2
Fundamentals

This section briefly describes a number of techniques and tools developed across different fields that are potentially interesting to serious games, by looking at the concepts of each major requirement.

Apart from those developed for the computer games field, all techniques and tools are somehow related to representing, analyzing, generating and simulating dynamic processes.

2.1
Computer Games

Existing computer games techniques is the first obvious place to look at. Unfortunately, since the gaming industry is so big and competitive, most companies usually keep their top technologies secret and only allow them to be published after becoming relatively obsolete and therefore losing most of their market value.

Over the last decades, one of the problems that received most attention in gaming is undoubtedly real-time rendering. With huge investments, modern graphic cards have been developed specifically for this purpose. Along with that hardware, software representations for virtual worlds also received considerable attention from Computer Science researchers, which led to the development of highly specialized data structures such as *scene graphs* (Strauss and Carey 1992).

Although interesting and challenging, real-time rendering is out of the scope of this thesis for its complexity. However, the following observations are important in the context of this thesis:

- In order to make full use of 3D gaming visual resources, it is necessary to store the visible 3D objects in highly specialized data structures.

- Those specialized data structures, such as scene graphs, are oriented towards rendering, not modeling (Sowizral 2000). Therefore, if traditional simulation techniques are to be integrated in a serious game architecture with modern game rendering techniques, it is necessary to find a way in which the simulation models act on those specialized data structures.

Since this thesis focuses on modeling and simulation, a special attention will be given on how games usually implement their dynamics. A closer look on the different kinds of *game loops* may help in that task.

2.1.1 Game Loops

Most computer games are inherently real time interactive applications. Their execution must be synchronized with the real time flow. Sometimes there is a need for accelerating the pace of a game. For example, in a training game that simulates an emergency situation that may last for days, the simulation should obviously not take the same amount of time. Periods requiring no decision making should be fast-forwarded. However, in these cases, the game also requires synchronization with the real time flow, only at a different rate in each game stage.

Real time applications consist, from the functional point of view, of three tasks being executed concurrently. First, they must continuously check for player input and process their commands accordingly. Second, the state of the world needs to be continuously updated. Finally, they must present the resulting world state to the player(s). These three tasks shall be referred to as *read input*, *update* and *render* respectively. The different ways these tasks can be interleaved in running time will define the game loop models.

As a first attempt, this concurrent execution could be achieved by running each task on a separate thread. However this approach may encounter difficulties because some hardware platforms fail to provide adequate thread support when precise timing is required (Dalmau 2003). Instead, most professional games simulate this concurrent behavior with regular single-threaded loops and timers.

Game loops can be classified into coupled and decoupled according to the implemented order of execution of its main tasks (Valente et al. 2005). In coupled loops, the three tasks are executed sequentially and at the same frequency. Coupled loops are only useful when the hardware on which the game will run is fixed and known in advance such as videogame consoles. It is not adequate for games that need to run on different machines such as computer games. To address this need, professional computer games usually implement an uncoupled game loop. This kind of loop has the advantage of allowing the execution of tasks at different frequencies. This is useful for example to increase the rendering frequency on powerful machines without changing the frequency of game logic processing. However, it is not so useful to increase rendering performance without increasing the frequency of world updates because the same scene would be rendered multiple times. What most professional games do is to separate their update task into two subtasks. Usually the game logic and artificial intelligence algorithms run at a fixed frequency while tasks that determine the positioning of visible objects into the game scene but do not affect the game logic run at the highest achievable frequency (Dalmau 2003). Animation interpolation is one example of such tasks. These types of game loops are exemplified in Figure 2.1.

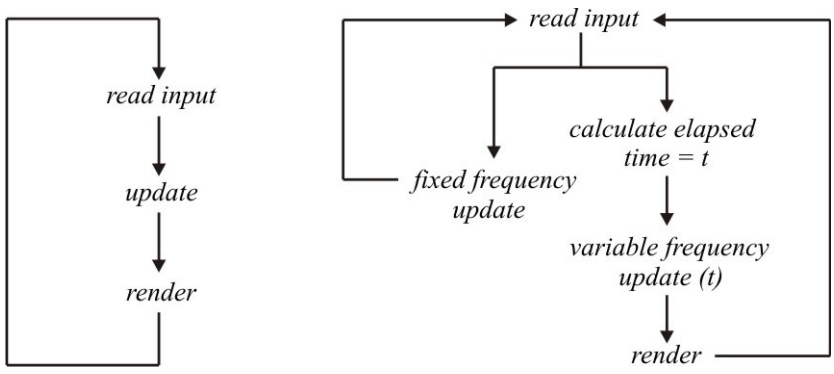

Figure 2.1 – Examples of Coupled and Uncoupled Game Loops. Source: (Valente et al. 2005).

