5
The InfoPAE Use Case

This chapter describes two software systems implemented with the objective of improving the efficiency of emergency response actions in the oil and gas industry. The first system is a plan simulator system that is responsible for simulating the results of contingency plans stored in a database of emergency response plans. It allows its users to import emergency scenarios and associated response plans. The simulation engine is used to test the efficiency of the response plans. The second system is a training game, which simulates emergency situations in order to train people to make efficient decisions in such situations. In the domain of emergency management, the term *plan* usually means a set of actions structured in a workflow.

The two systems are based on the same simulation engine, which implements the Process-DEVS framework described in chapter 3. This architecture shows how simulation elements can be reused by different systems.

Section 5.1 gives an overview of planning for emergency situations. Section 5.2 describes the domain of contingency planning for oil leaks, for which the two systems were designed. Section 5.3 describes the simulation models in terms of the Process-DEVS formalism. Section 5.4 describes the architecture of the two systems. Section 5.5 describes the time management technique developed for the systems. Finally, Section 5.6 concludes the chapter and reports the achieved results.

5.1
Planning for Emergency Situations

An emergency situation occurs when an unexpected incident has occurred and its potential consequences involve damage to human health and to the environment. In such situations, it is not only important to respond quickly in order to minimize the damages, but the response must be conducted in a well organized manner. The complexity of emergency management, coupled with the

growing need for multi-agency and multi-disciplinary involvement on these situations, increased the need for standardized methodologies. Particularly, the *Incident Command System* (ICS) (Bigley and Roberts 2001) standard is being increasingly adopted by public safety and private sector organizations.

In the ICS methodology, the initial response steps consist of notifications, initial assessment, command meeting, initial response and incident briefing using specific ICS forms. After this initial response period, the emergency handling process becomes cyclic. This kind of process is called Planning "P". Each cycle consists in a planning phase and an operational phase. The planning phase consists of situation assessment meetings, objective updates, tactics definition, planning, elaboration and approval of the incident action plan (IAP). The operational phase consists of executing a response plan and assessing its progress, after which a new cycle begins.

The InfoPAE system (Carvalho et al. 2001) was designed as a tool for managing this complex emergency handling process, making incident response quicker and more effective. It has been in use at Petrobras, a large Brazilian oil company, for more than ten years. It also proved to be a valuable training tool. The system offers a sophisticated database for response action plans and easy access to vital information and resources allocated for different types of scenarios.

One of the difficulties of such systems is that, even though it is possible to describe an emergency action plan at a reasonably detailed level, this is somewhat limited with respect to the representation of dynamic aspects. In (Frasca 2003), the author discusses two different approaches for modeling knowledge about dynamic phenomena: representation and simulation. According to the author, the main difference between both forms is that simulation attempts to model the behavior of the elements involved in the phenomenon, while representation is limited to retaining its perceptual characteristics. To make it clear, the author gives the example of a plane landing procedure. A representation of a specific landing could be a film, in which an observer would be incapable of interfering. On the other hand, a flight simulator would allow the player to modify the behavior of the system in a way that simulates the real plane. This flexibility is only possible due to the simulation characteristic of modeling the behavior of the elements independently of any specific scenario.

Traditionally, response action plans take a more representational form, usually adopting workflows. Although response action plans contain response strategies planned for different type of scenarios, one cannot tell whether the plans are well suited for all the possibilities of evolution of an emergency situation. For example, a plan can describe the action of sending two boats to intercept an oil slick. However, it may not be possible to do that before the oil reaches the coast under some specific conditions. If emergency managers were able to simulate the whole process in a more realistic way, it would certainly make the emergency plans more reliable.

Testing the quality of response action plans, as well as the performance of emergency response teams, is mandatory to minimize the impact of the incident. In addition to other initiatives such as field exercises, the use of computational simulation can be a cost-effective and efficient mechanism to validating action plans and training response teams. Simulation can take into account many details that are difficult to consider if the planning is done exclusively by humans. For example, it can take into consideration the location of the needed resources and the specific spatial characteristics of the emergency scenario to estimate, in advance, if there will be enough time to get the necessary resources in place for executing a specific action.

Specifically, the main benefits that simulation may bring to the InfoPAE system are:

- Simulation helps finding flaws in emergency plans.
- The spatial configuration of available resources can be evaluated and optimized so that they can be deployed to handle any scenario requirements as quickly as possible.
- Simulation-based games provide training that helps improving personnel performance.
- Computer simulation cost is significantly lower than functional or full scale exercises.

Simulations are commonly used for investigating physical phenomena, such as those involving dispersion of chemical products in the environment (Karafyllidis 1997; Chinmoy and Abbasi 2006). However, the pure simulation of

physical processes does not take into consideration the effects of contingency actions. More generally, it is not enough to simulate a specific process of an emergency situation in isolation. Essentially, one must concurrently simulate all relevant processes, considering the interferences between them. For instance, a response action plan modeled as a workflow may significantly interfere with the dispersion of chemical products, which can be modeled as a cell space process. The main problem is how to combine simulations of different processes modeled in different formalisms. That is precisely how Process-DEVS and the techniques described in chapter 3 can be of great help in equipping the InfoPAE system with the necessary simulation capabilities. It can combine simulations of physical phenomena with others processes related to Planning "P".

5.2 A Motivating Example - Contingency Plans for Oil Leaks

Oil leak emergency situations constitute a common scenario that the InfoPAE system has been used for. Accidents involving the spill of a considerable volume of oil into the ocean are critical because of their potential environmental impact. Additionally, oil removal from the environment is a costly process, ranging from USD$20 to USD$200 per liter (Fingas 2000). This kind of scenario is also interesting because it involves processes of different nature, such as oil dispersion on water and response action plans. For these reasons, oil leak emergency situations were chosen as the first simulation experiment using the InfoPAE system.

In oil leak situations, the response plan, at the highest level of abstraction, consists of three phases: (1) finding and stopping the leak; (2) restricting the oil propagation; (3) recovering all possible oil from the environment.

The first phase relates mostly to plants and installations. In this phase, the response plans are usually simple and response effectiveness depends mostly on the availability of engineering information and of quick communication. After the leak has been detected and proper measures for stopping it have been taken, the focus of the response plan is on containment and recovery of the leaked oil.

The highest environmental impact usually occurs when some amount of oil hits the shore, which also causes the oil removal to grow more expensive. Oil is

usually lighter than water and does not dissolve in it. When leaked into a water body, it remains concentrated on the surface, forming one or more oil slicks. These oil slicks are shaped and moved by external forces, such as wind and water currents. If these forces push an oil slick towards the coast, it is almost certain that it will cause a large concentration of oil along some particular coastal segments. There are many types of oil, but their dispersion and evaporation rates are usually too small to prevent coast hits.

A coastal segment is environmentally sensitive to oil because of the concentration and diversity of animal species and ecosystems found on the segment. Each type of coast has its own particular characteristics and sensitivity to oil. For example, the InfoPAE project divides the Brazilian coast into discrete segments, classified according to their environmental sensitivity characteristics. Each point in the coast belongs to exactly one segment, which in turn belongs to one sensitivity class.

The main method for preventing coastal damage is to restrict the oil propagation by employing floating *containment barriers*, which are also called *containment booms* (Fingas 2000). The barriers are usually deployed in U-shape in an attempt to trap the oil slicks according to the direction they are moving. The main resources needed to place a containment barrier are the barrier itself, one or two boats with a minimum crew and some source of information about the location of the oil. The spatial configuration of all those resources is very important to determine the time necessary to install a containment barrier at a given location. Therefore, planning in advance the locations where the resources are kept is crucial. For instance, in a badly planned resource configuration, depending on that time and on the velocity of the oil slick, it may not be possible to prevent the oil from reaching a critical coastal segment.

In order to optimize spatial resource planning, it is necessary to consider various factors, such as the set of likely locations of possible oil spills, the set of likely climate conditions relevant to the movement of oil slicks (mainly wind and water currents) and the location of the most vulnerable nearby coastal segments. As a general rule, recovering oil from the coast usually takes more time and money than from water. Therefore, resource planning and speed of response is critical for minimizing coast hits.

Finally, the last phase of the response strategy is to remove as much oil as possible from the environment. The recovery of oil starts when a stable situation is reached, namely when the oil stops moving, either because it is trapped in containment barriers, or because it has hit the coast. The oil recovery process is carried out by a number of different processes, with different equipments. The choice of the process depends on the type of oil and on the characteristics of the situation. For example, for oil slicks trapped in containment barriers, the use of skimmers from a boat is usually appropriate. As for recovering oil from the coast, there are many different procedures. Usually the best procedure depends on the type of oil and on the characteristics of the coast. Common procedures include manual removal, flooding or washing, use of vacuums, mechanical removal, tilling and aeration, sediment reworking or surf washing, and the use of sorbents or chemical cleaning agents (Fingas 2000).

