3
A Framework for Modeling and Simulation in Serious Games

3.1
Introduction

This chapter first discusses the desirable characteristics of a framework for modeling the dynamics of serious games, considering the requirements enumerated in section 1.3. Very briefly, the framework must allow:

1. The integration of different dynamic models, expressed in a variety of formalisms, avoiding the creation of dependency relations among them as much as possible. This is important to achieve modularity, allow flexible scenario composition, and facilitate reuse of simulation models.
2. The inclusion of dynamic models into a game architecture with minimum performance impact.
3. The communication with external asynchronous entities during game play. This communication may affect the outcome of the game simulation.

The discussion on the requirements led to the conception of the process-oriented simulation (POS) paradigm for modeling and simulation, whose characteristics are described in the form of design decisions, listed in section 3.2. Then, this chapter introduces a novel modeling and simulation formalism, called Process-DEVS, which is described in detail in section 3.3.

3.2
A Discussion on the Framework Requirements

This section provides a more detailed discussion on the requirements, which helps justify the framework design decisions.

PUC-Rio - Certificação Digital Nº 0821408/CA

The discussion is carried out at a considerably high level of abstraction. Some decisions are sometimes based on subjective arguments and they are not intended to suit all possible serious games. However, they do intend to produce a highly general and extensible architecture that will suffice for most cases. This discussion also aims at helping detect if the proposed framework is actually the best option for implementing a particular game.

3.2.1 On the Nature of Time

The requirement for realism in the context of serious games raises the central question of how to model dynamic systems so that they can be simulated during game play. With respect to how they model state change in time, dynamic models can be categorized, at the highest abstraction level, as *discrete* or *continuous*. In discrete models, changes are modeled by *state transition functions*, which, at a given point in time, are invoked to determine the next state of the system, taking the previous one as input. An example of a discrete model is a banking account which, when receiving a deposit, has its value immediately updated. In continuous models, the state of the systems changes continuously in time. At each time instant, the model defines a *change rate* for each numeric variable that composes the state of the system. An example of a continuous model is the level of water in a tank, which changes continuously as a function of the incoming and outgoing water flows.

Discrete models can be further categorized into *discrete event models* and *discrete time models*. The difference is that discrete event models operate in a continuous time base, while in discrete time models time may only assume values from a discrete set. In discrete event models, every state change is called an *event*, which always happens at one particular time instant. The bank account example fits in this category. Discrete time models consist of a stepwise mode of execution where the state transition functions are invoked at each time step. Cellular automata are an example of such kind of model.

Since the state changes in discrete time models are modeled by state transition functions that happen at specific points in time, discrete time models

PUC-Rio - Certificação Digital Nº 0821408/CA

can be seen as a specialization of the more general discrete event model class. Figure 3.1 illustrates the major classes of models.

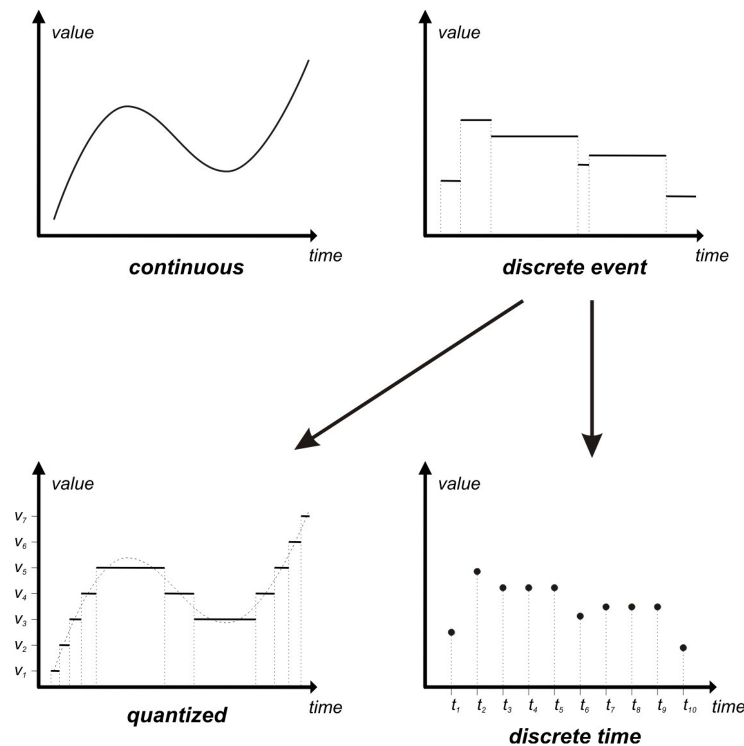

Figure 3.1 – Continuous, Discrete Event, Discrete Time and Quantized Process Models

Continuous models are usually described as systems of differential equations. Although these *differential equation systems* (DES) represent continuous processes, they may be simplified into discrete models either by discretization of time or discretization of the variables domains. While the first leads to a discrete time model, the later leads to a discrete event model, called a *quantized model* (Zeigler et al. 2000), which is also illustrated in Figure 3.1.

Continuous models provide potentially unbounded precision with respect to time. Ideally, DES simulators should be able to *solve* their models analytically. However, the great majority of the available simulators use numerical methods because of performance and scalability issues. Instead of solving models, these simulators employ numerical methods to *run* their models, generating an artificial history of the system and collecting observations to be analyzed (Banks et al. 2005). All that suggests that, even if a continuous formalism is used for describing the dynamic models, its underlying simulation machine should be of a discrete nature.

PUC-Rio - Certificação Digital Nº 0821408/CA

Although some traditional DES formalisms such as Systems Dynamics (Forrester 1972) and Bond Graphs (Paynter 1961) have been popularly used in areas such as physics, business, economics, and social modeling, several problems remain with these continuous approaches (Michel et al. 2009): (1) Only a global perspective is possible, which hurts modularity; (2) It is hardly possible to consider micro-level interactions, as in multi-agent systems; (3) It is not possible to model individual actions; (4) Integrating non-quantitative aspects is hard.

Even though continuous and discrete models are distinct in their nature, it is not always necessary to make an exclusive choice between them. The creation of *hybrid models* (Cellier 1986; Praehofer 1991; Deshpande et al. 1997; Lee and Zheng 2005) made it possible to simulate both kinds simultaneously. However, as Zeigler et al. (2000) points out, this incurs in performance loss. What is commonly seen in practice is the use of discrete formalisms to model continuous systems (Banks et al. 2005), easing the modeling and simulation tasks at the cost of some precision.

Considering the two main classes of discrete models, namely discrete time models and discrete event models, the discrete time class is clearly more specific and restricted. On the other hand, it is more intuitive and easier to modeling (Zeigler et al. 2000). However, there are two problems with the discrete time approach: (1) The granularity of time is fixed, which makes it difficult to integrate processes modeled with different time granularities (Banks et al. 2005); (2) In some cases where the state of most simulation elements is changed sparsely in time, the performance of a discrete time model can be rather poor, as compared to a corresponding discrete event model. Zeigler et al. (2000) illustrates well this problem in the domain of cellular automata.

Given this process modeling background, it is now possible to justify the first decision behind the framework for modeling and simulation in serious games:

Decision 1: The class of discrete event models will provide the basis on which to build all dynamic elements of the game.

PUC-Rio - Certificação Digital Nº 0821408/CA

This decision is grounded on the following arguments:

- Pure continuous or hybrid models were found to be more performance-costly.
- There are approaches to build platforms on which it is possible to integrate multiple models defined in any discrete subformalism in a scalable and parallelizable way (Praehofer et al. 1993; Vangheluwe 2000). This is much more difficult to be accomplished with continuous or hybrid models.
- Differential equation models can still be used in pure discrete simulation through discretization. The infinite precision of continuous systems may not be so important since the error can be controlled by increasing the granularity of value discretization.
- During the simulation of discrete models, it is easy to make the state always ready to be rendered for the players. In continuous or hybrid models, in order to render the state at time t, it is necessary to solve the equations for t, making rendering less immediate. One option would be to determine t in advance, but it was shown in section 2.1.1 that this is not possible for uncoupled game loops. Therefore, rendering performance will be hurt in that case.

3.2.2 On the Nature of Simulation Elements

The types of elements required for modeling and simulation in a serious game may vary in some aspects. The complexity of their behavior may range from a simple inanimate physical object to sophisticated artificial intelligence (AI) algorithms capable of simulating human reasoning in some context. Another aspect is the role of each game element. Simulation elements may play distinct roles such as parts of the game environment, proactive actors or natural phenomena.

