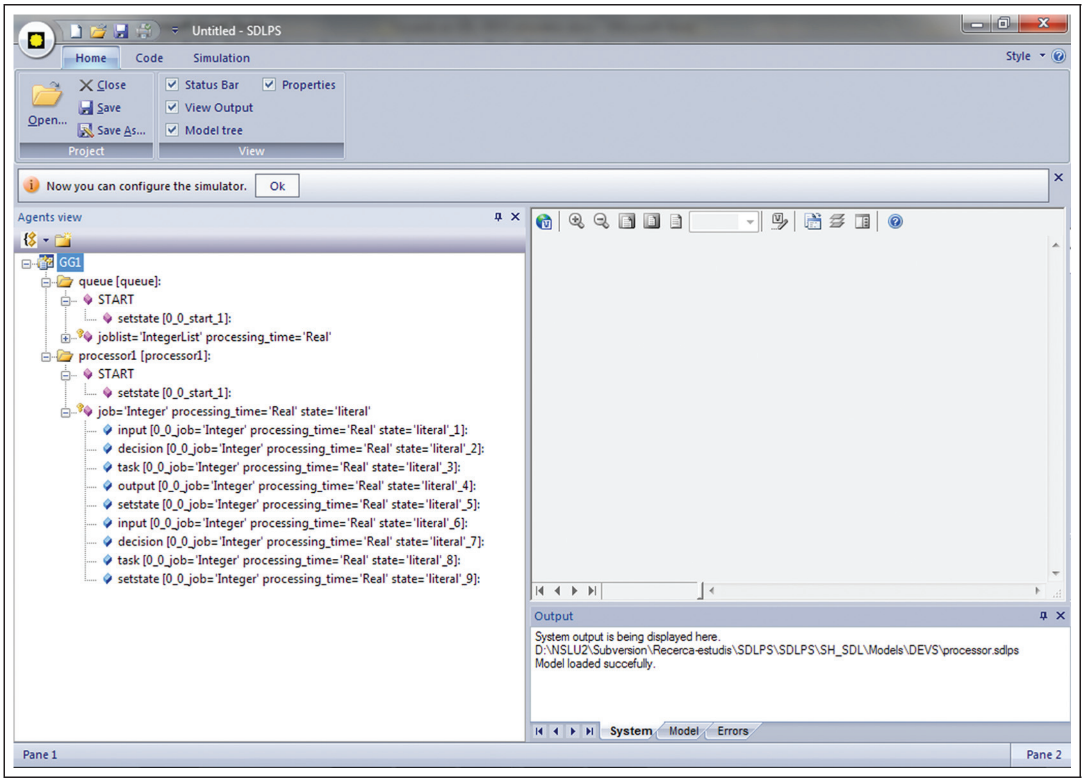


```
<procedure id="" name="is_job" type="boolean" value="">
  <params>
    <param name="job_id" type="Integer" defvalue="" ref="no"></param>
  </params>
  <body>
    <task>return true;</task>
  </body>
</procedure>
```

```
<?xml version="1.0"?>
<system id="0" name="GG1" implementation="" IP="" portRead="">
  <channels>...</channels>
  <process id="" name="queue" implementation="" IP="" portRead="">
    <DCLS>...</DCLS>
    <procedures>...</procedures>
    <start>...</start>
    <state name="joblist='Intege">...</state>
  </process>
  <process id="" name="processor1" implementation="" IP="" portRead="">
    <DCLS>...</DCLS>
    <procedures>...</procedures>
    <start>...</start>
    <state name="job='Integer' p">...</state>
  </process>
</system>
```

```
<process id="" name="queue" implementation="" IP="" portRead="">
  <DCLS>...</DCLS>
  <procedures>...</procedures>
  <start>...</start>
  <state name = "joblist='IntegerList' processing_time='Real' ">
    <input id="1" name="INT1"></input>
    <decision id="2" name="" iftrue="3" iffalse="5">true</decision>
    <task id="3" name="">
      processing_time="inf";
      joblist=remove_first_list_element(joblist,job_id);
    </task>
    <output id="4" name="INT1" self="yes" to="" via="">
      <param name="delay" value="processing_time"></param>
      <param name="priority" value="0"></param>
    </output>
    <setstate id="5" name="joblist='IntegerList' processing_time='Real' "></setstate>
    <input id="6" name="EXT1"></input>
    <decision id="7" name="" iftrue="3" iffalse="9">is_job(x)</decision>
    <task id="8" name="">
      joblist=add_new_job_to_list(x,joblist);
      processing_time="inf";
    </task>
    <setstate id="9" name="joblist='IntegerList' processing_time='Real' "></setstate>
    <input id="10" name="EXT1"></input>
    <decision id="11" name="" iftrue="12" iffalse="13">true</decision>
    <task id="12" name="">
      processing_time="0";
      joblist=add_new_job_to_list(x,joblist);
      processing_time="inf";
    </task>
    <setstate id="13" name="joblist='IntegerList' processing_time='Real' "></setstate>
  </state>
</process>
```