5.3 Simulation Dynamics

This section explains the dynamics of the InfoPAE simulation, which was modeled on top of the Process-DEVS framework.

5.3.1 The Environment

The environment is depicted in Figure 5.1 and contains all data necessary for the simulation. This data is mostly geospatial in nature and can be classified either as static or dynamic.

Static data is retrieved from the InfoPAE database and consists mostly of two-dimensional GIS data. It includes all coastal segments in a given area, with their sensitivity classification, the plant installations and other information relevant to the logistics of resource displacement, such as the location of piers. It should be noted that the coast segments are also used to determine the extension of the water bodies. Presentation information, such as satellite images, will not be listed here. Even though they are important for the final user of the system, they are relevant only to its user interface and not to its underlying simulation.

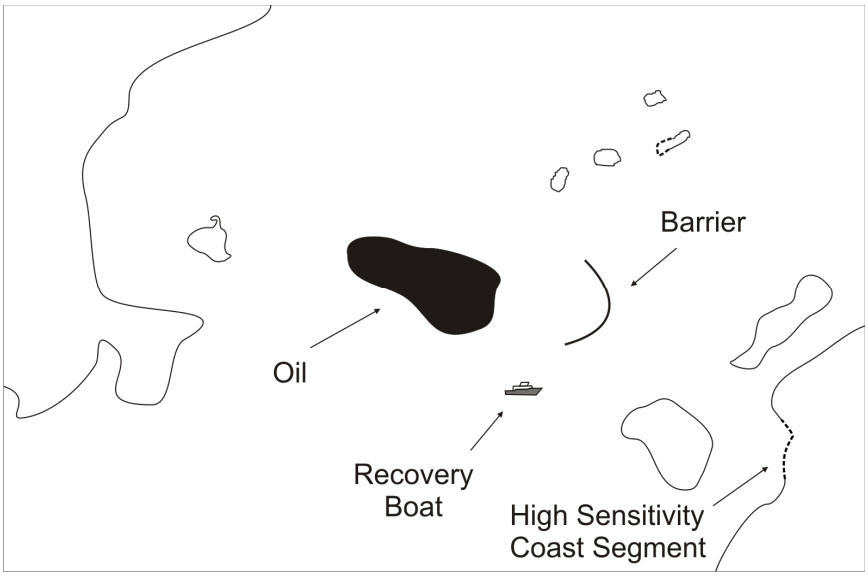

Figure 5.1. The environment with its elements

Dynamic data includes the weather conditions, the location of oil slicks and the location of resources, such as containment barriers, recovery boats and coastal cleaning teams. The relevant weather conditions include water currents and the direction and velocity of the wind. Oil slicks are represented in a regular grid cell space, where each cell contains a value that represents the amount of oil in it. Containment barriers are represented as lines which are basically sequences of points. Finally, recovery boats and coastal cleaning teams are represented simply as single points, and they are able to remove oil from any location within a fixed radius of their position.

According to the framework definition presented in chapter 3, the processes in a simulation access the state of the environment through *environment views*. The main views this environment provides are the *vector view* and the *cell view*, as depicted in Figure 5.2. In the vector view, all data is read in vector format, such as points, lines and polygons. In the cell view, everything is represented in a rectangular grid of cells. Although these two views are different in nature, both represent the same data. Elements that are fundamentally represented as vectors, such as those just mentioned, are presented in the cell view as if they occupy all cells that intersect their vector geometry. Likewise, cellular elements, such as oil slicks, are represented in the vector view as a set of points. In the case of oil slicks, each cell that contains some amount of oil is presented as a 2D point placed at the center of the cell, with an attribute indicating the amount of oil in that cell.

Those two views will feed processes of different nature, modeled in different formalisms. For example, the process that models the oil dispersion may be modeled as a cell space process using the cell view as input, and a process for barrier placement may use vector algebra, based on the vector view.

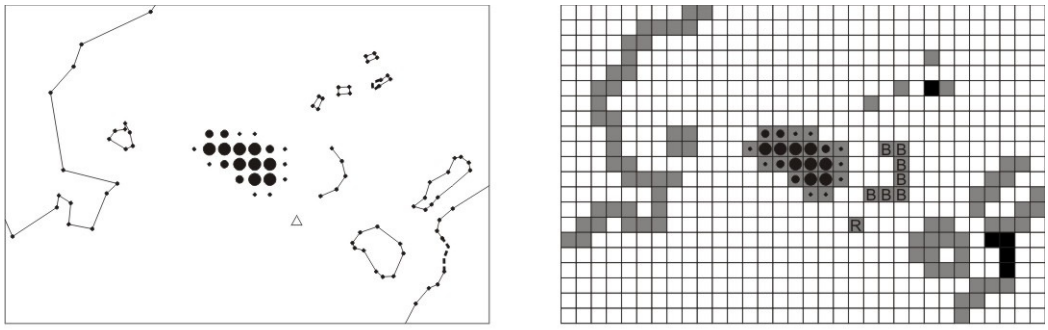

Figure 5.2. The vector view (a) and cell view (b)

Besides those two main views, the environment also provides the *properties view*, through which the processes can access all non-spatial data, such as the weather conditions. This view is accessed as a set of *property-value* pairs.

It is generally a good practice to put as little intelligence as possible in the environment. For this reason, the environment described here behaves like a database. It stores data, serves that data in the form of views, according to the needs of its clients, and processes transactions. The transactions consist of events sent by the processes. The events this environment can receive are:

OilLeakEvent(cell, amount) – adds the given amount of oil to the given cell.

OilRecoverEvent(cell, amount) – subtracts the given amount of oil from the given cell.

OilMoveEvent(origin cell, destination cell, amount) – moves the given amount of oil from/to the given cells. It subtracts the amount from the origin cell and adds to the destination cell.

ChangeResourceLocationEvent(resource, geometry) – changes the location of the given resource. The resource can be a containment barrier, a recovery boat or a coastal cleaning team. The new location is defined by the given geometry. The geometry must be checked against the type of resource. For recovery boats and coastal cleaning teams, the geometry must be a point. As

for containment barriers, the geometry must be a line with length no greater than the total length of the barrier.

ChangeWindEvent(direction, velocity) – changes the wind.

ChangeWaterCurrentEvent(direction, velocity) – changes the water current.

For simplicity, it is assumed that the wind and water current are uniform fields, with the same value at all points. This simplification may cause the simulation to behave unrealistically, if the simulated area is large enough. However, a detailed model for those conditions is out of the scope of this work. This simplification was made in order to keep the text more didactic with respect to the simulation mechanisms.

There are two modes in which this InfoPAE environment may operate during a simulation. In *memory mode*, the states of all elements are kept in main memory, i.e., there is no communication with any persistence device during the course of the simulation. The other mode is the *saving mode*. In this mode, every time an element has its state changed, the environment feeds a spatio-temporal database with the new state of that element. Hence, for each dynamic element in the simulation, there will be a time series in the database. With those time series, the sequence of world states of the simulation can be replayed after the simulation has finished.

The memory mode is used to achieve better performance. For example, when a user is designing a response plan for a given situation, he may run a large number of simulations until he is satisfied with his plan. It is not necessary to save them all. His work will be more efficient if the simulations are executed in a faster way. On the other hand, the saving mode is important when one must replay the simulation for analysis. A good example is a multi-player training game.

5.3.2 Processes

The set of processes is what gives life to the dynamic elements in the simulation, and they are responsible for modeling all kinds of behavior, from oil dispersion to the installation of containment barriers.

Recall that, in a simulation, processes act by sending events to each other and to the environment, and the coupling structure of the simulation defines the

connections through which the events are sent to other processes and to the environment. The overall structure of the processes in the InfoPAE simulation is detailed in Figure 5.3. For simplicity and ease of understanding, this figure only shows the hierarchy of processes and the flow of events that alter the environment. The environment views and other less important details were omitted from this figure. Every circle in the figure represents a process. The arrows represent either parental relations between processes or connections in the coupling structure of the simulation, through which the events flow.