We focus the discussion on two main classes of simulation approaches based on discrete-event time representation: *object-oriented simulation* (OOS) and

PUC-Rio - Certificação Digital Nº 0821408/CA

agent-oriented simulation (AOS). Uhrmacher and Swartout (2003) provide a good introduction to the main concepts of these approaches.

In OOS, a simulation is typically defined by a network of objects, which have hidden internal states and interact with each other by sending and receiving messages. The term *object* is not consensual in the simulation field. Some specific frameworks give their simulation components different names such as *models* (Eker et al. 2003) or *systems* (Zeigler et al. 2000).

Although the notion of object in OOS is distinct from that in object-oriented programming languages, they share some common principles. In object-oriented programming languages, objects are software entities with an internal state and a set of operations, through which they can interact with each other. The idea of object orientation is to increase modularity relative to plain procedural programming. Ideally software pieces with similar concerns should be brought together and organized in a single software entity. OOS follows the same principle by modeling simulation elements as objects with a definite boundary and a hidden internal state. Typically, OOS approaches also provide the notions of *classes* and *inheritance*, which are important properties to achieve code reuse.

Almost all of the main OOS formalisms support *composition*. Objects may be composed of other objects that are kept internal to it. This is the ground for multi-level modeling in object-oriented models. The *DEVS* formalism, as described in section 2.2.1, is a good representative of these object-oriented simulation concepts.

On the other main group of simulation approaches, agents are defined as autonomous entities which also have a hidden internal state. They are usually embedded in a *multi-agent system* which provides an environment that they can observe through sensors and change through effectors. They also communicate with each other by exchanging messages. These basic characteristics are present in most agent-oriented formalisms. More specialized characteristics of agents are not entirely consistent among researchers. However, a considerable number of them analyze the behavior of agents in mental terms such as beliefs, goals and desires. Therefore, the notion of an agent intuitively communicates the idea of something more complex than an object.

With respect to their acting, agents are classified as *deliberative*, *purely reactive* or both. Purely reactive agents base their present actions only on stimuli

PUC-Rio - Certificação Digital Nº 0821408/CA

received in a recent past. Agent deliberation is the act of predicting the future with the objective of planning its actions. Acting accordingly to plans based on an internal model of the world is what distinguishes between deliberative and reactive agents. Since objects typically intend to reproduce reactive rather than deliberative behavior by means of their internal states and transitions, the notion of deliberative agents seems more aggregative to the discussion on simulation elements for games. Therefore, in the context of this discussion, the term *agent* will denote an entity characterized by cognitive properties such as intentions, beliefs, desires and plans that are responsible for its goal-oriented rational behavior.

While OOS is aimed at modularity and reuse, AOS intends to improve interoperability by focusing on the interaction between agents and with a dynamic environment. It was not by chance that objects have grown as a standard way to model knowledge about dynamic systems and agents are usually used for the investigation of distributed AI phenomena such as cooperation and emergent behavior. Both objectives are useful for serious games. Therefore, a framework capable of incorporating the main benefits of both approaches would be appreciated.

The two kinds of dynamic systems are illustrated in Figure 3.2. In both approaches, entities have a definite boundary, an internal state and interact with others through message exchanging mechanisms. The difference is that agent-oriented approaches tend to work with relatively more specialized state and message sets. This suggests the view of *agents as specialized objects* (Uhrmacher 1997). Indeed, objects represent individual entities with some degree of autonomy who exchange messages when events are triggered. As an example, in the DEVS formalism, introduced in section 2.2.1, an agent can be modeled as an atomic DEVS model perceiving and effecting the environment through its input and output ports. It can react to external perturbations using its external transition function and also proactively with its internal transition function.

PUC-Rio - Certificação Digital Nº 0821408/CA

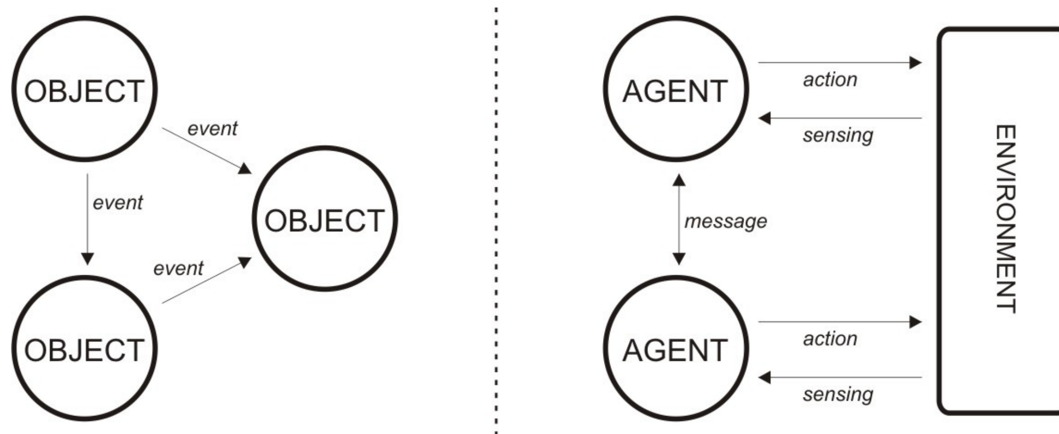

Figure 3.2 – Object- and Agent-Oriented Simulation

Several toolkits have been implemented using the AOS approach on top of OOS, such as JAMES II (Himmelspach and Uhrmacher 2007), SeSAm (Klügl 2009), RePast (North et al. 2006) and Swarm (Minar et al. 1996). Those toolkits also benefit from the greater maturity of OOS engines, since OOS is considerably older than AOS in the simulation field. Particularly, pure AOS toolkits tend to use equidistant time steps for all simulation elements, neglecting the fact that complex realistic simulations often use different time scales in different sub-elements of it (Troitzsch 2009). The toolkits mentioned above overcome this simplicity by inheriting the discrete-event time representation of their underlying OOS approaches.

By embedding agents into object-oriented simulation systems, it is possible to overcome some typical restrictions of multi-agent testbeds by combining agents and other types of objects in the same simulation. This flexibility helps integrating existing dynamic models within multi-agent systems, thereby producing a more realistic simulation (Uhrmacher 1997). These observations lead to the following decision:

Decision 2: The proposed framework will adopt the basic characteristics of object-oriented simulation. Its simulation elements will keep their internal states hidden and they will be organized as a network. The simulation elements will interact with adjacent elements by sending events to each other. Additional libraries will provide support for more specialized elements, such as agents.

PUC-Rio - Certificação Digital Nº 0821408/CA

The following arguments further justify the decision:

- If agents are modeled as objects, then the simulation platform will impose fewer restrictions on the nature of its elements.
- OOS models exhibit greater modularity than AOS models and thereby facilitate reuse. Additionally, OOS models make it easier to integrate other dynamic modeling formalisms, such as State Charts, Petri Nets and Cellular Automata (Himmelspach and Uhrmacher 2007).
- It is still possible to provide agent-oriented or other higher level formalisms by creating libraries on top of the basic OOS layer.

The adoption of these basic characteristics does not necessarily mean that the proposed framework is a special case of OOS. Indeed, non-OOS characteristics will still be considered in what follows. Therefore, it is still useful to further analyze other characteristics of AOS.

Unlike AOS, OOS lacks the notion of a global environment. Each object has its own environment defined by its input and output couplings. This is not an issue specific to OOS, but to all kinds of systems aiming at modularity and code reuse (Uhrmacher 1997). However, the notion of environment is present in most gaming frameworks. This happens naturally because one of the main features of games is precisely the simulation of the interaction of actors that takes place in some environment.

In OOS, common environments, such as spatial structures, can be modeled as a specialized object or a composition of objects. Models such as Timed Cell-DEVS (Wainer and Giambiasi 2001) have been tested in the domain of cellular spaces. Indeed, if agents are modeled as specialized objects, it seems natural that their environment is also modeled as specialized objects. This is a good example of how object-orientation strives for uniformity, treating communication and interaction with the environment indistinctly as discrete events, as in Figure 3.2.

Although keeping high levels of modularization, modeling the environment as a regular object has a potentially substantial drawback. Consider a large set of agents that need to sense a large volume of environmental data with some frequency. Since the internal states of all objects are hidden, the agents cannot

PUC-Rio - Certificação Digital Nº 0821408/CA

read all that data directly. The environment has to copy and transmit all necessary data to each sensing agent via messages, which may be unacceptable, depending on the number of agents and the frequency they need to sense the environment. In fact, this was one of the reasons for the rise of AOS, when OOS was already a well established modeling paradigm.