**Figure 13.** Extensible Markup Language (XML) representation of the Specification and Description Language model. The detailed XML representation of the PROCESS queue is shown on the right side. The PROCEDURE contains the code that is defined in the procedures Discrete Event System Specification model.



Now you can configure the simulator.

```
GG1
  queue [queue]:
    START
      setstate [0_0_start_1]:
    joblist='IntegerList' processing_time='Real'
  processor1 [processor1]:
    START
      setstate [0_0_start_1]:
    job='Integer' processing_time='Real' state='literal'
      input [0_0_job='Integer' processing_time='Real' state='literal'_1]:
      decision [0_0_job='Integer' processing_time='Real' state='literal'_2]:
      task [0_0_job='Integer' processing_time='Real' state='literal'_3]:
      output [0_0_job='Integer' processing_time='Real' state='literal'_4]:
      setstate [0_0_job='Integer' processing_time='Real' state='literal'_5]:
      input [0_0_job='Integer' processing_time='Real' state='literal'_6]:
      decision [0_0_job='Integer' processing_time='Real' state='literal'_7]:
      task [0_0_job='Integer' processing_time='Real' state='literal'_8]:
      setstate [0_0_job='Integer' processing_time='Real' state='literal'_9]:
```

Output

```
System output is being displayed here.
D:\NSLU2\Subversion\Recerca-estudis\SDLPS\SDLPS\SH_SDL\Models\DEVS\processor.sdlps
Model loaded succefully.
```

**Figure 14.** Specification and Description Language Parallel Simulator environment with the Discrete Event System Specification model loaded. The tree that contains all of the elements that define the model is on the left-hand side.

From the DEVS point of view, the simulation algorithms are methods to generate the model's legal behaviors, which means that the trajectories do not reach illegal states.[35] In Figure 15 we see the trajectories that we can obtain from the binary counter DEVS model; we depicted here the key elements that are represented in the SDL equivalent model. If with the equivalent SDL model we can obtain equivalent trajectories, both models can be