The coupled loop on the left executes all tasks sequentially and at the same frequency. The loop model on the right actually has two loops, one that executes at a fixed frequency and one that executes as frequently as possible. The executions of the two loops are interleaved according to the speed achieved at runtime. The important conclusion here is that professional computer games

usually require that their world update frequency be defined at runtime for the variable frequency update subtasks.

2.2 Modeling and Simulation

According to the realism requirement, serious games often need to simulate the dynamics of some situations in such a way that the outcome is similar to that of the real world. Therefore, it seems very logical to look at the techniques developed to simulate real-world systems (von Neumann 1966; van Deursen 1995; Zeigler et al. 2000). The main purpose of this area is precisely to develop computational models to simulate reality.

2.2.1 The DEVS Formalism

The *Discrete Event System Specification (DEVS)* formalism introduced by Zeigler (1972) provides a way to model dynamic systems. As a discrete formalism, it models state changes as discrete instantaneous events. For any period of time where there is no event, the state remains unchanged.

Systems are modeled in DEVS as having *input* and *output* interfaces. These interfaces represent the way the system interacts with other systems. Input is the interface from which the system receives external stimuli while output provides a way of observing and receiving stimuli from the system. Therefore, systems are *modular* since their inputs and outputs are the only way of interacting with them. Figure 2.2 illustrates this formalism.

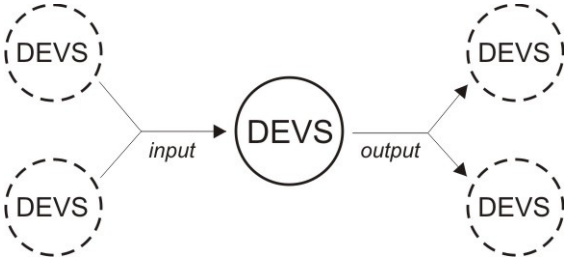

Figure 2.2 – Basic Discrete Event System Specification

A *basic DEVS* (also called *atomic DEVS*) is a structure

$$M = \langle X, S, Y, \delta_{int}, \delta_{ext}, \lambda, ta \rangle$$

where

X is the set of input values

S is the set of states

Y is the set of output values

δ_{int}: $S \rightarrow S$ is the *internal transition function*

δ_{ext}: $Q \times X \rightarrow S$ 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

λ: $S \rightarrow Y$ is the output function

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

In order to describe the interpretation of these elements, we shall assume the system has just entered some state s. If no input is received, the system will remain in s for time $ta(s)$. Once this time expires, the system outputs the value $\lambda(s)$ and switches to state $\delta_{int}(s)$. Note that, besides positive reals, $ta(s)$ can also assume the values 0 and ∞. If $ta(s) = 0$, the system will immediately go to the next state without allowing any possible input to intervene. In this case, s is said to be a *transitory state*. If $ta(s) = \infty$, the system will remain in s indefinitely until some input causes another state transition. In this case, s is said to be a *passive state*. If an input $x \in X$ is received before the expiration time, the system switches to state $\delta_{ext}(s,e,x)$, where (s,e) with $e \leq ta(s)$ is the total state at the time the input was received.

In short, the internal transition function defines the next state when no inputs are received, the external transition function defines the next state in case of an external input and the output function defines the system's output whenever the internal transition function is invoked.

The following example helps illustrate how DEVS works. Consider a controller system for a safe door that will only open it if it receives the correct password, which is 12345. If the user does not type anything for more than t_{reset},

the system is reset, the user is signaled of that and he will have to start over. If the user types the wrong password, he should wait for a system reset to start over.