In the context of serious games, one may have to sacrifice modularity in favor of better performance, which in turn implies direct access to the game environment state. This observation leads to the following decision:

Decision 3: The environment is modeled as a simulation element whose internal state will be directly queried by other simulation elements. In all other aspects, it will be treated as a regular simulation element. Since the environment is an exception to encapsulation, it will not be provided in additional libraries, like specialized agents. Instead it will be part of the framework specification.

The following argument further justifies the decision:

- If the environment is treated as a regular object, the performance overhead will be prohibitive, especially in the case of multi-agent systems.

This decision does not mean that the environment will be modeled monolithically, as a single data structure. Modularity can still be achieved through composition of smaller data structures. This form of composition will depend on the kind of data structure of a particular environment implementation. Therefore, it is not included in the framework, which is on a more general level of abstraction.

In OOS, each object is responsible both for keeping its own state and defining its own behavior. They interact directly by input-output coupling relations, which are usually organized in a fixed structure. However, in computer games, as commonly happens in AOS, interaction between game elements is often determined by spatial proximity. That interaction may be direct or indirect, through the environment. If two elements are sensing and acting on the same piece of environment, they will have established an indirect causal relation among

PUC-Rio - Certificação Digital Nº 0821408/CA

their actions. That environment in most cases represents a physical space where these causality links happen by proximity of the physical location of the game elements. This justifies the high importance given to collision detection in the area of gaming and, more generally, to spatial algorithms and spatial indexing in AOS.

There are three main reasons why most games use the centralized physical environment approach. First, as already mentioned in section 2.1, it is important to group all rendered game elements in a specialized data structure to improve rendering performance (Sowizral 2000; Metello et al. 2007). Second, it makes it more natural to model actions that depend on spatial relationships. Lastly, object-oriented approaches usually offer little support for dynamic changes to the coupling structure of its objects, which is necessary in the case of moving objects that interact by proximity, as in computer games.

All these arguments suggest that the environment object should keep not only the spatial structure of the game but also the physical representation of its elements. This approach differs from traditional OOS by splitting the physical and behavioral states of game elements into two different objects. Physical states should be part of the environment and be deprived of their proactivity. Elements representing behaviors should be responsible for animating their physical counterparts. These elements shall be called *processes*.

Decision 4: There will be two types of simulation elements: environment and processes. The space and physical state of simulation elements will be implemented as parts of the environment, which is deprived of any proactive behavior. Any kind of behavior will be implemented in processes, which are responsible for providing behavior to all physical elements in the simulation.

The main arguments for this decision are the following:

- It allows the grouping of all physical game elements in a specialized and possibly spatially indexed data structure, which helps improve the performance of the rendering algorithms as well as of the spatial algorithms.

PUC-Rio - Certificação Digital Nº 0821408/CA

- Depriving the environment of proactivity does not impose any restrictions to the supported simulation models. If a particular model, for some reason, considers an environment that evolves in time, that environment should be divided in two parts: one physical state and one process that will interact with the physical state and implement its behavior.
- Both agents and other types of simulation objects can be modeled in this framework by splitting them into physical and behavioral parts. The physical part is modeled as part of the environment and the behavioral part is modeled as a process.

3.2.3
On the Interaction between Elements

Decision 4 stated that a simulation is composed of an environment and a set of processes. Therefore, the possible types of interaction between elements are inter-process interaction and interaction between a process and the environment.

Considering the case of inter-process interaction, *decision 2* defined that the simulation elements interact with each other as in OOS, by sending events to each other. This form of interaction allows the modular design of complex processes as a composition of interacting sub-processes as, for example, in the *coupled DEVS* formalism, described in section 2.2.1. The same form of interaction could also be used to implement the communication between cognitive agents, where the processes that model the behavior of the agents need to exchange messages.

The case of process-environment interaction needs further discussion because processes are allowed to access the internal state of the environment, as stated in *decision 3*.

Since the framework provides the notion of a global environment, the traditional modularity of OOS approaches becomes less characterized. In OOS, the global state is typically distributed across the objects, which have dependency relations only with their immediate neighbors in the coupling structure. Therefore, some special care is necessary to design the interaction between the environment and other elements in order to keep a reasonable degree of modularity and reuse.

PUC-Rio - Certificação Digital Nº 0821408/CA

PUC-Rio - Certificação Digital Nº 0821408/CA

In order to reuse a process in different simulations with different environments, this process must perceive those different environments in the same way, with the same set of possible states. One simple approach to accomplish this is the notion of *environment views*, which is similar to the concept of *interface* in object-oriented programming languages. An interface basically defines a type of object with a definite set of possible states. An object that implements that interface can be perceived as an object of that type. The same notion applies to environment views. Different environments providing the same view can be perceived in the same way. That helps reducing dependency relations, improving modularity and reuse of simulation models.

Up to this point, the discussion covered the topic of environment perception, concluding that external elements should have *read access* to the internal state of the environment. This raises the question of whether external elements should also have *write access*, which in turn inevitably leads to the problem of concurrency.

This problem also arises in AOS and, more generally, in the broader field of multi-agent systems (Michel et al 2009). Practically all OOS formalisms provide some mechanism to deal with it. It has even led to the creation of new formalisms, such as Parallel DEVS (Zeigler et al. 2000).

Since our approach adopts the main characteristics of OOS, the straightforward solution is to adopt a well-established concurrency mechanism from some OOS formalism. In this case, any simulation element that intends to change the environment state has to do it by sending events to it and not by direct writing. Modeling actions that alter the state of the environment as regular OOS events certainly makes the simulation framework more uniform, in the sense that state transitions are always propagated in the same way throughout the simulation elements, including the environment.

One might argue that this approach could lead to performance loss due to the fact that a process is forced to make a copy of the information it sends to the environment, in order to send it as an event. This is the same case as the performance problem discussed in *decision 3*, only in the opposite direction. However, we believe that this will not be performance-costly in most cases because of the semantics of sending information to the environment. A process sends an event to the environment when it acts on it, causing changes. It is reasonable to assume that a process will know exactly what it wants to change

when it decides to act on the environment. Therefore, it can send only the necessary information to perform the change. The problem that led to *decision* 3 is that, when the information is flowing in the opposite direction, the environment hardly knows precisely which information the process will really need. Therefore, in that case, it is better to let the process query the environment state.

Decision 5: Processes interact with each other by exchanging events, as in pure OOS. Processes also affect the environment by sending events to it. However, to observe the environment state, processes will directly access an environment view. The environment will provide a set of views, each one defined by a set of perceivable states.

The following arguments further support this decision:

- The propagation of state transitions in the network of simulation elements is uniformly done by events, as in pure OOS. Hence, many formal properties of OOS are incorporated, such as concurrency control.
- Accessing the internal state of the environment indirectly via *environment views* increases modularity. The same set of processes can be reused in different simulations with different environments, as long as the environments provide the necessary views.

PUC-Rio - Certificação Digital Nº 0821408/CA

3.2.4 The Process–Oriented Simulation Paradigm

The decisions taken in sections 3.2.2 and 3.2.3 lead to a *process-oriented simulation* (POS) paradigm, which is a hybrid paradigm with notions adopted both from OOS and AOS.

If we consider only the operational characteristics of the simulation elements, the POS paradigm is very close to OOS, with the exception of the environment read access method. However, this exception can be *abstracted* as if each read access query were composed of two regular events, one for the query and one for the answer. However, it should not be *implemented* that way for the

performance reasons discussed in section 3.2.2. This abstraction gives POS the possibility of inheriting many interesting formal properties from OOS, such as universality and closure under coupling/composition (Zeigler et al. 2000).

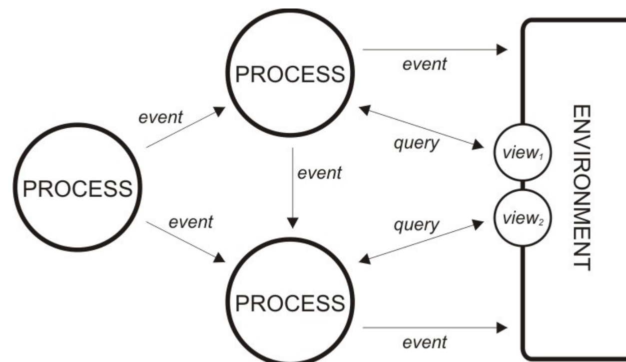

Figure 3.3 – Process-Oriented Simulation

If we consider the semantics of the simulation elements, the POS paradigm inherits the notion of environment from AOS. However, there is a conceptual difference between both. In AOS, the boundaries of the simulation elements are usually defined by the boundaries of entities in the real system they attempt to simulate. For example, in the popular case of the simulation of an insect colony, there is usually an agent for each insect. Likewise, in social simulations, there are agents representing people or institutions. In POS, the simulation elements are defined first by their nature – physical or behavior – then they are further divided according to their complexity in order to achieve modularity. For example, in an insect colony, there may be a single process responsible for implementing the behavior of all insects in the simulation. Likewise, in a social simulation, the behavior of a person could be divided into different parts, such as production, consumption and leisure, each one implemented by a different process.