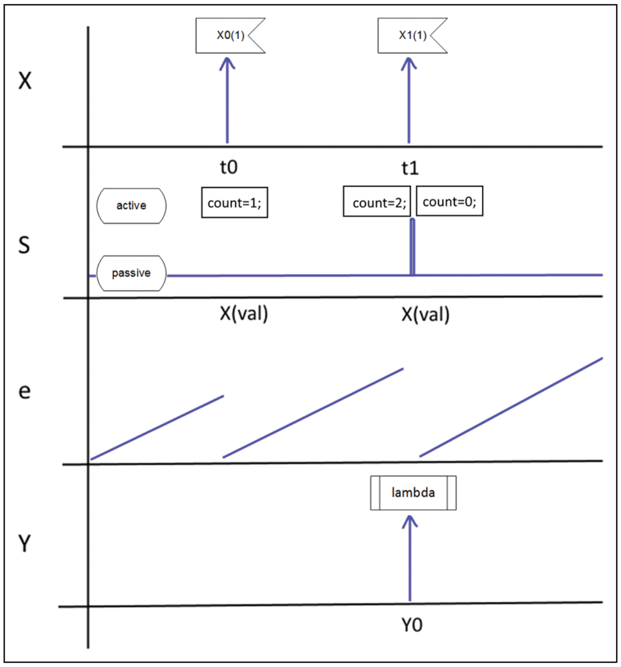


**Figure 15.** Trajectories for the binary counter model showing the Specification and Description Language (SDL) elements that will be represented on the equivalent SDL model. The states can be “passive” and “active”; when an external event is received (represented by the INPUT element), the variable “count” is increased by the value. In *S* we can see that the internal event is triggered when count $>$ 2. Also, *Y* represents the PROCEDURE that implements the lambda Discrete Event System Specification function.

considered equivalent. To assure this equivalence we are going to analyze the DEVS abstract simulators and compare them with the implementation we obtain from the SDL specification.

In SDL-2010, all the SIGNALs are stored at the input of the PROCESS, sorted by time and priority.[17] When a PROCESS consumes a SIGNAL, its time “$t$” is updated. A representation of the SDL abstract machine is described by Sanders[27] (see Figure 16), while a representation of the abstract machine for atomic DEVS can be reviewed in Zeigler.[35]

To assure that the trajectories we obtain with the SDL model are equivalent it is necessary to remember that the initial states for both models are equivalent (assured by the step 5 of the algorithm) and analyze the structure of the finite state machine of the SDL machine. Reviewing the DEVStoSDL algorithm, the resulting SDL branch for each internal event has the structure presented in Table 2 (right), while the abstract DEVS simulator is represented on the left.

Reviewing the DEVS abstract simulator, steps 1 and 2 are represented by lambda PROCEDURE CALL. If any variable in the function $\delta_{int}(s)$ is modified this can be represented on $\delta_{int}(s)$ PROCEDURE CALL ; step 3 is represented by the SET STATE($\delta_{int}(s)$), in the sense that if any conditional exists in $\delta_{int}(s)$ it will be represented on the SDL diagram using the DECISION element, which defines the different needed SET STATES. In the DEVS abstract simulator, two variables for representing time are defined, $t_l$ (representing the time of the last event) and $t_n$ (representing the time of the next internal event). Both variables are also represented in SDL, since the input port stores the events sorted by time (storing $t_n$), while the internal clock of the PROCESS stores $t_l$ time (these variables have the same meaning and are updated according to it). In SDL and in DEVS simulators we are expecting as the next internal event “$i$”, an event with the time $t_n$; if this is not true (in SDL this means that something is wrong in the SIGNAL queue) an error is handled ($t \neq t_n$). Note that if $ta = \infty$ then no OUTPUT is defined in SDL since $t_n = \infty$, meaning that this event never will be processed. In both simulators, the trajectories we are going to obtain are equivalent.

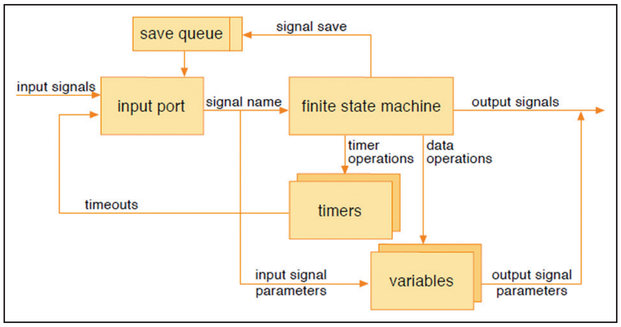


**Figure 16.** Specification and Description Language machine.[27] The finite state machine in our case is following a specific structure.

A similar analysis is done for the external events. Based on “View 1”,[35] Table 3 shows the DEVS abstract simulator and the PROCESS structure we got from the DEVStoSDL algorithm.