The model for this system is defined as

$$M = \langle X, Y, S, \delta_{int}, \delta_{ext}, \lambda, ta \rangle$$

where

$X = \{0,1,2,3,4,5,6,7,8,9\}$

$Y = \{\text{“reset”},\text{”open”}\}$

$S = \{\text{Wrong,Open,Reset},S_1,S_2,S_3,S_4,S_5\}$

$\delta_{int}(S_n) = \text{Reset}$

$\delta_{int}(\text{Wrong}) = \text{Reset}$

$\delta_{int}(\text{Open}) = \text{Reset}$

$\delta_{int}(\text{Reset}) = S_1$

$\delta_{ext}(S_5,e,5) = \text{Open}$

$\delta_{ext}(S_n,e,n),\ n \neq 5 = S_{n+1}$

$\delta_{ext}(S_n,e,x),\ x \neq n = \text{Wrong}$

$\delta_{ext}(\text{Wrong},e,x) = \text{Wrong}$

$\lambda(\text{Open}) = \text{“open”}$

$\lambda(\text{Reset}) = \text{“reset”}$

$ta(S_1) = \infty$

$ta(S_n),\ n \neq 1 = t_{reset}$

$ta(\text{Wrong}) = t_{reset}$

$ta(\text{Open}) = 0$

$ta(\text{Reset}) = 0$

X says that the system accepts any digit as inputs. If the user does not type anything for a long enough period, the system will eventually reach the state S_1. Each password digit typed correctly will take the system from S_n to S_{n+1}, until S_5, from which it will finally reach the state “Open”. Any digit typed incorrectly will take the system to state “Wrong” and will only change to “Reset” when it stops receiving digits for time t_{reset}.

It may not feel intuitive for a process to have single inputs and outputs. In the case of the safe, the “reset” output is directed to the user as feedback while the

"open" output may be directed to a door controller. In order to make modeling more intuitive, the *DEVS with ports* formalism was created as a simple extension to basic DEVS. It is illustrated in Figure 2.3.

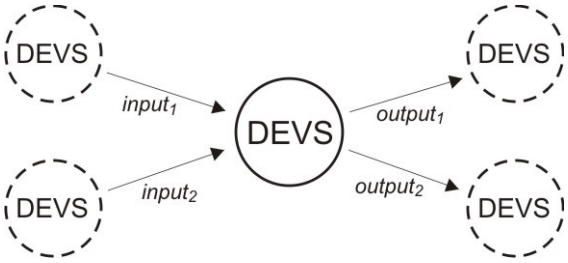

Figure 2.3 – DEVS with ports

The DEVS with ports is defined by the same structure

$$M = \langle X, Y, S, \delta_{int}, \delta_{ext}, \lambda, ta \rangle$$

where

$X = \{(p,v) \mid p \in InPorts, v \in X_p\}$ is the set of input ports and values

$Y = \{(p,v) \mid p \in OutPorts, v \in Y_p\}$ is the set of output ports and values

all other attributes are defined just as in basic DEVS

The DEVS with ports formalism allows the composition of models into higher level models as illustrated in Figure 2.4. This composition is achieved most simply by the coupling of input and output ports of different models. The *DEVS coupled model* formalizes the composition of different models. This abstraction capability makes it easier to build complex models part by part.

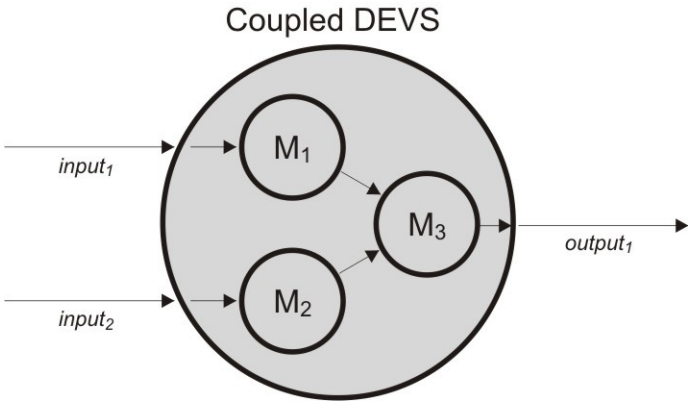

Figure 2.4 – DEVS coupled models

A DEVS coupled model is defined by the structure

$$N = \langle X, Y, D, \{M_d \mid d \in D\}, EIC, EOC, IC, Select \rangle$$

where

$X = \{(p,v) \mid p \in IPorts, v \in X_p\}$ is the set of input ports and values

$Y = \{(p,v) \mid p \in OPorts, v \in Y_p\}$ is the set of output ports and values

D is the set of components names

$M_d = \langle X_d, Y_d, S, \delta_{int}, \delta_{ext}, \lambda, ta \rangle$ is a DEVS with