These characteristics of POS aim clearly to isolate the physical representation of whatever is being simulated, while keeping traditional simulation properties. That makes POS, just like AOS, suitable for simulations with highly specialized forms of environment representation, such as those mentioned in section 2.1 and GIS-based spatial structures and databases (Gimblett 2002; Gonçalves et al. 2004).

PUC-Rio - Certificação Digital Nº 0821408/CA

3.2.5 Process Creation and Destruction

Predicting in advance what is going to happen in a simulation is usually hard. In fact, that is one of the reasons simulations exist. On top of that, simulations may involve sources of non-determinism such as coin flips or human interactions. That raises a relevant question on whether processes should be allowed to be created and destroyed during the execution of a simulation. If all simulation activity could be easily predicted in advance, all necessary processes could be instantiated at the beginning of the simulation and it would not be necessary to create or destroy them at execution time.

If we take for a fact that no process can be created at execution time, a reasonably complex simulation with many different possible outcomes could potentially produce one (or both) of the following situations:

- The appearance of complex processes, which can act in a number of different ways, according to the evolution path of the simulation. This would hurt modularity.

- A large number of smaller processes instantiated at the beginning of the simulation to handle every possible scenario that arises during execution. This potentially leads to huge inefficiencies, because it is not known in advance which processes will actually play some role in the simulation.

Allowing processes to be created at execution time makes it possible to avoid these situations. In fact, AOS-based toolkits typically allow the creation and destruction of agents at execution time. In opposition, many traditional OOS approaches, such as the basic DEVS, do not consider this kind of structural change during simulation execution. This limitation has been felt in a number of research works and has lead to extensions to some OOS formalisms to support variable object structures (Uhrmacher 2001).

In the context of discrete-event simulations, the creation of a process may be considered as an event. In fact, it can be abstracted as an instantaneous state

PUC-Rio - Certificação Digital Nº 0821408/CA

transition, where the process leaves the state of non-existence and assumes the initial state of its lifetime. Likewise, its destruction may also happen instantaneously as another event. This observation suggests a simple mechanism for process creation and destruction. Instead of sending events to other processes or to the environment, a process can output a special type of event causing the creation or destruction of another process. By analogy with execution threads, one can say that a process can *fork* other processes. In fact, processes can be seen as threads in a multithreaded programming environment. If we continue the analogy, a process that has forked another process is referred to as its *parent process*. Likewise, the forked process is called the *child process*.

The hierarchical structure of processes induced by the process forking model provides the additional benefit of allowing abstraction levels when reasoning about processes. For example, a workflow may be represented as a single parent process which forks a child process for each action that is executed in the workflow. Hence, the whole workflow may be seen as one single process or as a set of actions according to the desired abstraction level.

Decision 6: Processes may fork and destroy other processes by outputting a special type of event, which is part of the framework definition. The creation and destruction of processes happen instantaneously with respect to simulation time, just like regular events.

The following arguments further support this decision:

- Allowing a dynamic simulation structure helps keeping the simulation models modular and simple.
- Parental processes hierarchies allow reasoning about and designing processes in multiple abstraction levels.

3.3 The Process-DEVS Formalism for Process Modeling

The result of the discussion in section 3.2 was an abstract framework to model the dynamics of serious games. This section describes an instantiation of

PUC-Rio - Certificação Digital Nº 0821408/CA

the framework, called *Process-DEVS*, which extends the *DEVS* formalism (Zeigler 1972) to work with the process-oriented simulation paradigm. DEVS was chosen as basis for our framework because of its formal properties, such as universality and closure under composition (Zeigler et al. 2000).

3.3.1 Formal Model

We start the description of the formal model with the definition of an abstract *simulation element* or, simply, an *element*. Then, we define two classes of elements, *process* and *environment*. Finally, we introduce two specializations of processes, *input processes* and *output processes*, designed for interacting with external asynchronous entities.

An operational semantics of these concepts is described in section 3.3.2.

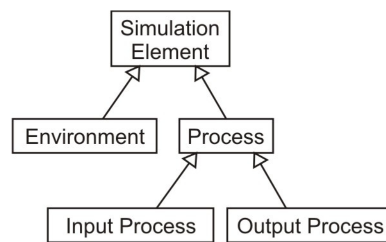

Figure 3.4 – The Hierarchy of Simulation Elements in Process-DEVS

Processes and environments have very similar behaviors. Therefore, defining how they are simulated in terms of the abstract notion of simulation elements leads to a more concise way of describing the simulation mechanisms.

Simulation Element

A *simulation element* is defined by a tuple of the form (an intuitive explanation of the components follows the formal definition):

$$\langle S, V, X, Y, E, P, \delta_{int}, \delta_{ext}, \lambda, \rho, ta \rangle$$

where

S is the set of possible *internal states*

PUC-Rio - Certificação Digital Nº 0821408/CA

$V = \{(V_i, \mu_i) \mid i=1, \ldots n\}$ is a set of *views* that provide external read access to the internal state of this element, where

V_i is the set of *view states* of the ith view

$\mu_i: S \rightarrow V_i$ is the *view mapping function* of the ith view

X is the set of *acceptable input events*

Y is the set of *possible output events*

E is a set of *environment view states*

P is the set of elements that this element can create and destroy

$\delta_{int}: S \times E \rightarrow S \cup \{finished\}$ is the *internal transition function*

$\delta_{ext}: Q \times E \times (X \cup \{finish\}) \rightarrow S \cup \{finished\}$

is the *external transition function*, where

$Q = \{(s, e) \mid s \in S, 0 \leq e \leq ta(s)\}$ is the *total state* set

e is the *time elapsed* since last transition

$\lambda: S \rightarrow 2^Y$ is the *output function*

$\rho: S \rightarrow 2^P \times 2^P$ is the *process structure transition* function

$ta: S \rightarrow [0, \infty]$ is the *time advance function*

The terms S, X, Y, δ_{int}, δ_{ext}, λ and ta have basically the same meaning as in the basic DEVS formalism, introduced in section 2.2.1. The set S defines all possible internal states an element may assume. The sets X and Y define all possible input and output events of the element, at any time in the simulation. These three sets have the exact same meaning as in DEVS.

The functions δ_{int} and δ_{ext} define all state transitions of the element. They are basically the same functions defined in the basic DEVS formalism, with some minor differences.

The function δ_{int} is the internal transition function. It is responsible for defining the *proactive behavior* of the element. This function is invoked by the simulator after $ta(s)$ units of time have passed since the last state transition, where s is the internal state that resulted in that last transition. The output of this function defines the next state of the element. The special output value *finished* means that this will be the last state transition of this element, which will cease to exist in the simulation from that time instant on. In order to compute the state transition, δ_{int} is

PUC-Rio - Certificação Digital Nº 0821408/CA

allowed to read the current state of the element, as well as the environment state, according to how this element perceives the environment.

The function δ_{ext} is the external transition function. It is responsible for defining the *reactive behavior* of the element. This function is invoked by the simulator whenever the element receives an event from another element. The output of this function is handled exactly in the same way as in δ_{int}. However, its input is quite different. It is allowed to read the event that the element is receiving. If it receives the special event *finish*, it means that this will be the last state transition of this element, which will cease to exist in the simulation. This last transition function call allows the process to finish in a friendly way, releasing resources and sending events to inform other processes. The external transition function is also allowed to access the environment state, the current element state and the time elapsed since the last transition, which is not necessary in δ_{int} because it can be computed by *ta(s)*.

The terms λ and *ta* have the exact same meaning as in DEVS. $\lambda(s)$ defines which events are output by the element after any state transition is performed. *ta(s)* defines the time delay the simulator will wait to call δ_{int} again, if no events are received until then.

The terms *V* and *E* define the way an element may access the internal state of another element. The set *V* defines the views that this element provides, through which other elements can access a particular view of its internal state. Note that the internal state is not accessed directly. Instead, external elements are only allowed to access view states of the views defined by *V*. Those view states are determined by the view mapping functions applied to the current internal state. The set *E* defines the environment view states in which the element can perceive its environment. The formalism only allows an element to access the state of exactly one other element. This follows because there will be only one environment in a simulation, and this is the only element that will have its internal state accessed through its views.