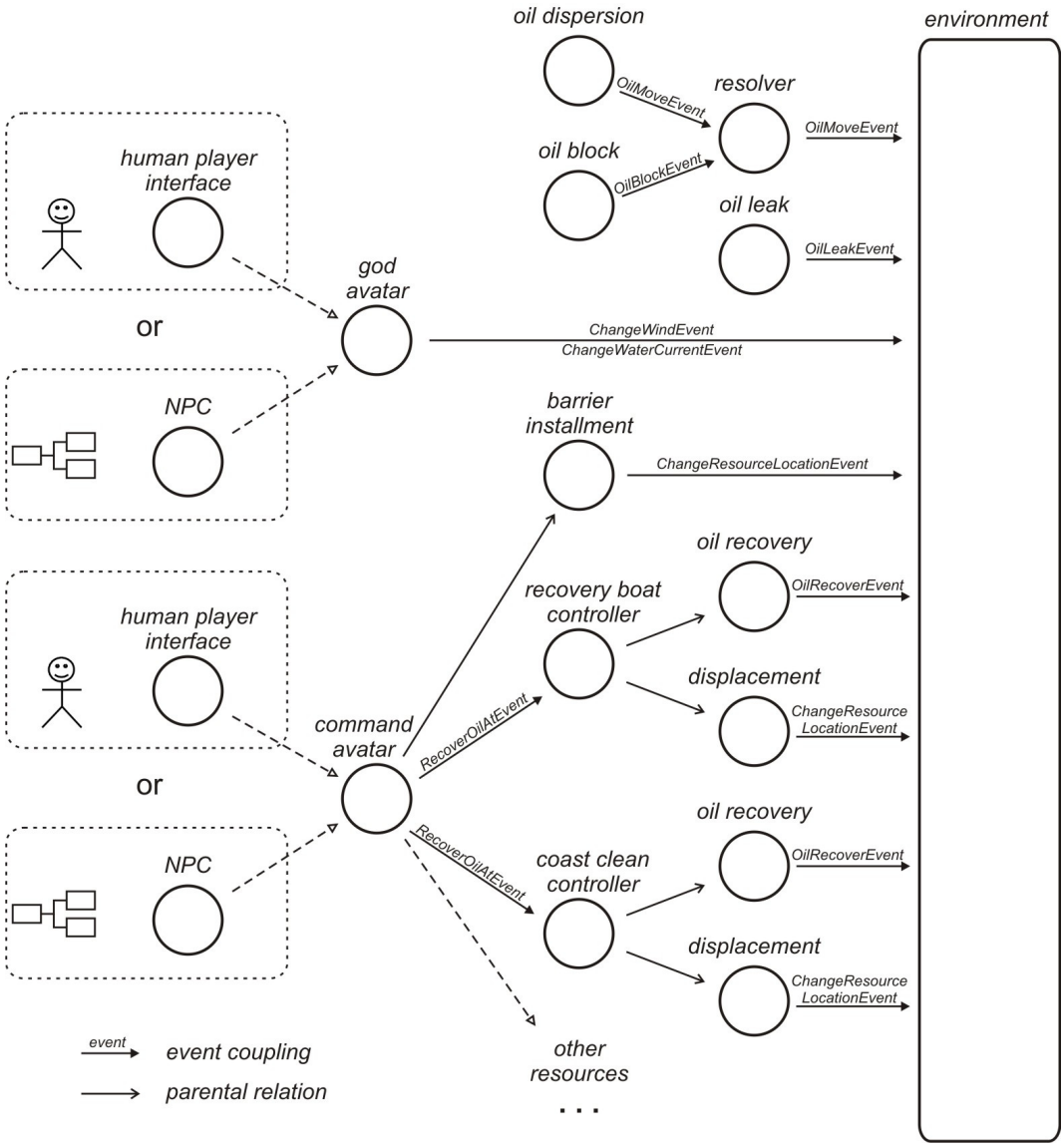

Figure 5.3. The process structure

The process structure is not fixed. The number of resources in a simulation may vary, and so does the number of processes to manipulate them. Additionally,

the kind of process that controls the *god avatar* and *command avatar* processes can also change, depending on the kind of simulation and the number of human players. In this context, an *avatar* represents a role in a simulation that is to be played either by a human player or by a fully automated process. The different types of processes are described next:

oil_leak(cell, *leak_amount*, *leak_rate*, *frequency)* – This is a very simple process which starts the whole simulation activity. The idea is that there is an oil leak at the given *cell*, which leaks at rate *leak_rate*. The total amount of oil to be leaked is given by *leak_amount*. This process updates the environment with the given *frequency* by periodically sending events in the form *OilLeakEvent(cell*, *amount)* to it. Since it is a periodic process, its time-advance function is constant *ta(s)* = 1 / *frequency*. It keeps generating these events until the total amount of oil leaked reaches *leak_amount*. Therefore, the total number *n* of generated events is equal to $\lfloor$*leak_amount / (leak_rate / frequency)*$\rfloor$. The parameter *amount* of each event is given by *(leak_rate / frequency)*, except for the last event, for which it is given by *leak_amount – (n* – 1*) * (leak_rate / frequency)*, where *n* is the total number of events. Once all those events are sent to the environment, the process is finished.

oil_dispersion – This process models the movement of oil slicks, considering the wind conditions and water currents. This process also generates events periodically. At each time step, it reads the weather conditions from the *properties view* and searches the *cell view* for all cells that contain some amount of oil. Then, it invokes a function that takes as input all this gathered data and outputs a set of events in the form *OilMoveEvent(origin cell, destination cell, amount).* As the coupling structure indicates, those events are sent to a *resolver process*. This function is complex and its internal details are out of the scope of this work. For the matter of understanding the simulation logic, it suffices to specify the format of its input and output.

oil_block – This is also a periodic process. At each time step, it checks the *vector view* for the location of all containment barriers. After that, it

calculates which cells intersect the geometries of the barriers and generates one event in the form *OilBlockEvent(cells)*, where *cells* is the set of cells that intersect some installed containment barrier.

resolver – The three processes *oil_leak*, *oil_block* and *resolver* are arranged in an *interference pattern*, which is described in section 4.4.2. The resolver process receives events of types *OilMoveEvent* and *OilBlockEvent*. It outputs only events of type *OilMoveEvent*. In its internal state $s \in 2^C$, where C is the set of all cells in the cell space, it keeps the set of cells that are blocked. Each time it receives an *OilBlockEvent(cells)*, its internal state becomes $s = cells$. Each time it receives an *OilMoveEvent(origin cell, destination cell, amount)*, it forwards it immediately as output only if *destination cell* $\notin s$, otherwise the event is ignored. Therefore, this resolver process acts like a filter of events, retaining all movement of oil that is contained by the barriers. This way, the logic of oil containment is separated from the complex logic of oil dispersion.

oil_recovery(resource_id, *action_radius*, *recovery_rate*, *max_capacity*, *frequency)* – This process removes oil from the environment. This process is used both by recovery boats and by coastal cleaning teams. The *resource_id* indicates which resource is recovering oil. The location of the resource is a point in a 2D space and can be obtained from the *vector view* at any time. The *action_radius* indicates the maximum distance from the resource's location where oil can be recovered. The *recovery_rate* indicates the rate at which this resource can remove oil from the environment. The *max_capacity* is the maximum amount of oil that can be recovered. Finally, the *frequency* has exactly the same semantics as in the *oil_leak* process. It indicates the frequency at which the environment is updated.

The internal state of this process is defined by the variable *remaining_capacity* $\in \Re^+$. Its initial value is *max_capacity*. At each step, this process invokes a function which outputs a finite set O of events of the form *OilRecoverEvent(cell, amount)*. The internal details of this function are omitted for simplicity. It is only important to know that this function must obey the following restrictions:

1. $sum(\{a \in \Re \,/\, OilRecoverEvent(c,a) \in O\}) =$
   $min((leak\_rate / frequency), remaining\_capacity)$
2. $(\forall o \in O)(o = OilRecoverEvent(c,a) \Rightarrow c \in R)$
   where R is the set of cells intersecting the circle centered at the resource's current location with radius *action_radius*

The first restriction imposes that the oil must be recovered at a rate equals to *recovery_rate*, and also that the process does not recover more oil than its capacity. The second restriction states that all recovering must be done within the action area of this recovery process. After the events have been generated, the internal state is updated. From the *remaining_capacity*, it is subtracted the value *min((leak_rate / frequency), remaining_capacity)*. When *remaining_capacity* = 0, the process is finished. Hence, it is guaranteed that the process will not recover more oil than its *max_capacity*.

displacement(resource_id, *trajectory*, *speed*, *frequency)* – Moves the resource defined by *resource_id* along the line defined by *trajectory* with the given *speed*. The *frequency* parameter defines the frequency at which this process will update the position of the resource. This is a periodic process with *ta(s)* = 1 / *frequency*. Consider a parametric function *d*: [0, *length*] $\rightarrow \Re^2$, where *length* is the total length of the trajectory and *d(x)* is the point in the trajectory reached by walking *x* space units along the trajectory, starting from its origin. The internal state of this process is defined by the variable *current_location* $\in$ [0, *length*], for which the initial value is 0. At each step, this process outputs one event in the form *ChangeResourceLocationEvent(resource_id*, *new_position)*, where *new_position* = *d(min(current_location* + *speed* / *frequency*, *length))*. After generating this event, its *current_location* is updated to *min(current_location* + *speed* / *frequency*, *length)*. When *current_location* = *length*, the resource has reached its destiny and the process is finished.