$X_d = \{(p,v) \mid p \in IPorts_d, v \in X_p\}$

$Y_d = \{(p,v) \mid p \in OPorts_d, v \in Y_p\}$

$EIC \subseteq \{((N, ip_N), (d, ip_d)) \mid ip_N \in IPorts, d \in D, ip_d \in IPorts_d\}$

$EOC \subseteq \{((d, op_d), (N, op_N)) \mid op_N \in OPorts, d \in D, op_d \in OPorts_d\}$

$IC \subseteq \{((a, op_a), (b, ip_b)) \mid a,b \in D$ with $a \neq b$,
$op_a \in OPorts_a, ip_b \in IPorts_b\}$

$Select: 2^D - \{\} \rightarrow D$ is the *tie-breaking function*

Each component M_d is a DEVS model itself. A component may be another coupled model, allowing the construction of hierarchical models. *EIC* defines the *external input coupling*, connecting external inputs to components inputs. Similarly, *EOC* defines the *external output coupling*, connecting components outputs to external outputs. The *internal coupling IC* connects component outputs to component inputs. Note that no output of a component may be connected to an input of the same component, i.e. *no direct feedback loops are allowed in DEVS*. Finally, the tie-breaking function defines the order in which to carry out computations when multiple components receive inputs at the same time.

2.2.2 Cellular Automata

Considering all different formalisms to model spatial dynamic systems, *cellular automata* (CA) (von Neumann 1996) are among the most popular. Despite their simplicity, they are capable of reproducing complex behavior of systems in several fields, such as land use cover change (Carneiro 2006), urban growth (Batty 2005) and many other human-driven and natural phenomena.

A cellular automaton works in a world representation where both time and space are discretized. Time is represented by the sequence of time values t_0, t_1, … while space is partitioned into *cells*. Usually, time values represent a sequence of equally spaced instants in time and cells are subdivisions of space defined by a regular grid. Each cell has a well-defined state for each time value. The set of cells that influence state changes of a particular cell is called the *neighborhood* of that cell. A CA is defined as

$$\text{CA} = \langle C, S, N, T \rangle$$

where

C is the set of cells

S is the set of possible cell states

N: $C \rightarrow C^{|N|}$, where $|N|$ is the neighborhood size and $c \notin N(c)$, for each $c \in C$, is the neighborhood function that defines the neighborhood of each cell

T: $S \times S^{|N|} \rightarrow S$ is the *transition function* that, given the state of a cell c and the states of all its neighbors, defines the next state for c

Figure 2.5 illustrates a simple CA for a two-dimensional grid cell space. The neighborhood of each cell is the well known *Moore neighborhood*, which is composed by the eight closest cells. Cells may assume only one of two states, 0 or 1. At each time step, the transition function states that each cell must assume, at the next time step, the state of the majority of its neighbors and that it should keep its current state in the case of a tie.

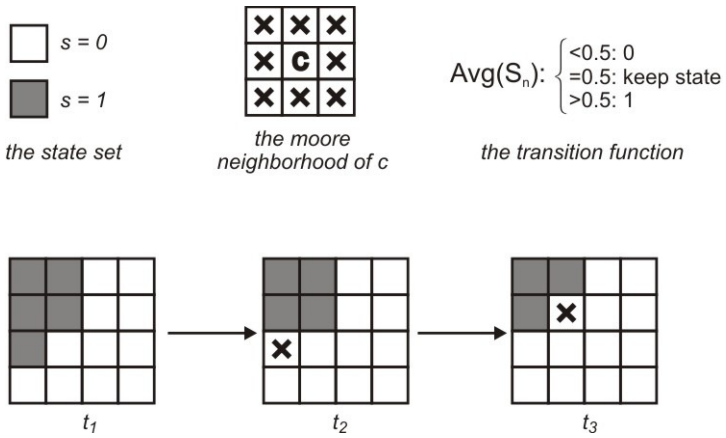

Figure 2.5 – A simple CA

The simplest procedure for simulating a CA is to scan the entire cell space at each time step applying the transition function for each cell. For this algorithm to yield the correct result, it must keep a second copy of the data structure that stores cell states. For each cell, the algorithm should read the necessary cell states from the first structure and store its next state in the second structure. This is necessary so that the cells that have not been scanned yet are not affected. Once the scan is complete, the second structure will hold the next global state.