Finally, the terms *P* and ρ define the mechanisms for dynamic creation and destruction of elements. The set *P* contains all elements that this element can create and destroy. The function ρ outputs two sets of elements, one for the

PUC-Rio - Certificação Digital Nº 0821408/CA

elements that are created and one for those that are destroyed, whenever a state transition has been performed.

As defined here, simulation elements do not have input and output ports, as in the DEVS with ports formalism (Zeigler et al. 2000), described in section 2.2.1. This is merely for notational simplicity, since this decision is unimportant at a conceptual level. It is understood here that the main benefit of ports for games is to improve performance by allowing a more efficient event routing method. Since performance is extremely important for games, the model should be easily extendable to embrace port support, even though it is not relevant to the discussion on an abstract level. Ports could be easily added to the framework with minor changes in this notation: simply by representing inputs and outputs by pairs *(port*, *event)*, exactly as in DEVS with ports.

Environment

An *environment* is an element $\langle S, V, X, Y, E, P, \delta_{int}, \delta_{ext}, \lambda, \rho, ta\rangle$ that satisfies the following constraints:

(1) $E = \{nil\}$

(2) $P = \emptyset \wedge (\forall s \in S)(\rho(s) = (\emptyset, \emptyset))$

(3) $(\forall s \in S)\ (\forall e_1 \in \Re^+)\ (\forall e_2 \in \Re^+)\ (\forall x \in (X \cup \{finish\}))$

$$(\delta_{ext}((s,e_1),nil,x) = \delta_{ext}((s,e_2),nil,x))$$

(4) $(\forall s \in S)\ (\forall e \in \Re^+)\ (\forall x \in (X \cup \{finish\}))$

$$(\delta_{int}(s,nil) \neq finished \wedge \delta_{ext}((s,e),nil,x) \neq finished)$$

(5) $(\forall s \in S)(ta(s) \in \{0, \infty\})$

Constraint (1) just states that the environment does not directly access the internal state of any other element. Constraint (2) states that the environment does not alter the simulation structure. Constraint (3) states that state transitions do not depend on the elapsed time since the last transition. Constraint (4) guarantees that the environment never finishes. Constraint (5) deprives the environment of *proactive* behavior, which means that its state does not change with the flow of time, if no events are received. The *ta* function is allowed to output the value 0 in order to allow *transient states* (Zeigler et al. 2000). Transient states are states that do not have duration. When reached, they are immediately changed again. They are commonly used as intermediate states in a state transition to produce different

PUC-Rio - Certificação Digital Nº 0821408/CA

outputs according to the previous state of a given system. Function *ta* is also allowed to output the special value ∞, meaning that its internal transition function will not be invoked at least until the next received event, when the external transition function will be invoked and *ta* will be evaluated again.

Even though the environment does not act proactively, it is still allowed to output events in response to state changes, which are always caused by the arrival of another event. The purpose of these output events is to alert processes about state changes in the environment, which is analogous to the *observer pattern* in object-oriented design patterns (Gamma et al. 1995). If we deprived the environment of the ability to send events to processes, any process that needs to respond to changes in the environment would have to check it with a minimum frequency, which would lead to inefficiencies.

For ease of notation, the environment is defined by the simplified structure $E = \langle S, V, X, Y, \delta_{int}, \delta_{ext}, \lambda, ta \rangle$, subject to constraints (3) and (5), representing the element $\langle S, V, X, Y, \{nil\}, \varnothing, \delta_{int}, \delta_{ext}, \lambda, \rho, ta \rangle$, where $(\forall s \in S)(\rho(s) = (\varnothing, \varnothing))$.

PUC-Rio - Certificação Digital Nº 0821408/CA

Processes

A process is a simulation element $\langle S, V, X, Y, E, P, \delta_{int}, \delta_{ext}, \lambda, \rho, ta \rangle$ such that $V = \varnothing$. This means that the internal state of a process is not directly accessible by any other element. For ease of notation, a process is defined by the structure $\langle S, X, Y, E, P, \delta_{int}, \delta_{ext}, \lambda, \rho, ta \rangle$, representing the simulation element $\langle S, \varnothing, X, Y, E, P, \delta_{int}, \delta_{ext}, \lambda, \rho, ta \rangle$.

Intuitively, a process represents an activity carried out in a simulation. Processes may be created and destroyed dynamically during a simulation. In order to reason about processes in time, it is possible to determine their *start times* and *finish times*. Process creation and destruction are usually defined by the function ρ, which is invoked right after every state transition, and before the simulation time is advanced any further. Therefore, the start and finish times of processes is determined by the time instants of the state transitions that triggered their creation and destruction. There is still another way to destroy a process, which is by suicide. Whenever a state transition leads to the special state *finished*, the finish time is naturally defined by the time instant of that transition. Having well-defined

start and finish times allows some interesting analytical properties such as, for example, the representation of process execution histories in a very similar way as in Sowa's discrete event process model (Sowa 2000).

Whenever a process creates another process, it is said that the *parent process*, which is the creator process, has *forked* a *child process*, which is the created process. Process forking, besides providing simulations with structural dynamism, also provides abstraction level capabilities to process modeling. The capability of process forking is considered a form of abstraction and modularity of process modeling in the sense that a process may delegate some of its sub-tasks to its children. Hence, modularity and abstraction is achieved in a different way, as compared with the coupled DEVS formalism (Zeigler et al. 2000), described in section 2.2.1.

I/O Processes

PUC-Rio - Certificação Digital Nº 0821408/CA

Input processes and *output processes* are processes dedicated to manage the communication with asynchronous entities external to the simulation. This communication is modeled as exchange of events between a process and an external entity. Hence, any process can communicate with some external entity in the same way it communicates with other processes. The input and output processes act as one-way channels. They receive events from the sender side and store those events in their internal state until the receiving side requests the events to be flushed. Therefore, I/O processes act as streams of events.

In order to represent event streams, it is necessary to use lists, instead of sets. The following notation is used to represent lists: S^* is the set of all possible lists formed with elements of S; $[e_1, e_2, \ldots , e_n]$ represents the list formed by the elements $e_1, e_2, \ldots , e_n$, in that order; $[]$ represents the empty list; $[head \mid tail]$ represents a list that has the element *head* as its first element, followed by all the elements in the list *tail*, in the same order. For example, $[e_1 \mid [e_2, e_3]]$ represents the list $[e_1, e_2, e_3]$.

An *input process* receives events from external entities, where the events are taken from a set I, and sends them to other processes. An input process over I is formally defined as $p_{in} = \langle S, X, Y, E, P, \delta_{int}, \delta_{ext}, \lambda, \rho, ta\rangle$, where

$S = I^* \times (I \cup \{nil\})$

$X = I$

$Y = I$

$E = \{nil\}$

$P = \varnothing$

$\delta_{int}((list, out), nil) = (list, nil)$ if $list = []$

$= ([e_1, \ldots, e_{n-1}], e_n)$ if $list = [e_1, \ldots, e_n] \neq []$

$\delta_{ext}(((list, out), e), nil, x) = ([x \mid list], nil)$ if $x \neq finish$

$= (list, nil)$ if $x = finish$

$\lambda((list, out)) = \{out\}$ if $out \neq nil$

$= \varnothing$ if $out = nil$

$\rho(s) = (\varnothing, \varnothing)$

$ta((list, out)) = \infty$ if $list = []$

$= 0,$ if $list \neq []$

PUC-Rio - Certificação Digital Nº 0821408/CA

When an event is received by the input process, it is stored in the internal state. As soon as possible, the input process outputs the events stored in its internal state to other processes in the same way as any other simulation element.

An *output process* receives events from other processes, taken from a set O of events, and sends them to external entities. An output process over O is formally defined as $p_{out} = \langle S, X, Y, E, P, \delta_{int}, \delta_{ext}, \lambda, \rho, ta \rangle$, where

$S = O^*$

$X = O$

$Y = \varnothing$

$E = \{nil\}$

$P = \varnothing$

$\delta_{int}(list, nil) = list$

$\delta_{ext}((list, e), nil, x) = [x \mid list]$ if $x \neq finish$

$= list$ if $x = finish$

$\lambda(s) = \varnothing$

$\rho(s) = (\varnothing, \varnothing)$

$ta(s) = \infty$

The output processes store the events they receive in a list. This list is used to generate a stream of events for entities which are external to the simulation. This will be formally defined in section 3.3.2.