barrier_installment(resource_id, *location*, *frequency)* – Installs the barrier defined by *resource_id* at the given *location*. Of course, the resource with

the given *resource_id* must be a containment barrier. The location is defined by a polyline in the 2D space with length no greater than the total length of the given barrier. The detailed procedure for installing a containment barrier involves two boats, which should meet at a particular point, set the barrier on water and start moving in opposite directions, each one holding one end of the barrier. According to the climate conditions, some complex movement may be required to keep the barrier in the desired shape. However, since the focus of the simulation is training and resource planning, it is not necessary to model this process in such a level of detail. Instead, this process just calculates and waits for the total time spent until the boats meet at the location desired for the barrier. It then starts sending periodic events in the form *ChangeResourceLocationEvent(resource_id, location)* as new segments are added to the geometry of the installed barrier. The periodicity of these events is given by 1 / *frequency*. When the barrier is totally installed, this process finishes. The internal calculations of this process are complex and are omitted here for simplicity.

recovery_boat_controller(resource_id) and *coast_clean_controller (resource_id)* – These processes control the resources that are responsible for removing oil from the environment. Unlike containment barriers, which are treated as passive objects, those resources are active elements in the sense that they perform actions that alter the environment, hence the need for controllers. Each recovery boat and each coastal cleaning team must have one controller process. The controller processes receive commands in the form *RecoverOilAtEvent(location)* from the *command avatar* process and orchestrates its children, namely *displacement* and *oil recovery* processes, in order to execute those commands.

Initially, the controller process reads the attributes of its controlled resource, which is defined by *resource_id*. Those attributes define values for properties such as speed, recovery capacity and recovery rate. Then, it waits for a *RecoverOilAtEvent(location)* command. Once it is received, it checks the location and traces a route to it by using some routing algorithm, whose details are omitted for simplicity. Then, it forks a *displacement* process and

waits for it to move the resource to the desired location. Once the resource is properly located, the *oil recovery* process is used to perform the recovery action. If some other *RecoverOilAtEvent* is received, all current activity, if any, is cancelled and the operation starts over. This way, the controller process provides a high-level abstraction for the recovery resources by using the *composite pattern*, as described in section 4.4.3.

command_avatar – This process provides the abstraction of an avatar for the response command in the simulation. Its functionality consists basically in receiving commands and delegating them to its children. It provides one additional level of abstraction with the *composite pattern*. In fact, the whole tree of processes below the command avatar represents the execution of the emergency response. This process receives events of the form *DeployResourceAtEvent(resource_id, location)*. If the resource identified by *resource_id* is a containment barrier, it forks a new process *barrier_installment(resource_id*, *location)* if that barrier has not been deployed yet. If the resource is a recovery resource, it simply sends an event *RecoverOilAtEvent(location)* to the appropriate controller.

god_avatar – This process provides an avatar for manipulation of the weather conditions. It receives commands as events in the form *ChangeWindEvent(direction, velocity)* or *ChangeWaterCurrentEvent (direction, velocity)* and simply forwards them directly to the environment. The purpose of this process is merely to make the process structure more uniform. It is analogous to implementing an avatar interface where one can plug either a human player interface or another fully automated process.

human_player_interface and *non-playing character (NPC)* – These are the processes that can be attached to the avatars. A human player interface is an input process, as defined in section 3.3.1, which is able to receive commands from a *human-computer interface* (HCI), which is external to the simulation. This way, a human can interfere with the simulation. The capabilities of the avatar will define the human's role in the simulation. Another possibility is to attach a fully automated process to the avatars. In this case that process would be a *non-playing character* (NPC). In the

computer games field, this term is used to denote a fully automated character that plays a specific role in the game. The NPCs implemented for the InfoPAE simulation are processes that act based on workflow definitions, as described in section 4.1. For each action in the workflow, the workflow process, as defined in that section, forks a child process which communicates with the avatar process by sending the events relative to that action.

Hence, the set of human players is flexible. Each avatar may be controlled either by a human or by a predefined workflow. In a multi-player game, all avatars may be controlled by humans. In a fully automated simulation, all avatars may be controlled by predefined workflows.

Most processes in the simulation are periodic, i.e., they could be defined in a discrete time formalism. However, they work with different time steps, as listed below:

oil_leak – 10seg	oil_dispersion – 5min
oil_block – 1min	oil_recovery – 10seg
displacement – 10seg	barrier_installment – 30seg

Some processes are rigid with respect to their time steps. For example, the *oil_dispersion* process only works correctly with the right time step. However, most of them are somehow flexible with respect to the time step because they use their frequency parameter to do their calculations. For example, if we double the time step of the *oil_leak* process, it will automatically double the amount of oil that is leaked at each time step. Such processes can have their time steps adjusted to optimize the simulation performance. However, there is a minimum granularity required by each of them so that the simulation results remain correct, according to a given criteria.

It should be reminded from the simulation operational semantics that all events are time-stamped and totally ordered. Therefore, although the environment acts like a database, there are no concurrency issues, since all events are always processed in the same order.

5.4
The InfoPAE Plan Simulator and Training Game

Two different systems were implemented with the InfoPAE simulation. The first one is a simulator for the InfoPAE planning module, which provides an environment in which InfoPAE users can test the response action plans they design with the InfoPAE plan editor. The second system is the InfoPAE training game, which provides an environment to simulate an emergency situation with which multiple humans can interact.

Both systems are based on the same simulation model. The only difference between them is that, in the planning module, NPC processes are used to control the avatars, while in the game, these processes are replaced by human player interface processes, as described in the previous section. All other processes are reused with the same configuration.

The following sections describe the architecture and functionality of each system.

5.4.1
The InfoPAE Plan Simulator

As already mentioned in section 4.1.1, simulation can be a valuable tool in the process of business process planning. The idea of the plan simulator is to act as a fast and low cost tool for simulating response plans designed in the InfoPAE system. Hence, the plan designer may quickly detect flaws in his plans and test different alternatives, searching for more efficient plans. The architecture of the plan simulator is depicted in Figure 5.4.

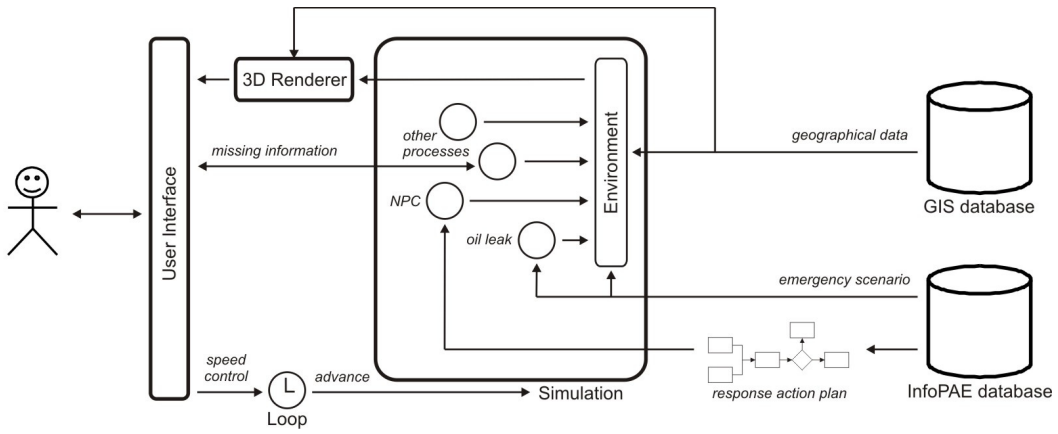

Figure 5.4. The Plan Simulator Architecture

In the InfoPAE planning module, which is not illustrated in the architecture, the user defines the emergency scenario and designs a response action plan for that scenario. The result is then stored in the InfoPAE database. The plan simulator reads this information from the InfoPAE database to build its simulation with the structure defined in section 3.3.1. A response action plan is modeled as a workflow in the InfoPAE database. During simulation execution, this workflow is used by an NPC to control the command avatar. In addition to the scenario and response plan information, the plan simulation also needs geographical information such as the coast geometry with its oil sensitivity data, which is necessary for simulating coast hits and calculating the total environmental impact.

One interesting point here is that the emergency scenario and the response plan in the InfoPAE database do not provide all the information needed for a simulation. Detailed information about the emergency, such as the exact oil leak coordinate, the leak rate and the total amount of leaked oil, are often missing from the scenario definition, so are the exact weather conditions, such as the wind direction and speed. That happens because scenario definitions are required to be a little abstract so that they can represent a larger number of concrete emergency situations. Otherwise, if the user was always forced to provide complete details, the number of scenario definitions in the InfoPAE database would grow beyond the reasonable. The same happens with response plans, which rarely define all the exact parameters for every action.