This simple algorithm has two main drawbacks. First, it cannot handle infinite cell spaces. Second, it may be inefficient because it keeps scanning and making calculations for cells that are in the same situation as in the previous time step.

Another method for CA computation is described by Zeigler et al. (2000) as the *discrete event approach to CA simulation*. The idea of this method is to concentrate on events. In the context of a CA, an event occurs when a cell changes its state. The algorithm then works as follows: at each time step it keeps track of the set of cells that actually changed state. Then, it collects the set of all neighbors of those cells. Finally, the union of these two sets defines the cells that are going to be scanned at the next time step. All other cells will be left unchanged. This procedure assures that, if neither a cell nor any of its neighbors have changed state at a given time step, that cell will not be scanned at the next time step.

Cell Space Models

Cell space models are a more general class of dynamic models that comprises cellular automata, where the definition of local neighborhood and transition rules are relaxed (Batty 2005). The main difference from strict CA models is that cell space models allow *action at a distance*, which is characterized by causality relationships between cells at distance. Usually, physical phenomena are more easily mapped to the strict CA form, while anthropic phenomena often require some sort of action at a distance.

Cell space models are not formalized in this section because of the lack of consensus on exactly how and to what extent the CA properties should be relaxed. Section 4.2 proposes a formalism for this class of models.

2.3 Multi-Agent Systems

Multi-Agent Systems (MAS) are not targeted at any specific kind of application. Instead, the term stands for any system which is based on the agent modeling paradigm. In fact, this may be the reason why there is a lack of consensus among researchers about what are the basic concepts for modeling agents. Usually, toolkits for building MAS are targeted at one of the following types of application (Theodoropoulos et al. 2009): (1) MAS for studying complex systems, such as social models, insect colonies, artificial life and logistics; (2) MAS for distributed intelligence; (3) development of software MAS, i.e., software systems that distribute their functionalities among a set of agents, such as semantic Web agents, cognitive agents in expert systems and agents for network meta-management (e.g., load balancing or service discovery).

These different kinds of applications have different requirements and, therefore, impact the functionality offered by the toolkits for building MAS. Since this thesis is focused on the simulation of realistic situations, the first type of application is more adequate because it provides an environment which is most recognizable as a simulation engine.

Even after filtering the available MAS toolkits by the type of application, they are still too many to allow a complete study. Instead, two of them were chosen based on their fitness for the requirements enumerated in section 1.3 and also on their popularity among researchers. They are described in the following sections.

2.3.1 Jason

Jason (Bordini and Hübner 2009) is a platform for multi-agent simulation. It is a good representative of the approaches based on the BDI agent architecture (Rao and Georgeff 1992), which is one of the most popular architectures for modeling the cognitive behavior of agents. BDI stands for “Belief-Desire-Intention”. Beliefs are facts that an agent thinks are true, and together they constitute its world view. The belief set is dynamically updated as the agent interacts with its environment and other agents. Desires are the goals of the agent.

Both beliefs and desires constitute the input of the reasoning process. Intentions are the result of that reasoning process and they determine the next actions of the agent.

There are two aspects of Jason that makes it an interesting case for this work. First, it allows an agent to keep a *library of plans*, which are possible courses of action to achieve a particular goal. Second, it works with the notion of *events*, which are triggered at every change in a belief or a goal. Plans are always started by events.

Agent reasoning is done in *reasoning* cycles as depicted in Figure 2.6. Each cycle starts by the agent updating its belief base by sensing the environment. That sensing may trigger one or more events which, in turn, may trigger one or more plans, producing intentions. Then, the set of intentions compete in the *choice of intention* of the agent to be further executed in the reasoning cycle. The details of how an agent chooses which intentions will become actions are beyond the scope of this work. It suffices to mention that there is a specific procedure for that.