Simulation

Environments and processes are parts of the broader notion of *simulation*. A simulation is basically a container of simulation elements with some additional information. Besides the environment and a set of processes, it also defines the *event coupling structure* between these elements and a *view-process coupling map*. The event coupling structure defines the recipients of events generated by any element, while the view-process coupling map defines which environment view is accessible to each process.

In the formal definition of a simulation, the operator "·" (dot) is used to access a property of a given structure. Therefore, "$S{\cdot}p$" should be interpreted as "property p of structure S".

A *simulation* is formally defined as

$$SIM = \langle SE, s_0, P_0, cs, v_{map}, \tau \rangle$$

where

SE is the set of all simulation elements, which must include a single environment element. We define the following subsets of SE and single out the environment in SE:

P_{in} is the set of input processes in SE

P_{out} is the set of output processes in SE

$\varepsilon = \langle S, V, X, Y, \delta_{int}, \delta_{ext}, \lambda, ta \rangle$ is the (only) environment in SE

s_0: $SE \rightarrow \bigcup_{e \in SE} e{\cdot}S$, where $s_0(elem) \in elem{\cdot}S$, is the *initial state* map

P_0 is the initial set of running processes, which must be a subset of the set of processes in SE

cs: $SE \times SE \rightarrow \{true, false\}$ is the *event coupling structure*

v_{map}: $SE - \{\varepsilon\} \rightarrow \varepsilon{\cdot}V$ is the *view-process coupling map*

PUC-Rio - Certificação Digital Nº 0821408/CA

τ: $2^{FC} - \{\varnothing\} \rightarrow FC$ is the *tiebreak function*, where $FC = \{itf_call(e) \mid e \in SE\} \cup \{etf_call(e, evt) \mid e \in SE \wedge evt \in (e \cdot X \cup \{finish\})\}$ is the set of all possible transition function calls, which is explained below

During the lifetime of a simulation run, a number of elements are simultaneously simulated. An element can be either a process or the environment. The special predicate *isEnvironment(e)* will be used when it is necessary to differentiate between both types. For any given element *e*, *isEnvironment(e)* implies that the constraints of the environment definition apply to *e*. Likewise, $\neg$*isEnvironment(e)* implies that the constraints of the process definition apply to *e*.

The set *SE* contains, besides the environment, all processes that can be executed in a simulation run. It is possible that some of the processes in *SE* are never started, depending on the course of the simulation run. Function s_0 maps each element into its initial internal state. The set P_0 defines the processes that are started exactly at the simulation start time. Each simulation is allowed to have only one environment ε.

When an element outputs an event, the coupling structure *cs* determines which elements are receiving it. $cs(E_{send}, E_{receive}) = true$ means that the element $E_{receive}$ should receive events from the element E_{send}. The view-process coupling map serves a similar purpose with respect to the capabilities of processes to query the internal state of the environment. $v_{map}(p)$ returns the environment view that the process *p* is allowed to access.

The tiebreak function τ defines the order in which concurrent events are processed. State changes in simulation elements are caused either by the receiving of an event from another element or by the expiration of the time returned by the *time advance function (ta)* of the element in its last state transition. In the first case, the *external transition function* (δ_{ext}) of the element is called, while in the second case the *internal transition function* (δ_{int}) is called. In both cases, the call to the transition function returns the next state of the element. The set *FC* in the definition of the tiebreak function τ contains all possible calls to any transition function of any simulation element, where *itf_call(e)* and *etf_call(e, evt)* represent, respectively, calls to the internal and external transition functions of *e* with their

PUC-Rio - Certificação Digital Nº 0821408/CA

respective parameters. Given any set of transition function calls, the tiebreak function defines a total ordering over it.

Let $SIM = \langle SE, s_0, P_0, cs, v_{map}, \tau\rangle$ be a simulation. Recall that

P_{in} is the set of input processes in SE

P_{out} is the set of output processes in SE

$\varepsilon = \langle S, V, X, Y, \delta_{int}, \delta_{ext}, \lambda, ta\rangle$ is the (only) environment in SE

The simulation SIM must obey the following constraints:

(1) $(P_0 \subseteq SE)$

(2) $(\forall e \in SE)(e{\cdot}P \subseteq (SE - \{e, \varepsilon\}))$

(3) $isEnvironment(\varepsilon)$

(4) $(\forall e \in SE)(isEnvironment(e) \Rightarrow e = \varepsilon)$

(5) $(\forall e_{from} \in SE)\ (\forall e_{to} \in SE)(cs(e_{from}, e_{to}) = true \Rightarrow e_{from}{\cdot}Y \subseteq e_{to}{\cdot}X)$

(6) $(\forall p_{in} \in P_{in})(\forall p \in SE)(cs(p, p_{in}) = false)$

(7) $(\forall p_{out} \in P_{out})(\forall p \in SE)(cs(p_{out}, p) = false)$

(8) $(\forall p \in SE)(v_{map}(p) = (V_i, \mu_i) \Rightarrow V_i \subseteq p{\cdot}E)$

(9) $(\forall S \in 2^{FC})(\forall c \in FC)(\tau(S) = c) \Rightarrow$

$$(c \in S \wedge (\forall S' \in 2^{FC})((S' \neq \varnothing \wedge S' \subset S) \Rightarrow \tau(S') = c))$$

Constraint (1) assures that the initial processes are all simulation elements of SE (the I/O processes and the environment are simulation elements in SE, by definition). Constraint (2) assures that all dynamically-created elements are processes of this simulation. Constraints (3) and (4) determine that there must be only one environment in a simulation. Constraint (5) assures that any element that receives an event from another element will know how to handle it. Constraints (6) and (7) state that no process can send events to input processes or receive events from output processes. The I/O processes are one-way event streams. Constraint (8) assures that all processes will receive an understandable state when they query their environment view. Constraint (9) assures that the tiebreak function τ represents, in fact, a total ordering over the set of all possible transition function calls.

The basic working model of a simulation is illustrated in Figure 3.5. Process P_1 has two child processes P_{11} and P_{12}. All of them, including the parent P_1 can send events to the environment and to other processes. Processes P_{out}, P_{in1} and P_{in2}

PUC-Rio - Certificação Digital Nº 0821408/CA

are *I/O processes*. They are responsible for the communication between human players and the rest of the simulation. Through environment views V_1 and V_2, the processes P_{11} and P_{12} observe the state of the environment.

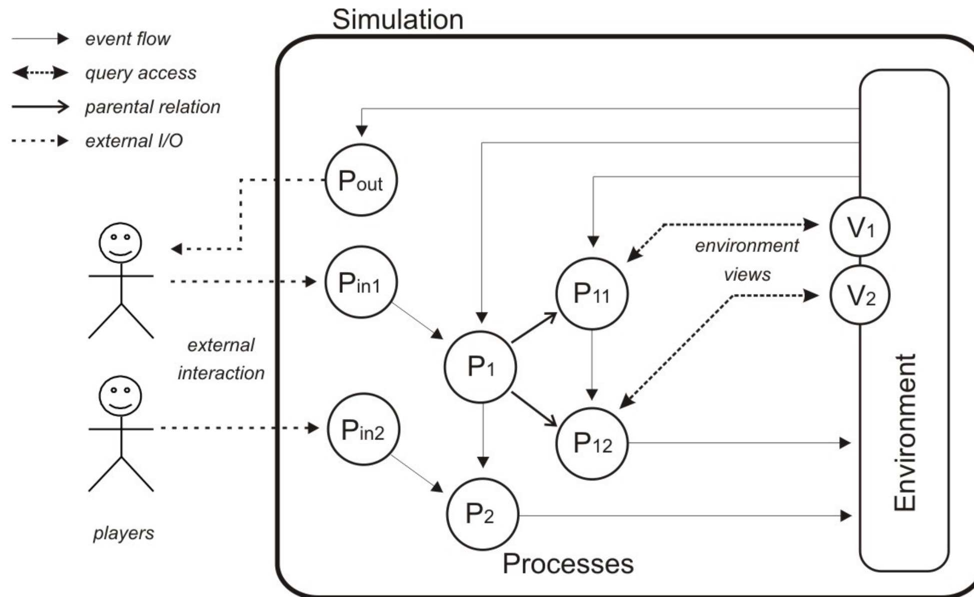

Figure 3.5. The Simulation Model

The two environment views act as interfaces, providing a mechanism for processes to get information about the internal state of the environment at any time. The idea is to allow processes to access the environment through simplified views. Hence, modularity can be increased because processes need not understand the full environment state. The same process can work on any environment that provides the view used by that process.