When the user imports a scenario definition and a response plan to build the simulation, the plan simulator provides an interface for defining all the missing detailed information. However, the user may still leave some information undefined. In this case, once the simulation has started executing, as soon as a simulation process needs missing information, the simulation is automatically paused and the plan simulator queries the user for that information so that the simulation can proceed. This is accomplished by the exchange of events between the user interface and the simulation through I/O processes, as specified in section 3.3.1.

The user interface provides controls for the user so that he can play, pause and set the speed of the simulation whenever he wishes. Those requests are sent to the loop component, which implements the *StableGameLoop* described in section 5.5.2. The current situation is presented to the user in a 3-dimensional scene,

which is rendered by a 3D engine similar to those used by entertainment games. In order to optimize the rendering performance, the environment implementation used by this plan simulator stores all its data in main memory and in data structures specialized for rendering by the 3D engine, as discussed in section 3.2.2 (decision 4). Screenshots of this graphical interface are shown by Figure 5.5.

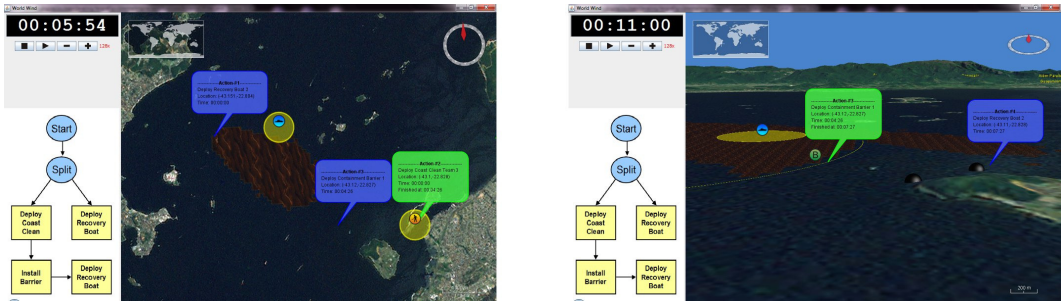

Figure 5.5. Screenshots of the Plan Simulator User Interface

The 3-dimensional interface provides the users with a realistic view of the situation evolution. Besides the position of moving objects such as oil and the resources, some additional information is rendered on the map. The action radius of some resources such as recovery boats and coast cleaning teams helps the user visualize the efficiency of their deployment and think about alternatives. The 3D visualization also allows the users to check what is visible to the people on specific points in the action field, such as helicopters, boats and coastal points.

5.4.2 The InfoPAE Training Game

The second implemented system was a training game. Its architecture is depicted in Figure 5.6. This game uses a multi-touch table as a device where several players can work together to handle the simulated emergency situation. The table is provided with a horizontal screen capable of processing multiple touch inputs simultaneously. Below that screen is a regular PC-like computer that is connected to a game server via a network. This computer hosts a small interface program that translates the inputs of the players into commands for the game server. This game server contains the simulation, the game loop and a Web Map Service (WMS) (Percivall 2003), which implements a standard way of serving

map images on the Web. One of the benefits of the multi-touch table is that it facilitates collaboration between players.

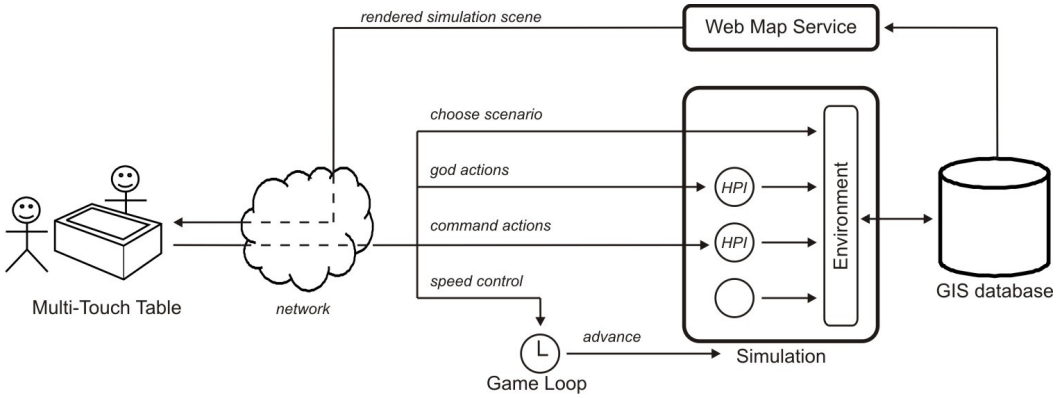

Figure 5.6. The Multi-Player Training Game Architecture

Before the game starts, one player has to choose one emergency situation out of a number of predefined ones. These predefined emergency situations differ from the scenarios stored in the InfoPAE database in the sense that they contain all the detailed information needed to run a simulation.

Once the initial situation is chosen, the game server builds the simulation and starts executing it. The simulation of the game is basically the same as in the plan simulator. Only the NPC processes are replaced by human player interface (HPI) processes, as described in section 5.3.2. The simulation receives from the table both response action commands and requests for changing the weather conditions. Response action commands are forwarded to the command avatar, while requests for changing weather conditions are forwarded to the god avatar. Finally, there is one last type of input from the table, which consists of requests for changing the game speed. Those requests are sent to the game loop and handled as described in section 5.5. Speeding up the game speed may be desirable when there is no decision making by the players.

The *game loop* component implements the *StableGameLoop* described in section 5.5.2. It keeps a thread that continuously advances the simulation time and provides the multi-touch table with updates on the simulation environment. These updates contain the state of the elements in the environment that are rendered to the players, such as the oil position and the locations of the resources. All this

information is rendered on top of a map, which is provided by the Web Map Service. A screenshot and a picture of the game are shown in Figure 5.7.

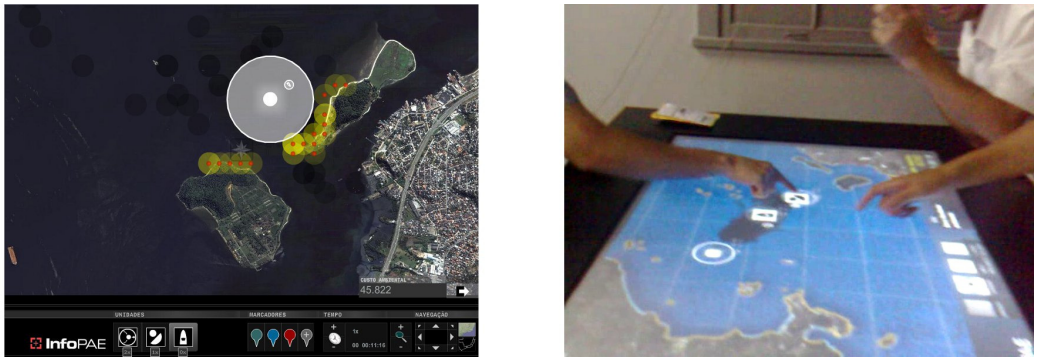

Figure 5.7. The Multi-Player Training Game in Action

The multi-touch table added considerable value to the game. The players can talk to each other and discuss the correct strategy while allocating the resources to mitigate the oil leak. It is interesting that, although this game was not designed for entertainment, its users found it fun to play with. This shows the power of games to engage people, which can be exploited by companies to stimulate discussions, to develop solutions and to propagate knowledge about a given problem.

5.5 Time Management

During the implementation of the two InfoPAE modules, problems with time management were detected in the context of current game loop techniques. None of them seemed to handle properly changes in the simulation speed and the processing peaks generated by complex simulation models. This section informally discusses the principles involved in dealing with those requirements and presents the loop model developed for the InfoPAE system.

The problem of time management lies in that computer games and, more generally, interactive simulations, need to implement some way of synchronization between the speed at which the simulation advances and the real time flow. This problem is not as simple as it might look. The game loop techniques described in section 2.1.1 show how entertainment games usually deal with this problem. However, these loops do not take into consideration the

requirements of changing the simulation speed during play and handling simulation processing peaks.

Since serious games often attempt to simulate real situations, it is natural that the simulation time represents the real time of those situations. However, simply synchronizing the simulation time with the real time flow may not be enough for all serious games. The ability to accelerate and slow down the pace of the game may be quite important for the usability of serious games. For example, consider a game which simulates an emergency situation which may last for days. The game simulation should obviously not take the same amount of time. Periods requiring no decision making should be fast-forwarded. Likewise, periods of intense decision making could be slowed down for training purposes.

