4
Integrating Existing Formalisms

The previous chapter introduced a framework for modeling interactive simulations based on a discrete-event approach. This chapter shows how to implement some of the common dynamic modeling formalisms on top of this framework. The formalisms were chosen according to the requirements enumerated in section 1.3.

Section 4.1 describes how to define a simulation process from a workflow description. Section 4.2 provides a modular way for modeling processes that act on cell spaces. Section 4.3 explores how to model multi-agent systems. Finally, Section 4.4 extracts some knowledge from these three sections about interesting patterns in which to structure processes in a simulation for greater modularity. Section 4.5 summarizes the chapter.

4.1
Workflows

As it was mentioned in section 2.4, action plans are usually represented as workflows in AI planning, which is increasingly being adopted in game AI, not only for serious games but also for entertainment games (Nareyek 2004). Therefore, it would be interesting to be able to simulate workflows on the Process-DEVS formalism thereby allowing the use of automatic planners within the simulation logic.

Additionally, most of the so-called Business Process Management (BPM) systems represent business processes as workflows (Weske 2007). In fact, business process simulation (BPS) provides a more powerful way of analyzing the performance of business processes than static analysis tools and methods (Tumay 1996; Modarres 2006). Therefore, being able to simulate workflows would allow serious games based on Process-DEVS to participate in the optimization and reengineering of business processes.

Since this work is focused on the requirements of serious games which are not so common in entertainment games, the ability to simulate business processes will be used as the main motivation.

4.1.1 Motivation: Business Process Modeling

In business administration, almost all kinds of organizations have developed the need for formally expressing their activities. In fact, standardization of business processes becomes crucial for organizations to keep control of what is happening as they become larger and more complex. In order to meet that goal, the adoption of the so-called Business Process Management (BPM) systems has grown significantly over the years (Weske 2007).

Testing the quality of business processes and the performance of the teams responsible for executing them is important to ensure efficiency. Beside other initiatives, such as field exercises, the use of computational simulation can be a cost-effective and efficient mechanism to help testing, validating, improving and reengineering business processes. Simulation can be used for different purposes such as estimating, in advance, if there will be enough resources and time for executing a specific action or supporting complex decisions when there are too many possibilities. One additional benefit of simulation is the possibility of simulating how the environment and other entities will respond to the execution of the business process. As an example, it is extremely important to anticipate the behavior of physical phenomena, such as the dispersion of leaked chemical products in the environment, considering a business process to handle this kind of emergency scenario.

In the context of computer serious games focused on training, it is also important to evaluate whether a given player has taken the proper decisions during play. Therefore, for this kind of player performance evaluation, it seems natural to model player activity as workflows. This would help comparing his actions with predefined business processes or action plans considered as the right way to handle the situation.

One other direct benefit of integrating workflows with simulation is the possibility of detecting flaws in established business processes and help improving them. In fact, simulation could be integrated in a cyclic way with the business planning process as illustrated in Figure 4.1.

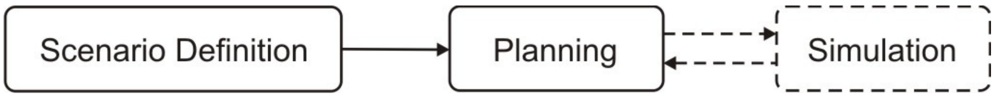

Figure 4.1 – The role of simulation in the planning process

In the context of this work, integrating workflows with other simulation formalisms mean to model workflows on top of Process-DEVS, introduced in chapter 3. Since previous research work has suggested that discrete-event simulation is the most adequate tool for simulating business processes (Tumay 1996), Process-DEVS should be adequate for the task.

Since there are numerous formalisms for workflow representation (van der Aalst 2003; Weske 2007), it is necessary to select one first.

4.1.2
A Discussion on Workflow Representation

Since there are many different languages and representations for workflows, this section starts by describing the workflow representation used here.