3.3.2 Operational Semantics

This section presents the operational semantics of simulations in Process-DEVS. Before the introduction of the model, it is necessary to define some basic notation that will be used throughout the rest of this thesis.

As in the previous section, the operator "·" (dot) is used to represent a property of a given object. Therefore, "$O \cdot p$" should be interpreted as "property *p* of object *O*". Lists are represented in the form $[e_1, e_2, \ldots, e_n]$, where *[]* is the empty list.

PUC-Rio - Certificação Digital Nº 0821408/CA

The operator $\leftarrow$ is used in expressions of the form $f = g \leftarrow (d,r)$, where f and g are functions with the same domain and range sets and $f(x)$ is equal to $g(x)$ for all the values of x, with the exception of the value d, for which $f(d) = r$.

The abstract notion of *simulation element* will help simplify the definition of the operational semantics because most of the time processes and the environment are treated in the same way. Whenever it is necessary to distinguish them, the special predicate *isEnvironment(e)* will be used, with the same semantics as in the definition of simulation, in section 3.3.1.

Each element has a definite *start time* and a definite *finish time*. No element can start before the simulation run starts and no element continues to execute after it has finished. The environment is always in execution during the simulation run and it does not make sense for it to have start and finish times different from the simulation start and finish times.

Let $SIM = \langle SE, s_0, P_0, cs, v_{map}, \tau \rangle$ be a simulation.

The execution of *SIM* is determined by a sequence of *simulation states* in time. The foundations of discrete-event based simulation require that all changes to the simulation state be instantaneous. A simulation state change is caused either by the creation, destruction, or state transition of any of its simulation elements.

A simulation execution state of *SIM* is defined as

$$SS = \langle t, E_{active}, E_{state}, E_{last_t}, EQ \rangle$$

where

t is the current simulation time

$E_{active} \subseteq SE$ is the set of active simulation elements

$E_{state}: SIM \cdot SE \rightarrow \bigcup_{e \in \mathrm{SIM} \cdot SE} \mathrm{e \cdot S} \cup \{finished\}$

where $E_{state}(e) \in (e \cdot S \cup \{finished\})$ is a function that maps each element $e \in SIM \cdot SE$ into the current internal state of that element

$E_{last_t}: SIM \cdot SE \rightarrow \Re^{+}$ is the *last transition time map*

EQ is the *event queue*, which stores the next scheduled events

PUC-Rio - Certificação Digital Nº 0821408/CA

The initial state of a simulation *SIM* is

$$\langle 0, P_0, s_0, lt_0, eq_0 \rangle$$

where

$(\forall e \in SE)(lt_0(e) = 0)$

$eq_0 = \{(t_{init}, itf_call(e)) \mid e \in P_0\}$

The initial simulation time is *0*. It is increased as the simulation advances. Each simulation state *ss* stores the current simulation time *t*. Naturally, a simulation run does not produce a different simulation state for each possible time instant. Instead, the state *ss* may jump directly to another state *ss'*, with current time $t+\Delta t$, provided that no event is scheduled to happen in that time interval. The simulation state also includes the current *execution state* of all simulation elements, given by E_{active}, E_{state} and E_{last_t}. The set E_{active} contains all elements that are currently active. The functions E_{state} and E_{last_t} give the internal state and the timestamp of the last state transition of each element. Finally, the simulation state keeps an *event queue*, which stores all events currently scheduled to happen.

The event queue is the main component of most discrete-event simulators. In our case, it contains scheduled calls to the transition functions of simulation elements. Each element of the queue assumes one of the two forms *(ts, itf_call(e))*, for internal transition function calls, or *(ts, etf_call(e, evt))*, for external transition function calls, where *ts* is the time instant of the scheduled call, *e* is the simulation element and *evt* is the event that is passed as parameter in the case of external transition function calls.

All transition function calls are serialized with respect to time. If two calls are scheduled to happen at the same time, the tiebreak function of the simulation defines the order in which they are called. Given an event queue *EQ*, the next transition function to be called is given by the *next_call* operation:

(1) $next_call(EQ) = (\infty, nil) \quad if\ EQ = \varnothing$

$\quad = (t, c) \quad if\ EQ \neq \varnothing$

where $(t, c) \in EQ \wedge (\forall (t', c') \in EQ)(t' \geq t) \wedge (t, c) = \tau(\{(t, c'') \in EQ\})$

This ensures that the next call always has the least timestamp in *EQ*. If there is more than one call with that timestamp, the tiebreak function defines

PUC-Rio - Certificação Digital Nº 0821408/CA

which one is the next call. If EQ is empty, it returns a *nil* call with an infinite timestamp.

The next operators are defined in order to provide means of manipulating the event queue. Each operator returns another event queue as the result of the operation.

(2) $remove_calls(EQ, e) = \{(ts, c) \in EQ \mid c \cdot e \neq e\}$

This operator removes all transition functions calls of simulation element *e*.

(3) $schedule_itf_call(EQ, e, ts) =$

$(EQ - \{(t, c) \in EQ \mid c = itf_call(e)\}) \cup \{(ts, itf_call(e))\}$

This operator schedules an internal transition function call for simulation element *e* at *ts*, replacing all other calls to that function in *EQ*.

(4) $send_events(EQ, Events, E_{to}, ts) =$

$EQ \cup \{(ts, etf_call(e_{to}, evt)) \mid e_{to} \in E_{to} \wedge evt \in Events\}$

This operator schedules calls to external transition functions generated by the act of sending a set of events to a set of simulation elements. That means scheduling calls to all receiving elements, one for each event.

(5) $destroy(EQ, e, ts) =$

$send_events(remove_calls(EQ, e), \{finish\}, \{e\}, ts)$

This operator performs the changes in *EQ* when a simulation element is to be destroyed. It removes all calls to *e* and sends a *finish* event to it. This is done so that, when receiving the *finish* event, the process has a chance of releasing resources and informing others of its destruction.

(6) $create_destroy_elements(EQ, E_{create}, E_{destroy}, ts) =$

$$schedule_itf_call(\ldots schedule_itf_call(DTQ, ec_1, ts) \ldots , ec_n, ts)$$

where

$DTQ = destroy(\ldots destroy(destroy(EQ, ed_1, ts), ed_2, ts) \ldots , ed_m, ts)$

$E_{create} = \{ec_1, \ldots , ec_n\}$

PUC-Rio - Certificação Digital Nº 0821408/CA

$E_{destroy} = \{ed_1, \dots, ed_m\}$

This operator performs the changes in EQ relative to the creation of the elements in E_{create} and the destruction of those in $E_{destroy}$. When creating an element, it is only necessary to schedule an initial internal transition function call at the time the element is created.

(7) $schedule_transition_events(EQ, e, s_{next}, ts, Events, E_{to}, E_{create}, E_{destroy}) = schedule_itf_call(send_events(create_destroy_elements(EQ, E_{create}, E_{destroy}, ts), Events, E_{to}, ts), e, ts + e{\cdot}ta(s_{next}))$

This operator performs all changes in *EQ* generated by a state transition of a simulation element. First, it creates and destroys the elements defined by E_{create} and $E_{destroy}$. Then, it propagates the events in the set Events to the processes in E_{to}. Finally, it schedules the next internal transition function of *e*.

Now that the operations on the event queue are defined, we can define the operators to manipulate the simulation state. The *element_state_transition* operator defines how the simulation state is changed in the case of a state transition of an element.

(8) *If* $s_{next} \neq finished$:

$element_state_transition(SS, e, s_{next}, ts) =$

$$\langle ts, SS{\cdot}E_{active} \cup E_{create}, SS{\cdot}E_{state} \leftarrow (e, s_{next}), SS{\cdot}E_{last_t} \leftarrow (e, ts), schedule_transition_events(SS{\cdot}EQ, e, s_{next}, ts, \mathrm{E}vents, \{p_{to} \in SIM{\cdot}SE \mid SIM{\cdot}cs(p, p_{to}) = true\} \cap SS{\cdot}E_{active}, E_{create}, E_{destroy})\rangle$$

where

$Events = e{\cdot}\lambda(s_{next})$

$(E_{create}, E_{destroy}) = e{\cdot}\rho(s_{next})$

This operator produces a new simulation state, after a state transition of element *e* to the state s_{next}, at the time instant *ts*. The set of elements created by the transition is added to the set of currently active elements. However, the set of destroyed elements is not subtracted yet, since they still need to receive and treat the special event *finish*, before they are deactivated. The

PUC-Rio - Certificação Digital Nº 0821408/CA

new state s_{next} is assigned as the new internal state of *e* and *ts* becomes the timestamp of its last transition. Finally, it is only necessary to update the event queue with the effects of this state change by invoking the proper operator.