The first step is represented by the SET STATE; again if $\delta_{ext}(s, t-t_l, x)$ modifies any variable this can be represented by $\delta_{ext}(s, t-t_l, x)$ PROCEDURE CALL; also, if any conditional exists on $\delta_{ext}(s, t-t_l, x)$ it will be represented in the SDL diagram using the DECISION element. Adding in the SDL diagram the OUTPUT element depends if the final state an internal transition is defined or not. Also, if we are receiving an event and ($t_l \leqslant t$ and $t \leqslant t_n$) == false then we must report an error; in SDL this condition must be assured also, since we cannot receive SIGNALS from the past and we cannot process a SIGNAL prior to processing others that have a smaller time stamp . This error implies a problem in the SIGNALS queue.

Since the trajectories we obtain on both simulators are equivalent, both models can be considered equivalent.

**Table 2.** Comparing the Discrete Event System Specification abstract simulator with the obtained PROCESS to be executed on the Specification and Description Language simulator for internal events.

1. generate an output ($y \leftarrow \lambda(s)$)
2. send this output to his parent (send *y*-message(*y*,*t*) to parent)
3. update the state ($s \leftarrow \delta_{int}(s)$), and
4. update the times $t_l$ that represents the time of the last event and $t_n$ that represents the time for the next event ($t_l \leftarrow t$; $t_n \leftarrow t_l + ta(s)$;)

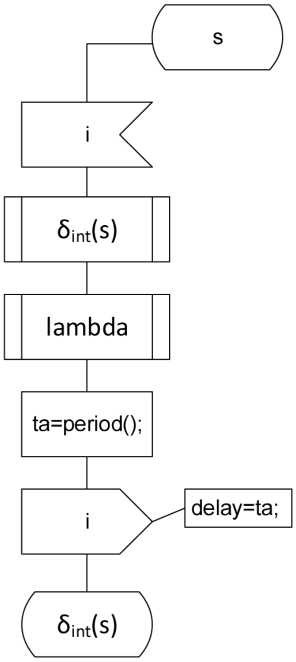
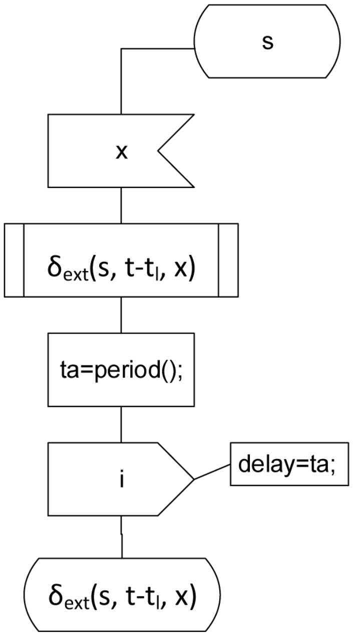


**Table 3.** Comparing the Discrete Event System Specification abstract simulator with the obtained PROCESS to be executed on the Specification and Description Language simulator for external events.

1. update the state, $s \leftarrow \delta_{ext}(s, t\text{–}t_l, x)$
2. update the time $t_l$ that represents the time of the last event and $t_n$ that represents the time for the next event ($t_l \leftarrow t$; $t_n \leftarrow t_l + ta(s)$;)


s
x
$\delta_{ext}(s, t\text{-}t_l, x)$
ta=period();
i
delay=ta;
$\delta_{ext}(s, t\text{-}t_l, x)$