A workflow is essentially defined by a set of actions and a control structure. The actions define what should be done and the control structure defines in which order the actions should be executed in a given situation (van der Aalst 2003). The control structure is usually defined in the form of a graph. Although some representations restrict this form to a tree structure, this is clearly a specific case of the graph structure. Therefore, for the sake of generality, we shall represent workflows as graphs. The nodes of the graph represent either an action or a control flow pattern. The most common types of patterns are splits and joins, in various flavors (van der Aalst 2003). The edges of the graph are connections that inform, for any given node, which nodes should be triggered next when its execution finishes. There are many different workflow representations which differ from each other in some of the patterns they allow. For the sake of simplicity, we shall consider only the five basic patterns defined in (van der Aalst et al. 2003): *sequence*, *parallel split*, *synchronization*, *exclusive choice* and *simple merge*. Figure 4.2 shows a very simplified version of a contingency plan for a situation where some oil has leaked into the sea. This example is composed with these five basic patterns.

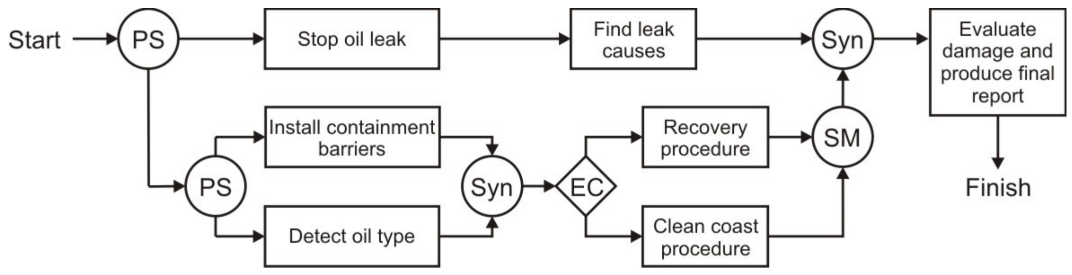

Figure 4.2 – Workflow for an oil-leak situation with the five basic patterns

At the start of the plan, there are two parallel split nodes (PS), meaning that the actions of stopping the leak, installing containment barriers and detecting the oil type should be started in parallel. After the leak has been stopped, the sequence pattern states that the action of finding the causes should be started. After the actions of installing the barrier and detecting the oil type have finished, the workflow reaches a synchronization point (Syn). That means that both actions must finish before the workflow execution can continue through that path. After both actions have finished, the exclusive choice (EC) pattern queries the environment state to find out whether the barrier actually prevented the oil from reaching the coast. If the oil has not reached the coast, the recovery (of the oil from the sea) procedure is started. Otherwise, a cleaning (of the oil from the) coast procedure should be executed. Either the recovery procedure or the coast cleaning action will be executed, after which the simple merge (SM) pattern allows the workflow to continue through its outgoing path. After both the proper procedure has been executed and the causes of the leak have been found, a final report is produced. In order to guarantee that these two preconditions are met, there is a synchronization point right before the final report production action.

Both actions and each of the basic patterns, except for the sequence pattern, are nodes in the workflow graph. Each node connects some *incoming nodes* to some *outgoing nodes*. When a node finishes its execution, it may trigger some of its following nodes. Likewise, a node is triggered only when some incoming node has finished its execution. Table 4.1 lists some of the characteristics of each basic pattern.

pattern	number of incoming nodes	number of outgoing nodes	trigger condition	following activity
sequence	1	1	completion of incoming node	trigger outgoing node
parallel split	1	n	completion of incoming node	trigger all outgoing nodes
synchronization	N	1	completion of all incoming nodes	trigger outgoing node
exclusive choice	1	2 (or n)	completion of incoming node	trigger one of the outgoing nodes
simple merge	N	1	completion of any incoming node	trigger outgoing node

Table 4.1 – Characteristics of basic workflow patterns

The number of incoming and outgoing nodes indicates the number of incoming and outgoing connections each pattern may have. The trigger condition indicates the condition upon which the pattern is triggered. Finally, the following activity describes which of the outgoing nodes should be triggered after the pattern is triggered.