Serious games often make use of complex simulation models. This easily becomes a time management issue because, unlike entertainment games, these models cannot be tricked or simplified when they produce processing peaks. Therefore, game loops designed for serious games cannot assume that their simulation models will not exceed certain processing time limits.

5.5.1 Simulation Speed and Game Loops

The human beings always work in *real time*. It cannot be accelerated or slowed down. Therefore, simulations that interact with humans must implement some sort of synchronization mechanism. The synchronization problem consists of adapting the simulation of automated elements to the real time flow by monitoring the speed at which the simulation is running and adapting its advance policy accordingly. The average simulation speed is calculated by $speed = \Delta t_{sim} / \Delta t_{real}$, where t_{sim} is the *simulation time* and t_{real} is the real time. The desired value of the speed will vary unpredictably in time depending on the will of the user. One example of how the speed can be changed during play is shown in Figure 5.8. When $speed = 1$, the simulation is synchronized with the real time flow. Greater values mean that it is in accelerated mode and lesser values in slow motion. Pauses obviously have $speed = 0$.

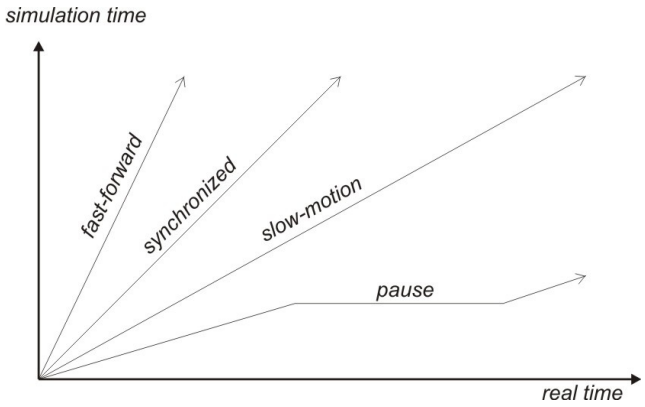

Figure 5.8 – Simulation speed being changed during play

As already mentioned in section 2.1.1, time management in single-player games is traditionally done by a loop which interleaves calls to the three functions *input*, *update* and *render*. The term *frame rate* is used in gaming to denote the frequency in real time that the render function is invoked. Both input and render functions represent simulation I/O. For simplicity, we shall consider only two functions: *update* and *process_io*.The update function is responsible for advancing the simulation time, while process_io is assumed to handle all I/O, including rendering. The idea is that the simulation system alternates between advancing its internal simulation and communicating with external entities. In a single player desktop game, it means to receive user input and render the user view. However, considering the case of a network game, it could mean exchanging update messages with its peers instead of rendering to the screen.

To maintain consistency with gaming terms, we shall use the term *frame rate* to denote the frequency in which the *process_io* function is invoked, even if this function does not render a frame for the user as, for example, in the network case just described.

Both functions are defined as

```
process_io()
{
  current_state := current_state.flush_io()
  render() //if necessary
}

update(dt)
{
  current_state := current_state.advance(dt)
}
```

where *dt* is the simulation time advance, *current_state* is the current simulation execution state and the *advance* and *flush_io* functions are as defined in section 3.3.2. After the call *update(dt)*, the simulation time is increased by *dt* and its state is update accordingly.

Some game loops define their *update* function without the *dt* parameter, considering a fixed predefined time increase. This kind of loop assumes a *discrete time* simulation model, which is not enough to handle *discrete event* simulation formalisms, such as Process-DEVS.

Other more sophisticated and highly interactive game loops divide the update tasks between two update functions. One that is executed in a fixed frequency and another one that runs at a variable frequency. The first is used for tasks that do not present relevant results in brief time intervals such as the game logic. The second is used for tasks like animation interpolation, which produces smoother results if executed in a high frequency (Valente et al. 2005). In this case it makes sense to make multiple calls to the variable frequency update and process_io pair of functions between two consecutive calls to the fixed frequency update, as depicted in Figure 5.9 (a). The fixed frequency update must be called exactly once in a given real time period. The remaining time is then used to make calls to the variable frequency update and process_io functions.

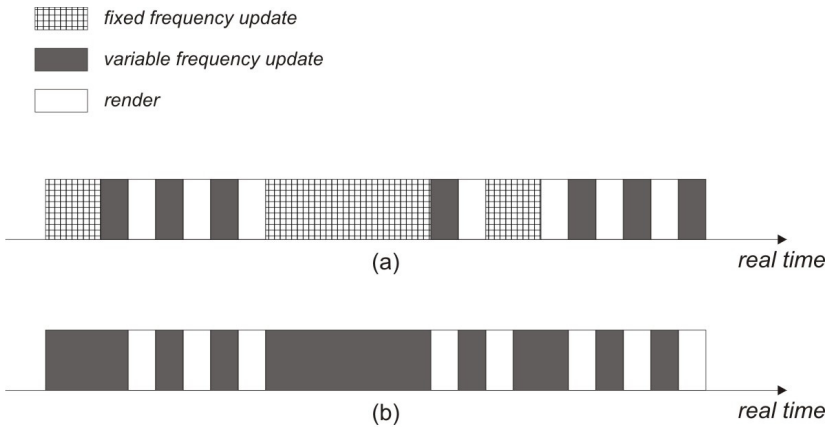

Figure 5.9 – Game loops profiles

This kind of loop forces most of the game logic to be modeled in *discrete time*, which can limit the integration and reuse of simulation models, especially if they work at different time scales as discussed in section 3.2.1. However, with the *discrete event* approach, it is not necessary to have a fixed frequency update

function. The simulation can be advanced by any time period *dt* at any time, reaching a perfectly valid and defined state. Besides, removing the fixed frequency update does not restrict the simulation models. In the context of the Process-DEVS framework, any logic that is modeled in discrete time can be embedded in a process for which $ta(s) = c$, where c is a constant. In this case, even though the update function is called with a variable frequency, that process will be executed as if it was modeled in discrete time.

Since it is not necessary to provide a fixed frequency update function, the loop is considerably simplified. It is only necessary to alternate calls to the update and process_io functions as depicted in Figure 5.9 (b). In this case, the main question is to figure out which parameter *dt* to use in each update call, considering that it is not known in advance how long those calls will take to execute. The next section provides a study on some loop models in order to answer this question.

5.5.2 A Loop Model Study

In order to test different loop models, we shall consider a simulation composed entirely by i processes of the form $P_i[\Delta t_i, wt_i] = \langle S_i, X_i, Y_i, E_i, P_i, \delta_{int\ i}, \delta_{ext\ i}, \lambda_i, \rho_i, ta_i \rangle$. Each process P_i generates events periodically every Δt_i simulation time units. Each event is assumed to take wt_i real time units to be processed. In short, $ta_i(s) = \Delta t_i$ and the process does nothing besides consuming a processing time equal wt_i in its internal transition function. Two processes are defined for the test: $P_1[50ms, 5ms]$ and $P_2[250ms, 100ms]$. These two processes will determine how much processing time each call to the *update* function will take. The *process_io* function is assumed to take a constant time equal to $5ms$ and the desired frame rate for this simulation is $10fps$. The purpose of this test is to study the effects of a high processing load of a simulation model in an interactive simulation. Particularly, the relatively sparse processing peaks generated by P_2 and the exhaustion of the processing resources caused by a speed increase shall be studied in detail. In order to achieve that, the simulation starts normally with *speed* = 1. When the real time reaches $4s$, the speed is increased to 4. When the real time reaches $6s$, the speed returns to 1. When the simulation time reaches $15s$, the simulation is finished.

The first and simplest loop considered in this study is the *MaxFpsLoop*, which is defined as

```
current_time = get_system_time()

while(!is_finished())
{
  last_time = current_time
  current_time = get_system_time()

  update((current_time – last_time) * get_speed())
  process_io()
}
```

This loop is quite simple and useful. It simply measures the real time it took to execute the previous cycle and uses it to feed the update function. Note that it is multiplied by the speed given by the *get_speed* function. For example, if the speed equals 2, the simulation will be advanced twice as fast as the real time.

This loop clearly attempts to maximize the frame rate. The faster the update and process_io functions are executed, the higher the frame rate is. Although loops like this are used in some single player computer games, it has two drawbacks if we consider the serious games requirements discussed in the beginning of this section. First, it always attempts to use all available computational power to increase the frame rate, even in the cases where that will not improve the user experience. Second, it does poorly when trying to run at speeds that surpass the processing limits. In that case, the frame rate drops dramatically and the *dt* parameter of the update function grows indefinitely.