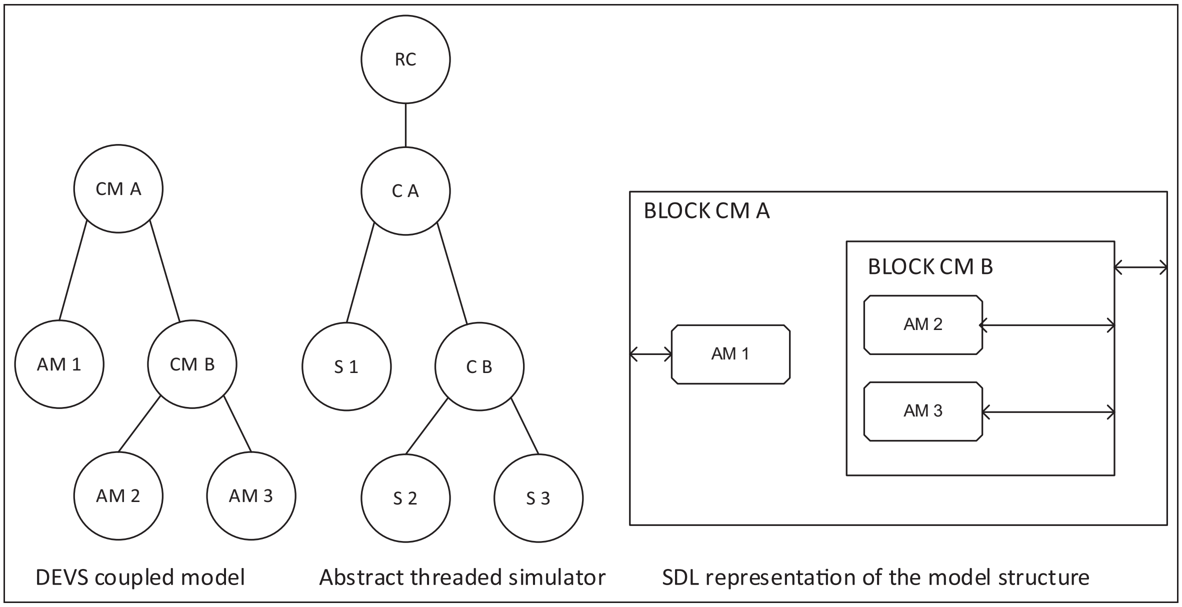


**Figure 17.** Mapping of the abstract threaded simulator tree, based on Wainer and Mosterman.[36] AM: atomic models; CM: coupled models; RC: root coordinator; CL: coordinators; S: simulators; DEVS: Discrete Event System Specification; SDL: Specification and Description Language.

### *7.1 Interoperability*

Here we discuss briefly the interoperability between SDL and DEVS models and how this algorithm improves it. As stated by Vangheluwe,[1] several alternatives to combine simulation models and simulators exist. Obviously, with the proposed algorithm, a common simulator can be achieved when a modeler transform DEVS models to SDL. However, this is not the only alternative to combine both formalisms. In Figure 17 we can see the mapping of a DEVS coupled model to an abstract threaded simulator and a representation of the model structure based on a SDL. It is remarkable that the hierarchy in SDL is so similar, allowing also to express those elements that can be executed on parallel threads with BLOCKS and those that must be executed sequentially with PROCESS.

As is studied in this paper, the behavior of the DEVS atomic model can be represented in SDL PROCESS; hence, the DEVS model can be represented graphically, on the last level, using SDL diagrams that come from a previous DEVS specification. Also starting from a DEVS specification, we can obtain a SDL representation enabling the use of SDL tools such as PragmaDev SARL,[21] CINDERELLA SOFTWARE,[18] IBM. TELELOGIC[19] or that described by Fonseca i Casas[20] to simulate DEVS models.

It is outside the scope of the paper to show how to automatize the transformation of coupled models or parallel models, but we want remark on the huge similarities between DEVS and SDL at couple level (structure), as shown in Figure 17. Also, due to the versatility that currently exists in SDL to represent priorities a parallel DEVS model can be achieved; in SDL-2010 we can add priorities at PROCESS level or at SIGNAL level, allowing complete control of how the SIGNALS are going to be processed in the final input port of the PROCESS (see Figure 16).

## 8. Concluding remarks

In this paper, we presented an algorithm that allows atomic DEVS formalisms to be transformed to SDL PROCESSES. Thanks to this, both formalisms can be used in the same project to represent a simulation model following the common formalism methodology. In addition, SDL diagrams can be used as a graphical representation for DEVS models. In summary, we can construct simulation models by combining the best features of these two formalisms and define a simulation model using the powerful DEVS syntax and the standard graphical representation of SDL.

In addition, in this paper we propose a XML representation for the DEVS and SDL formalisms. For SDL, a standard textual representation of the model exists following SDL/PR. However, no standard XML representation for SDL exists, (development of a standard is an ongoing project of the ITU-T working groups). Currently, there is not a standard to represent a DEVS model textually. The schema presented here can be used as a starting point for groups that are involved in defining a much-needed common representation for DEVS.