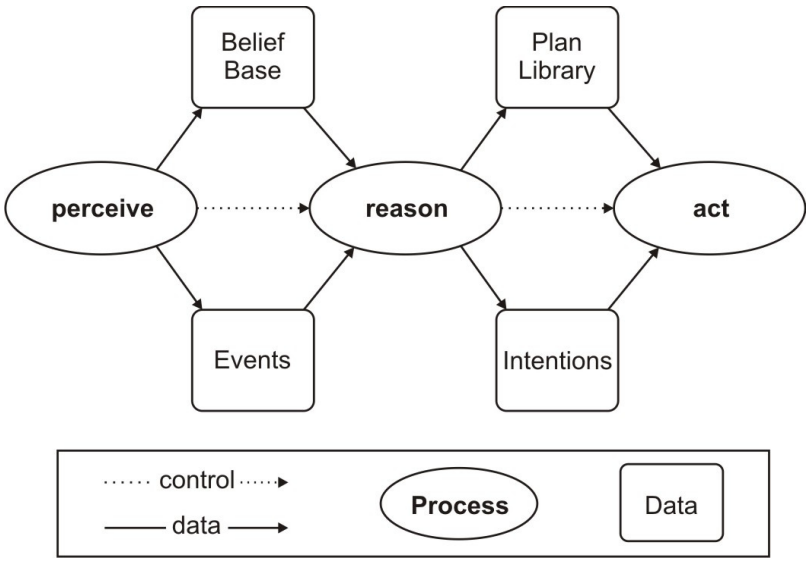

Figure 2.6 – Working Model of Jason Agents. Source: (Bordini and Hübner 2009), p. 457.

The reasoning process is defined in a prolog-like declarative language which is an extension of the AgentSpeak language (Rao 1996).

In Jason, as in most MAS, there is the notion of an *environment*, through which agents can interact. The environment is responsible for: (1) keeping its current state; (2) simulating how the actions of the agents alter that state; (3)

providing the agents with a symbolic representation of that state when they sense it. Differently from agent reasoning, environments are specified by a Java code using the Jason API for the environment. This API is flexible enough to allow the implementation of different types of environments. For example, the synchronization of the execution of the actions of an agent is totally flexible. While some simulations allow an agent to execute multiple actions in parallel, others may require the notion of a *simulation step*, in which one action is executed at each step.

Jason also provides some flexibility in its execution mode, which can be either asynchronous or synchronous. In the asynchronous mode, each agent executes its next reasoning cycle as soon as the previous cycle has finished. In the synchronous mode, each agent performs exactly one reasoning cycle at every *global simulation step*.

Comparing the basic aspects of the Jason formalism with DEVS, a few remarks can be drawn:

- In Jason, there are two distinct kinds of elements: environment and agents. In DEVS, there is only the notion of systems.
- In Jason, time is not explicitly modeled. For example, one cannot specify how long an action takes to be executed.
- Jason provides more specific constructs for implementing complex cognitive agents. DEVS provides a lower level language.
- Jason provides multiple ways to execute a simulation model. In DEVS, the result of a simulation derives entirely from the simulation model.

2.3.2 SeSam

SeSam (Shell for Simulated Agent Systems) (Klügl and Puppe 1998) is a generic purpose multi-agent simulation platform. One of its main focuses is to provide a modeling and simulation tool that is easy to use, not requiring deep programming knowledge.

Simulation modeling is done with three types of objects: *world*, *agents* and *resources*, as illustrated in Figure 2.7. All objects have an internal state defined by a set of variables. In each simulation there may be only one world. The notion of world is similar to the notion of environment in most MAS. Agents are active entities which interact with the world by sensing it and acting on it. Finally, resources are static and passive objects that are accessed and manipulated by the agents. Note that the world inherits the behavior from the agent type. This means that the world is active and can change with time, even if there is no agent acting on it.

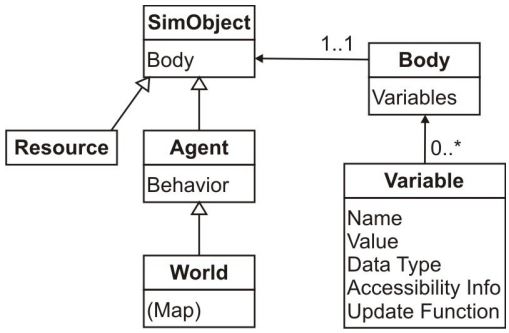

Figure 2.7 – Object Types in SeSam. Source: (Klügl 2009), p. 485.

Behaviors are defined by a set of graphs in which nodes represent activities and edges represent transition rules, as depicted in Figure 2.8. It is interesting that the behavior of an agent can be composed of multiple activity graphs. This provides a means of composing behaviors of agents from smaller behavior definitions. These activity graphs are called *reasoning engines* in SeSam nomenclature.