Figure 5.10 depicts the results of the test executed with the *MaxFpsLoop*. The chart on the top shows the evolution of the simulation time with the real time flow. The chart on the bottom shows the frame rate. The frame rate values are calculated using a time window of 0.5*s*. The results show clearly that this kind of loop is inadequate to meet the requirements. First, its frame rate in normal speed is much higher than desired, therefore wasting resources, which might be a problem for the serious games industry, where the games might compete for processing power with other corporative information systems. Second, the frame rate drops almost to zero when the simulation is accelerated beyond the processing capacity. Third, it continues to run fast for some time after the speed has returned to normal at 6*s*. This is because this loop accumulates time debts

during the period where it does not reach the desired speed. After the speed has returned to normal, it attempts to compensate by continuing to execute faster for some time. This is good only for small time debts such as those caused by the processing peaks of P_2. Indeed, the line at the top chart has reached with precision the point (4,4) because of this property. However, if the time debt is large enough so that the time necessary for compensation is perceivable to the user, it should not be compensated. This would give the user a sense of losing control over the simulation speed.

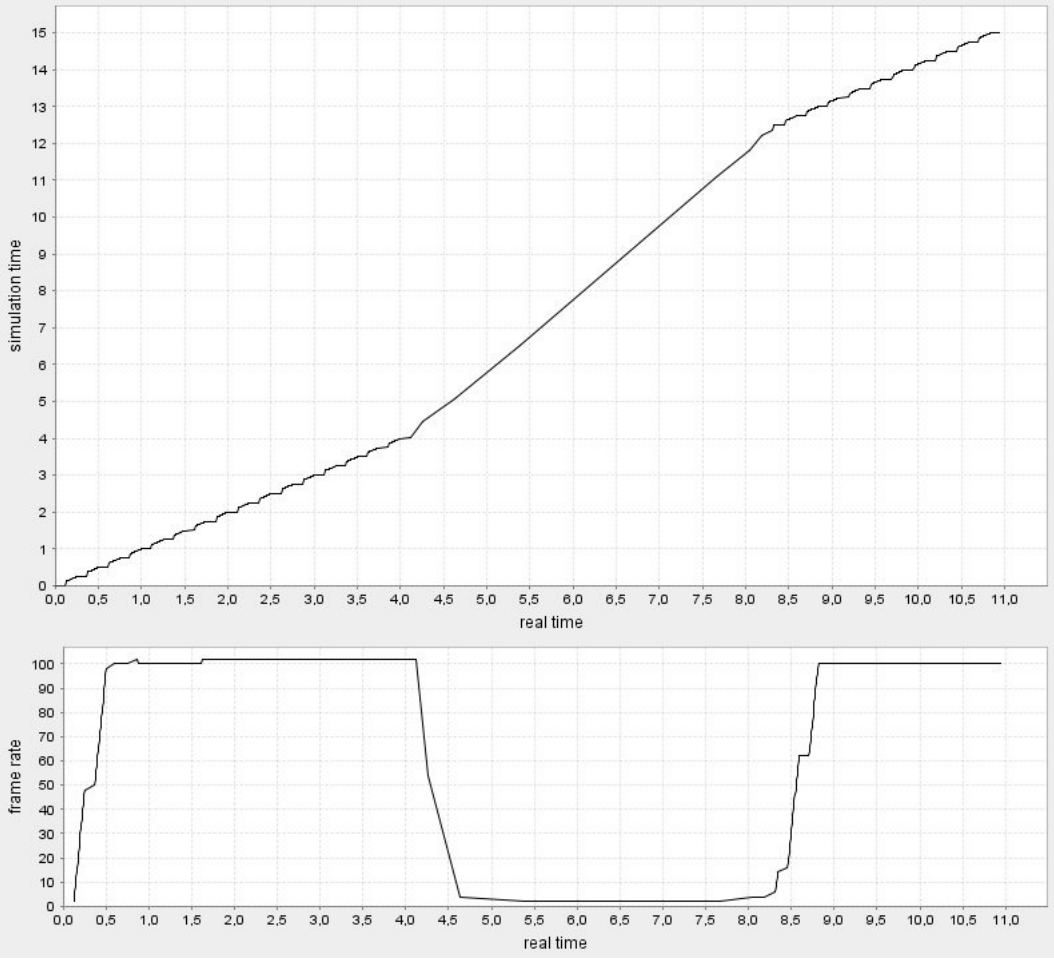

Figure 5.10 – MaxFpsLoop

The *MaxFpsLoop* is based on a catch-up principle. It checks the time it took to execute the last loop and set the next update *dt* parameter accordingly. One alternative also used in computer games is the opposite strategy. Set a fixed *dt* parameter and set the loop time accordingly. This is done by taking the time the

update call took and setting a sleep time accordingly. The *FixedStepLoop* implements this approach in a simple way.

```
parameters(
  desired_frame_rate = 10.0
)

loop_time = 1.0 / desired_frame_rate
last_time = get_system_time()

while(!is_finished())
{
  processIO()
  update(loop_time * get_speed())

  remaining_time = loop_time - (get_system_time()-last_time)

  if(remaining_time > 0)
  {
    sleep(remaining_time)
  }

  last_time = get_system_time()
}
```

This loop clearly expects that there will always be enough processing time to execute the *update* and *process_io* on time to keep the frame rate constant at the desired level. In fact, at normal speed, the frame rate is very well behaved as shown in Figure 5.11. It still drops when the processing capacity is stressed but less than in the *MaxFpsLoop*. One other problem solved by this loop is that it does not accumulate time debts when the processing capacity is reached. This can be easily seen in Figure 5.11. After 6*s*, the speed returns to normal almost immediately.

Although *FixedStepLoop* solves most of the problems of *MaxFpsLoop*, it raises one new problem. Since it does not accumulate time debts, the processing peaks caused by P_2 forces the simulation to go slower than the desired speed, even when there is enough processing capacity. This can be easily checked in the top chart of Figure 5.11. The line does not reach the point (4,4) as expected. This could be an issue in a computer simulation that is mixed with real dynamics, for example.

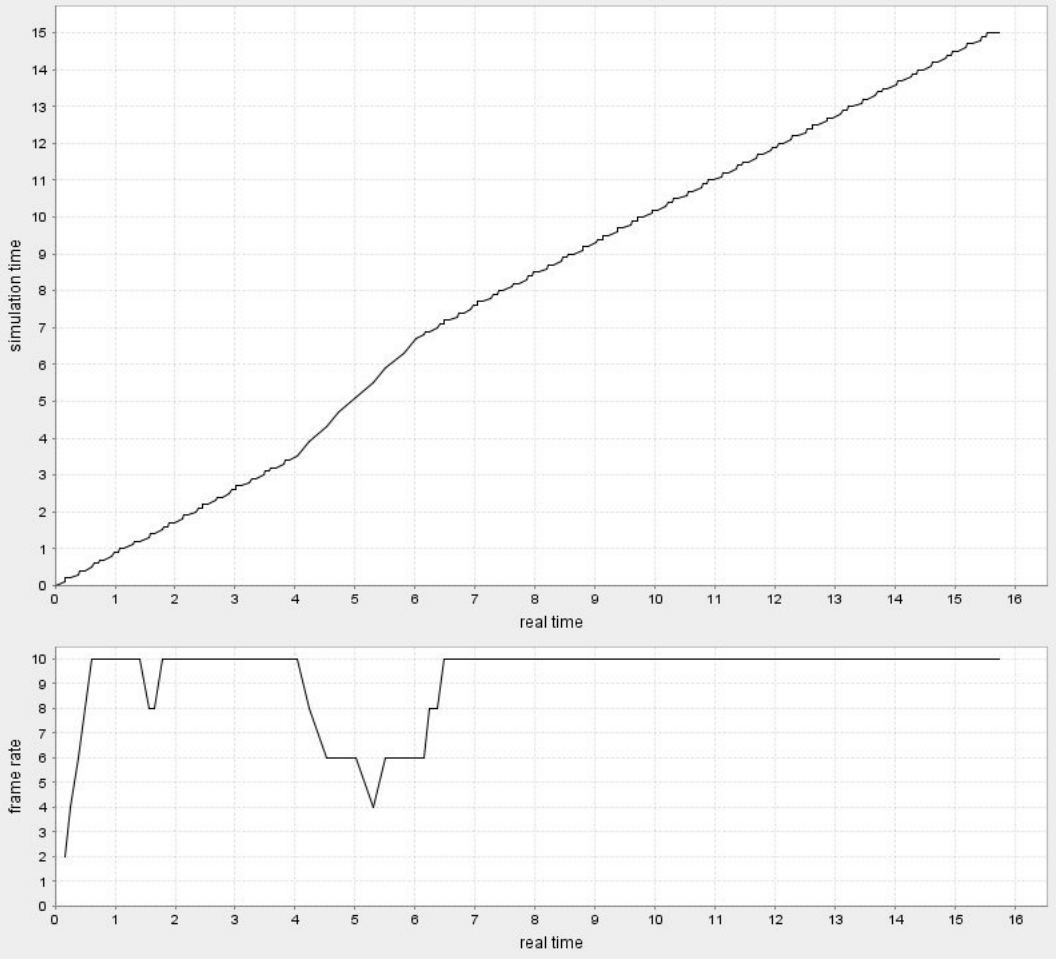

Figure 5.11 – FixedStepLoop

In order to solve the problems raised by these two loop studies, a looping strategy consisting of the following steps was developed:

1. **Update the simulation in small steps until it is time to call *process_io* or all time debts have been paid**. Updating the simulation in small steps is good to detect when the next call to *process_io* is late. If all time debts have been paid, the simulation is up to date and there is no need to update it any further.
2. **If all time debts have been paid, wait for the time to call *process_io***. This is important to release the processing resources if they are not fully needed to achieve the desired frame rate.
3. **Call *process_io***. It is called once per loop. Therefore each loop should ideally last the inverse of the frame rate.