(9) If $s_{next} = finished$:

$element_state_transition(SS, e, s_{next}, ts) =$

$\langle ts, SS{\cdot}E_{active} - \{e\}, SS{\cdot}E_{state} \leftarrow (e, finished), SS{\cdot}E_{last_t} \leftarrow (e, ts),$

$remove_calls(EQ, e)\rangle$

This operator computes the new simulation state when an element performs a transition to the special state *finished*. This operation is relatively simple and consists basically of removing all execution information about the element *e*.

The following operators perform transition function calls on elements of the simulation:

(10) $remove_call(SS, call) = \langle SS{\cdot}t, SS{\cdot}E_{active}, SS{\cdot}E_{state}, SS{\cdot}E_{last_t}, SS{\cdot}EQ - \{call\}\rangle$

This simply removes a transition function call from the event queue *EQ*.

(11) $process_call(SS, ts, itf_call(e)) =$

$element_state_transition(remove_call(SS, (ts, itf_call(e))), e, s_{next}, ts)$

where

$s_{next} = e{\cdot}\delta_{\text{int}}(SS{\cdot}E_{state}(e), view)$

$view = SIM{\cdot}v_{map}(e){\cdot}\mu(SS{\cdot}E_{state}(SIM{\cdot}\varepsilon))$

This operator executes an internal transition function call on element *e*, at time *ts*. It first removes the scheduled call from the event queue. Then, it computes the state change caused by the function call.

(12) If $evt \neq finish$:

$process_call(SS, ts, etf_call(e, evt)) =$

$element_state_transition(remove_call(SS, (ts, etf_call(e, evt))), e, s_{next}, ts)$

where

PUC-Rio - Certificação Digital Nº 0821408/CA

$s_{next} = e \cdot \delta_{ext}((SS \cdot E_{state}(e), ts - SS \cdot E_{last_t}(e)), view, evt)$

$view = SIM \cdot v_{map}(e) \cdot \mu(SS \cdot E_{state}(SIM \cdot \varepsilon))$

This operator executes an external transition function call on element *e*, at time *ts*.

(13) If $evt = finish$:

$process_call(SS, ts, etf_call(e, evt)) =$

$element_state_transition(element_state_transition(remove_call(SS, (ts, etf_call(e, finish))), e, s_{next}, ts), e, finished, ts)$

where

$s_{next} = e \cdot \delta_{ext}((SS \cdot E_{state}(e), ts - SS \cdot E_{last_t}(e)), view, finish)$

$view = SIM \cdot v_{map}(e) \cdot \mu(SS \cdot E_{state}(SIM \cdot \varepsilon))$

This operator executes an external transition function call on element *e*, at time *ts*, when the special event *finish* is received by *e*.

PUC-Rio - Certificação Digital Nº 0821408/CA

The basic procedure for computing how the simulation state changes with time consists basically of retrieving transition function calls from the event queue and executing them in the right order. To determine the simulation state at simulation time *t*, it is necessary to execute all state transitions scheduled to happen between the current time and *t*. The *advance* function provides a recursive procedure for advancing the simulation state.

(14) $advance(SS, \Delta t) =$

$\langle SS \cdot t + \Delta t, SS \cdot E_{active}, SS \cdot E_{state}, SS \cdot E_{last_t}, SS \cdot EQ \rangle$, if $nc \cdot ts > SS \cdot t + \Delta t$

$advance(process_call(SS, nc), \Delta t - (nc \cdot ts - SS \cdot t))$, if $nc \cdot ts \leq SS \cdot t + \Delta t$

where

$nc = next_call(SS \cdot EQ)$

The simulation, as defined so far, does not interact with any external entities. The *advance* operator is responsible for updating the simulation execution state considering solely the internal dynamics of the simulation. In order to make the simulation interactive, it is also necessary to describe how the simulation state

changes when receiving input, as well as when generating output to some external entity.

All input and output are handled by the input processes and output processes. They are part of the simulation definition. Each input process receives events from an external entity and stores them in its internal state. As soon as possible, it transmits those events to their recipients. The output processes work in the opposite direction. They receive events during the simulation advance and store them in their internal states. When the simulator decides to flush the output events, the internal states of the output processes are read and cleared. The simulation inputs and outputs are represented as lists of the form

$$[(e_1, p_1), (e_2, p_2), \ldots, (e_n, p_n)]$$

where e_i is an event and p_i is its corresponding I/O process, as defined in section 3.3.1. In the case of processing an input, the simulation state is changed as defined by the *flush_input* operator:

(15) $flush_input(SS, Input) =$

$process_call(\ldots process_call(SS, SS{\cdot}t, etf_call(p_1,e_1))\ldots, SS{\cdot}t, etf_call(p_n,e_n))$

where

$Input = [(e_1, p_1), \ldots, (e_n, p_n)]$

$\{p_1, \ldots, p_n\} \subseteq (SIM{\cdot}P_{in} \cap SS{\cdot}E_{active})$

This operator basically generates one external function call on the corresponding *input process* for each received event.

In the case of processing the output generated by the simulation, it is necessary to read the information stored in the *output processes* and clear them afterwards, so that the same information is not read again in the next output:

(16) $read_output(SS) = CONCAT(SS{\cdot}E_{state}(p_1), SS{\cdot}E_{state}(p_2), \ldots, SS{\cdot}E_{state}(p_n))$

where

$CONCAT(l_1, l_2, \ldots, l_n)$ is the concatenation of lists $l_1, l_2, \ldots, l_n$

$\{p_1, p_2, \ldots, p_n\} = SIM{\cdot}P_{out} \cap SS{\cdot}E_{active}$

This operator reads all information stored in the output processes. Note that the state of an output process is a list of events. Therefore, the elements from $SS{\cdot}E_{state}(p)$ can be concatenated directly.

PUC-Rio - Certificação Digital Nº 0821408/CA

(17) *clear_output_processes(SS)* =

$$element_state_transition(\ldots element_state_transition(SS, p_1, [\,], SS{\cdot}t) \ldots , p_n, [\,], SS{\cdot}t)$$

where

$$\{p_1, p_2, \ldots , p_n\} = SIM{\cdot}P_{out} \cap SS{\cdot}E_{active}$$

This operator clears all information stored in the output processes by forcing a transition to the ∅ state.

The *flush_io* function consolidates all the input and output operations. It receives the current simulation state and an input set, and outputs the next simulation state and the output set.

(18) *flush_io(SS, Input)* =

$$(flush_input(clear_output_processes(SS), Input), read_output(SS))$$

The *advance* function describes how the simulation state is changed in time considering only the internal simulation mechanisms. The *flush_io* function describes how it changes when the communication with external entities is synchronized, without changing the simulation time. A full simulation run with external communication is described by a sequence of interleaved calls to these two functions, depending on when the external messages were exchanged. There are several ways to define how to interleave simulation time advance with external communication synchronization. This definition of the Process-DEVS operational semantics does not restrict implementations in that sense. Some examples of how to implement different interleaving mechanisms are discussed in section 5.5.

3.4 Summary

In order to design a framework for modeling and simulation in serious games, section 3.2 discussed the identified requirements. That discussion led to the conception of the *process-oriented simulation (POS)* paradigm, which helped fulfilling the following requirements:

PUC-Rio - Certificação Digital Nº 0821408/CA

- Integration of different formalisms: Modeling processes in a similar way as objects in OOS and using the discrete-event paradigm for modeling changes in time has been shown to be a good way of integrating different simulation formalisms (Praehofer et al. 1993; Vangheluwe 2000; Himmelspach and Uhrmacher 2007).

- Game-like user experience: Modeling an environment with environment views allows one to represent the internals of the environment in specialized data structures, such as those used by game engines for increasing realism in audiovisual media, as discussed in section 2.1.

- Modularity: The separation between state and behavior imposed by POS, the use of environment views and the fact that processes are modeled as objects in OOS contribute to increase the modularity and composition capabilities of the dynamic models in serious games.

Even though the requirements originated from the domain of serious games that simulate real situations, the decisions do not contain any specific semantics of this domain. Therefore, it is quite possible that the decisions that resulted from the discussion also apply to other simulation domains, especially those that require highly specialized data structures for the environment and those that involve processes of different nature interfering with each other.

The abstract framework was instantiated as the Process-DEVS modeling and simulation formalism, formally presented in section 3.3.1 as an extension of the original DEVS [Zeigler 2000]. Finally, section 3.3.2 formally defined the operational semantics of Process-DEVS.

PUC-Rio - Certificação Digital Nº 0821408/CA