All reasoning engines of a simulation object are executed in parallel. Each reasoning engine may have only one activity being executed at a time. The transition rules define the conditions on which the reasoning engine terminates an activity and starts the next one. Each transition rule defines a Boolean expression that is evaluated at every simulation step. When a transition rule connecting an executing activity to another one becomes true, the executing activity is terminated and the other is started. Note that, when executing an activity, multiple transition rules may be evaluated as *true* concurrently. In this case, some sort of tiebreak rule must be applied.

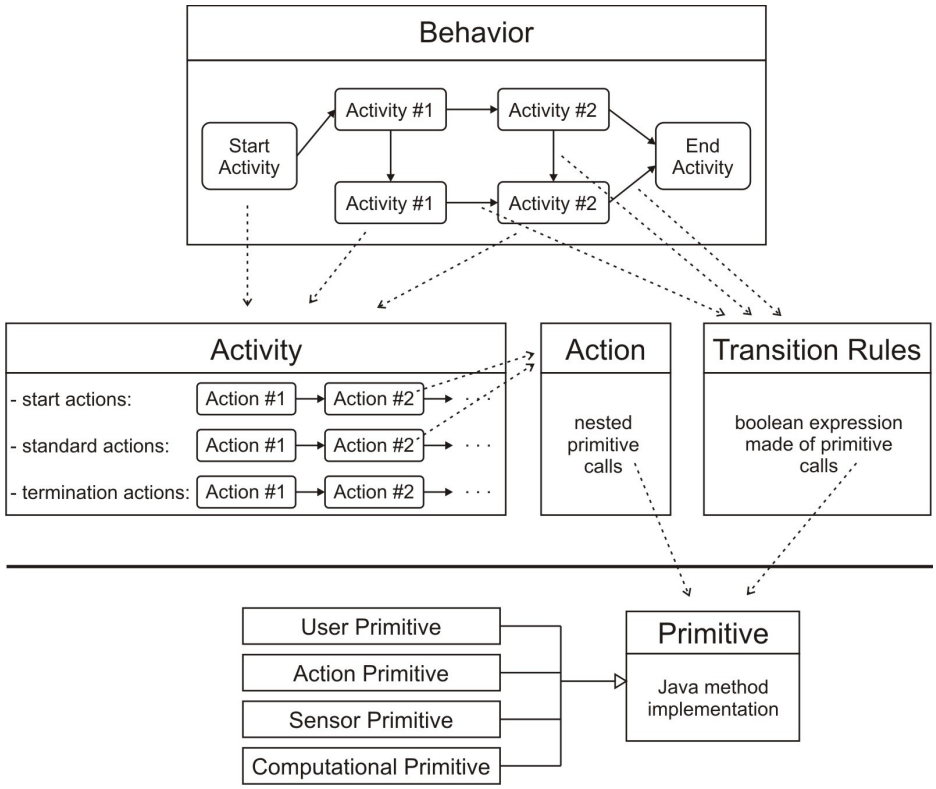

Figure 2.8 – Behaviors in SeSam

Since activity graphs can become quite large, SeSam provides means for hierarchical behavior composition, where it is possible to define *composite activity nodes*, which contain themselves another activity graph.

An activity encapsulates three sequences of actions: start actions that are performed when the activity is selected anew, standard actions that are performed once every time step as long as the agent is executing that activity, and termination actions that are executed for cleaning up when the activity is finished. An action is basically a nested set of primitive calls. The transition rules are also defined by Boolean expressions built of primitive calls.

Primitive calls are the basic building blocks of the dynamic models of SeSam. They connect the model to the underlying programming language, which is Java in this case. Each primitive is implemented as a Java class with a method named *execute*. They also define the input and output argument types. The primitive categories are: (1) action primitives, which are used to manipulate the agent's internal state or environment; (2) sensor primitives, which collect information from the agent's environment; (3) computational primitives, which

provides computations of higher or lower complexity; (4) user primitives, which consist of macros that combine calls to other primitives.

Although seemingly complex, this behavior structure allows the separation of dynamic models in two levels: the lower level, which requires Java programming skills, and the higher level, in which the lower level primitives are used as building blocks and no programming skills are necessary. This is an attempt by SeSam to make simulation more accessible to a broader class of researchers and businesses.

SeSam provides a third and higher modeling level in which the user defines a full simulation experiment. Once all the definitions for agents, world and resources are complete, the user defines a *situation*. A situation is basically a set of instance descriptions defining all instances of agents, resources and the world that are going to compose the simulation. Additionally, the user may define other properties of his experiment, such as the values that are going to be observed during the simulation execution. Although interesting, the details of this level of modeling are out of the scope of this work and will not be detailed further.