A specialist can use DEVS models, translate them to the SDL-XML representation and apply SDL commercial tools that provide automatic code generation to implement a DEVS model. Co-simulation and the reuse of existing models in new simulation models are possible with this approach.

A growing number of people recognize the need for the use of simulation tools to understand and possibly predict the problems of a system. However, in complex environments, with complex systems, the diversity of tools and languages used by multidisciplinary teams can be a hindrance. Mechanisms to share and combine models must be established. Mechanisms to improve models by combining the strengths of all of the tools are imperative.

## Funding

This research received no specific grant from any funding agency in the public, commercial or not-for-profit sectors.

## References

1. Vangheluwe HLM. DEVS as a common denominator for multi-formalism hybrid systems modelling. In: *IEEE international symposium on computer-aided control system design*, 2000. pp.129–134.
2. Zeigler BP, Praehofer H and Kim TG. *Theory of modeling and simulation*. Orlando, FL: Academic Press, 2000.
3. Traoré MK. A graphical notation for DEVS. In: *proceedings of the 2009 spring simulation multiconference*, http://dl.acm.org/citation.cfm?doid=1639809.1655391 (2009, accessed 1 October 2009).
4. Ighoroje UB, Maïga O and Traoré MK. The DEVS-driven modeling language: syntax and semantics definition by meta-modeling and graph transformation. In: *proceedings of the 2012 symposium on theory of modeling and simulation*, 2012.
5. Song HS and Kim TG. DEVS diagram revised: a structured approach for DEVS modeling. In: *proceedings of EUROSIS 2010*, Ostend, Belgium, 2010.
6. Kidisyuk K and Wainer GA. CD++ Modeler: a graphical toolkit to develop DEVS models. In: *proceedings of the 2008 spring simulation multiconference*; San Diego, CA, http://dl.acm.org/citation.cfm?id=1400696 (2008, accessed 1 October 2012).
7. Bonaventura M, Wainer GA and Castro R. Advanced IDE for modeling and simulation of discrete event systems. In: *proceedings of the 2010 spring simulation multiconference*, San Diego, CA, 2010.
8. Wainer G and Liu Q. Tools for Graphical specification and visualization of DEVS models. *Simulation* 2009; 81: 131–158.
9. Sarjoughian HS.CoSMoSim, http://cosmosim.sourceforge.net/ (2009, accessed 28 November 2014).
10. Sarjoughian HS, Sarkar S and Mayer GRA novel visual CA modeling approach and its realization in CoSMoS. In: *proceedings of the 2010 spring simulation multiconference*, San Diego, CA, 2010.
11. Villalba CM, Urquia A, Moallemi M, et al. DEVS-GRAPH in Modelica for real-time simulation. In: *proceedings of European conference on modeling and simulation (ECMS)*, http://cell-devs.sce.carleton.ca/publications/2012/VUMW12 (2012, accessed 1 October 2012).
12. Zeigler BP, Song HS, Kim TG, et al. DEVS framework for modelling, simulation, analysis, and design of hybrid systems. In: *hybrid systems'*94, 1994.
13. Wainer GA, Al-Zoubi K, Dalle O, et al. DEVS standardization: foundations and trends. In: Wainer GA (ed.) *Discrete event simulation and modeling: theory and applications*, 2010.
14. Ighoroje UB, Maïga O and Traoré MK. The DEVS-driven modeling language: syntax and semantics definition by meta-modeling and graph transformation. In: *proceedings of springsim 2012*, Orlando, Florida, 2012.
15. Sarjoughian HS and Chen Y. Standardizing DEVS models: an endogenous standpoint. In: *proceedings of the 2011 symposium on theory of modeling & simulation: DEVS integrative M&S symposium*, 2011.
16. Mittal S and Martín JLR. DEVS modeling language: DEVSML, http://devsml.sourceforge.net (2006, accessed 18 February 2015).
17. ITU-T. Z.100. Specification and description language—overview of SDL-2010, http://www.itu.int/rec/T-REC-Z.100-201112-I/en (2012, accessed 20 November 2012).
18. CINDERELLA SOFTWARE. Cinderella SDL, http://www.cinderella.dk (2007, accessed 31 March 2009).
19. IBM. TELELOGIC, http://www.telelogic.com/ (2009, accessed 31 March 2009).
20. Fonseca i Casas P. SDL distributed simulator. In: *winter simulation conference 2008*, Miami, 2008.
21. PragmaDev SARL. PragmaDev—code generation, http://www.pragmadev.com/product/codeGeneration.html (2012, accessed 18 February 2015).
22. ITU-T. Z.109. Specification and description language—unified modeling language profile for SDL-2010, http://www.itu.int/rec/T-REC-Z.109-201204-I/en (2012, accessed 10 November 2012).
23. ITU-T. Testing test control notation version 3: TTCN-3 core language. ITU-T Z-series recommendations. International Telecommunication Union ITU-T Z.161, Available from: http://www.itu.int/ITU-T/recommendations/rec.aspx?id=12346&lang=en (2013, accessed 18 February 2015).
24. Risco-Martín JL, Cruz JMdl, Mittal S, et al. eUDEVS: executable UML with DEVS theory of modeling and simulation. *Simulation* 2009; 85: 29.
25. Kapos GD, Nikolaidou M, Dalakas V, et al. An integrated framework to simulate SysML models using DEVS simulators. In: Fonseca i and Casas P (ed.) *Formal languages for computer simulation: transdisciplinary models and applications*. Hershey, PA: IGI Global, 2013, pp.305–332.
26. Reed R.SDL-2000 form new millenium systems. *Telektronikk* 2000; 4: 20–35.
27. Sanders R.Implementing from SDL. *Telektronikk* 2000; 120–129.
28. Telecommunication standardization sector of ITU. Specification and Description Language (SDL), http://www.itu.int/ITU-T/studygroups/com17/languages/index.html (1999, accessed April 2008).