4. **Compute the loop time and increase the time debt for the next loop forgiving all debts beyond a given threshold**. The desired time for executing a loop cycle is determined by the desired frame rate. The debt calculation should consider the difference between the desired and actual loop time.

This loop requires two additional parameters besides the desired frame rate. One for defining the granularity of the steps in which the simulation should be advanced and another for the debt forgiving threshold. The *StableFpsLoop* implements those steps. Its pseudo-code is given below.

```
parameters(
  desired_frame_rate = 10.0
  max_debt_factor = 2.0
  update_granularity = 0.25 //should be between 0.0 and 1.0
)

loop_time = 1.0 / desired_frame_rate
current_time = last_time = get_system_time()
advance_debt = 0.0

while(!is_finished())
{
  advance_debt += loop_time * get_speed()
  advance_step = loop_time * get_speed() * update_granularity

  do
  {
    advance_step = min(advance_step, advance_debt)

    update(advance_step)
    advance_debt -= advance_step

    remaining_time = loop_time-(get_system_time()-last_time)

    //if no more debts, waits until time to call process_io
    if((advance_debt <= 0.0) && (remaining_time > 0.0))
    {
      sleep(remaining_time)
      remaining_time = 0.0
    }

  } while(remaining_time > 0.0)

  processIO()

  current_time = get_system_time()

  //add time difference between desired and actual loop time
  advance_debt +=
    ((current_time - last_time) - loop_time) * get_speed()
```

```
    //forgive debts beyond threshold
    debt_threshold = max_debt_factor * loop_time * get_speed()
    advance_debt = min(advance_debt, debt_threshold)

    last_time = current_time
  }
```

The results of actually running this loop are depicted in Figure 5.12. It can be easily seen that the *StableFpsLoop* behaves better than the previous loop. Like the *MaxFpsLoop*, it is capable of compensating for small processing peaks, keeping the average speed as desired. However, if the simulation keeps a speed beyond the limits of the processing capacity for a large time period, it does not accumulate all the time debt. It is clear that after 6*s*, the speed returns to normal rather quickly.

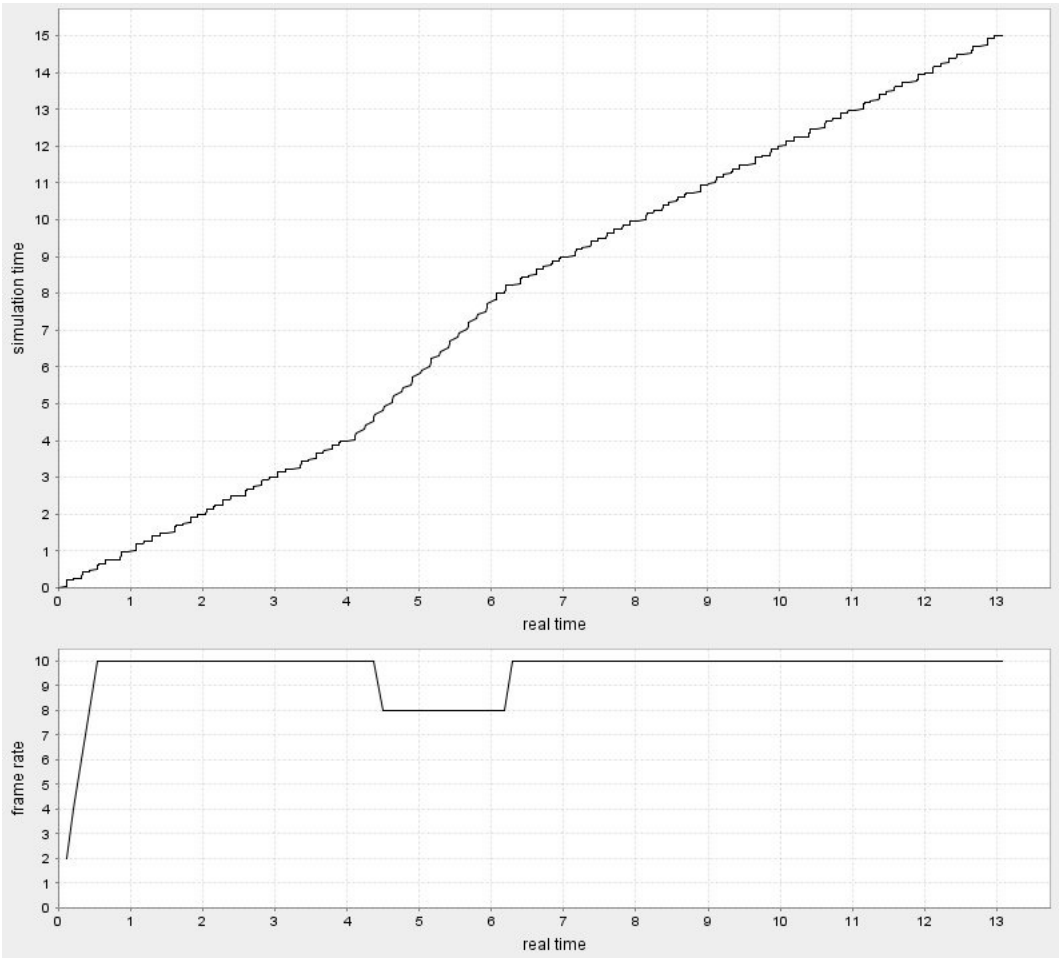

Figure 5.12 – StableFpsLoop

This loop also behaves well with respect to the frame rate. At normal speed, it keeps the frame rate in the desired value. Therefore, it saves as much processing

time as possible for other concurrent applications. The frame rate still drops a little under stressing conditions but the impact is less than in the two previous loop models.

All in all, the *StableFpsLoop* is the first loop model that handled the test case in an acceptable way. The four steps identified for implementing the loop strategy seem to be a good guide to deal with the speed change requirement.

5.6 Summary

This chapter described the implementation of two applications on top of the Process-DEVS formalism, introduced in section 3.3. Both applications are part of the InfoPAE system, which is targeted at managing emergency response in the oil industry. The first application consists of a planning module while the second is a training game.

The first result of the discussion in this chapter is that almost all the simulation model could be successfully reused by the two applications thanks to the high level of modularization. Only a small number of processes and the internal implementation of the environment had to change in order to allow different types of user interaction and to optimize the 3D rendering performance of the plan simulator.

Another result is that processes modeled in different formalisms such as resource displacement, cell space processes and workflows could work well together while keeping them independent of each other. The process of oil dispersion, which was the one with the most complex logic in the simulation, could be easily expressed in terms of the Process-DEVS formalism without encountering any restrictions. The same happened for the workflow operators of the InfoPAE response action plan representation. No limitations were faced with respect to the expressivity of the simulation framework.

As expected, the 3D rendering performance of the plan simulator and the network traffic of the training game did not show significant changes when the simulation speed is increased, even when the processing capacity is reached. The time control techniques described in section 5.5 worked well for that result.

In the case of the plan simulator, even though the users can compose different simulations by defining new scenarios and response plans, they have expressed the desire of defining different specific simulation processes for certain cases. However, programming directly on top of the Process-DEVS formalism would be too difficult for non-programmers. Therefore, just as in the case of SeSam (described in section 2.3.2), it would be nice to provide users with a simpler language on top of Process-DEVS to define processes.

Another desirable feature for the InfoPAE system is to provide the notion of time in its workflow notation. Most workflow representations only allow one to define before-after relationships between actions. Some InfoPAE users expressed the desire to represent more detailed and quantized time relations. Since the Process-DEVS formalism models time explicitly, it would likely be capable of supporting interesting representations of workflows with time.