Comparing the basic aspects of SeSam with the DEVS formalism, a few remarks can be drawn:

- In SeSam, like DEVS, all simulation elements are specializations of the same abstract type *object*.
- In SeSam, time is not explicitly modeled. Every action is scheduled by a *global time step mechanism*. For example, one cannot specify directly how long an action takes to be executed. It is necessary to implement a loop that counts the time steps.
- SeSam does not restrict the behavior of agents to any particular format.
- SeSam provides means for composing object states and behaviors.
- Behaviors are defined intuitively in the form of graphs, which are similar to workflows, but without parallel activity execution.

2.4
Workflows and Planning

Workflows are activities involving the coordinated execution of multiple tasks, where each task defines some work to be done by a person or a software system (Casati et al. 1995). Workflows can be very interesting to serious games because business processes are usually modeled as workflows. In fact, integrating serious games with the so-called business process management (BPM) systems is likely to bring many benefits for personnel training and business planning.

Workflows are usually represented by *flow charts*. A flow chart is basically a graph notation for representing procedures. It uses boxes to represent events that change some data and diamonds to represent decisions, which may change the direction of the process (Sowa 2000), as depicted in Figure 2.9. A workflow is a specific case of a flow chart in which the events are actions executed by some participant. Therefore, workflows are basically structured sets of actions (van der Aalst et al 2003).

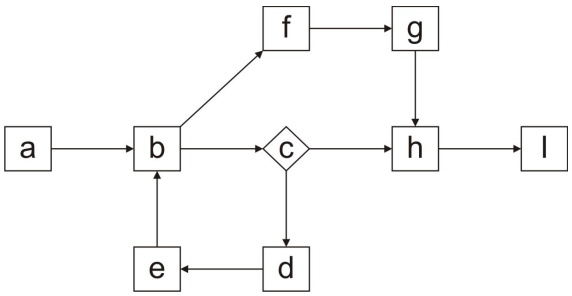

Figure 2.9 – A Flow Chart

Artificial Intelligence (AI) planning, or simply planning, is also an area of high interest to serious games. It is also an old Computer Science area. Plans traditionally refer to plans of action, which can be often represented in the form of *workflows*. The ability to dynamically generate workflows to pursuit specific objectives is widely used in gaming to model the behavior of intelligent automated characters.

Most of the pioneer work in the planning area, such as STRIPS (Fikes, Nilsson 1971) and NOAH (Sacerdoti 1977), was devoted to automatic planning in deterministic domains. These planners take as input the current state of the world, a set of possible actions with their corresponding pre- and post-conditions and a

goal proposition. Their objective was to output a course of actions that would take the world from its initial state to another state where the goal proposition is true.

These early planners assumed that the initial world state was entirely known, that the world state would not be altered by any other factor during the execution of the plan and that the effects of actions were always deterministic. Those are rather restrictive assumptions. Later work tried to relax some of them and to plan under uncertainty (Blythe 1999). Later work also allowed the design of workflows for situations with those uncertainty factors, where a planner mechanism incrementaly synthesizes new pieces of workflow during the workflow execution (Fernandes et al. 2007).

Even though most of the work in planning is devoted to automatic planning algorithms, there is much more to planning than that. Serious games could also benefit from much of the work that has been done in plan evaluation, plan recognition (Kautz 1991), hierarchical planning (Erol 1995; Giunchiglia et al 1997), searchable plan repositories and many other interesting planning problems.

2.5 Summary

This chapter briefly overviewed a few techniques and systems, selected from the areas of gaming, modeling and simulation, multi-agent systems and planning. The techniques and systems include game loops, DEVS, cellular automata, Jason, SeSam and workflows, all of which will be referred to throughout the rest of the thesis. The DEVS simulation formalism serves as the basis for the Process-DEVS formalism, described in chapter 3. In the same chapter, the properties of Jason and SeSam platforms are also referred to in the discussion preceding the formal definition of Process-DEVS. Chapter 4 defines how workflows, cellular automata and multi-agent systems can be modeled on top of Process-DEVS. Finally, game loops are used in the InfoPAE implementation case described in chapter 5. Also in that chapter, it is shown how traditional simulation models can be integrated with the specialized data structures of 3D renderers used in modern games.