29. SDL Tutorial. IEC international engineering consortium, http://www.sdl-forum.org/SDL/Overview_of_SDL.pdf (accessed 18 February 2015).
30. Zeigler BP, Kim TG and Praehofer H. DEVS formalism as a framework for advanced distributed simulation. In: *first international workshop on distributed interactive simulation and real time applications (in conjunction with the international symposium on modeling, analysis and simulation of computer and telecommunication systems (MASCOTS'97)*, Eilat, Israel, 1997.
31. Zeigler BP and Vahie S. EVS formalism and methodology: unity of conception/diversity of application. In: *Winter simulation conference*, Los Angeles, California, 1993, pp.573–579.
32. Fonseca i and Casas P. *Formal languages for computer simulation: transdisciplinary models and applications*. Hershey, PA: IGI Global, 2013.
33. Fonseca i Casas P. Towards an automatic transformation from a DEVS to a SDL specification. In: *procediings of the 2009 summer simulation multiconference*, Istanbul, Turkey, 2009.
34. Risco-Martín JL, Mittal S, López-Peña MA, et al. A W3C XML schema for DEVS scenarios. In: *spring simulation multiconference 2007*, Norfork, Virginia, 2007, pp.279–286.
35. Zeigler BP. *Multifacetted modeling and discrete event simulation*. London/Orlando, FL: Academic Press, 1984.
36. Wainer GA and Mosterman PJ. Discrete-Event modeling and simulation. Boca Raton, FL: CRC Press, Taylor & Francis Group, 2011.

## Author biography

**Pau Fonseca i Casas** is a professor in the Department of Statistics and Operations Research of the Polytechnic University of Catalonia, teaching in the statistics and simulation areas. He obtained his masters degree in computer engineering in 1999 and his PhD in 2007 from Polytechnic University of Catalonia. He also works in the InLab FIB (http://inlab.fib.upc.edu/) as a head of the environmental simulation area, developing simulation projects since 1998. His website is http://www-eio.upc.es/~pau/. His research interests are discrete simulation applied to industrial, environmental and social models, and the formal representation of such models.