Optionally, the parameters or inputs of the individual actions are also represented. Likewise, an action may also produce some data as output. That data could be consumed either by another action executed after it or by some conditional split operator in the control flow. Therefore, in order to represent action input and output, a data flow may also be represented. Note that the data flow does not follow the same paths as the control flow. However, it should obviously obey the ordering restrictions imposed by the control flow, since an action cannot consume output data from another action that has not yet been executed.

The simplest way to model the data flow is to define an environment state which is accessible from the workflow process and its actions. Each time an action or a control operator needs some input data, it can get it from this environment state. Likewise, when an action produces some data, it should store it in the environment state so that later actions can read it. Hence, any information stored in the environment state can be used by the exclusive choice operators to evaluate their conditions when they are triggered. Environment states are most commonly defined as a set of variables. Figure 4.3 depicts a workflow with an environment state of this kind. It shows actions writing and reading from variables and one exclusive choice operator reading from them.

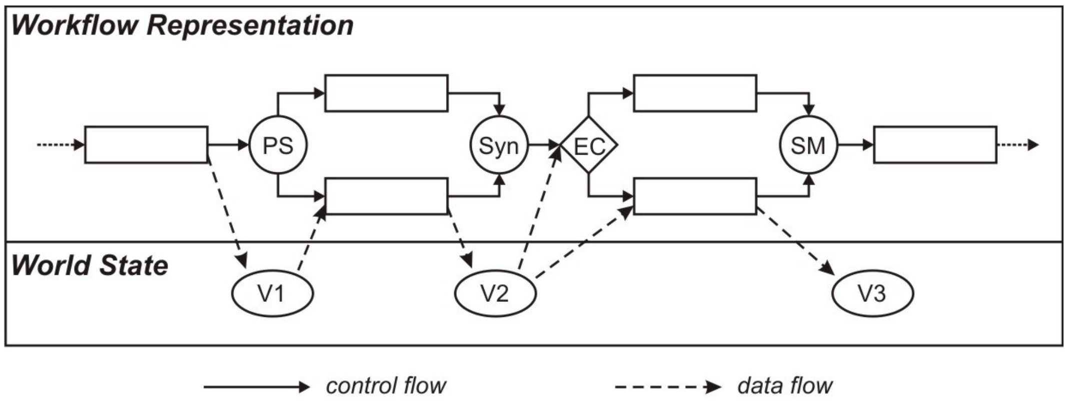

Figure 4.3 – Workflow with an environment state defined by the variables v1, v2 and v3

Most workflow representations do not define the time at which the actions should be executed, and how long they will take to finish. In fact, many representations assume that actions are atomic.

Workflows do not necessarily define how actions affect the environment. One approach is to define actions through their pre- and post-conditions, following the tradition of AI planning systems (Fikes and Nilsson 1971). But note that this representation still assumes actions to be atomic. However, the assumption that actions are atomic may be too restrictive. For example, consider the action of walking. It may not be realistic to change the position of the character from the origin to its destination in one single instantaneous step. Instead, it is more realistic to simulate the trajectory of the character to the destination point through multiple state changes, so that his trajectory may be observed. If the environment model requires that actions have duration and make changes to the environment during their executions, it is necessary to model actions as processes in time. For this purpose, the definition of a process in section 3.3 welcomes in hand. In fact, process-oriented simulation, on which Process-DEVS is based, provides an excellent basis for simulating workflows. Some previous work based on object-oriented simulation had to extend the basic formalism to accommodate workflows (Wagner et al. 2009).

Modeling workflow actions as processes makes the workflow itself a form of process composition. Going one step further, the whole workflow may also itself be represented as a process. Since the Process-DEVS framework allows processes to fork children processes during their execution, the *workflow process* can be modeled as a process which orchestrates the execution of its children processes, namely the *action processes*. Figure 4.4 depicts this process structure.

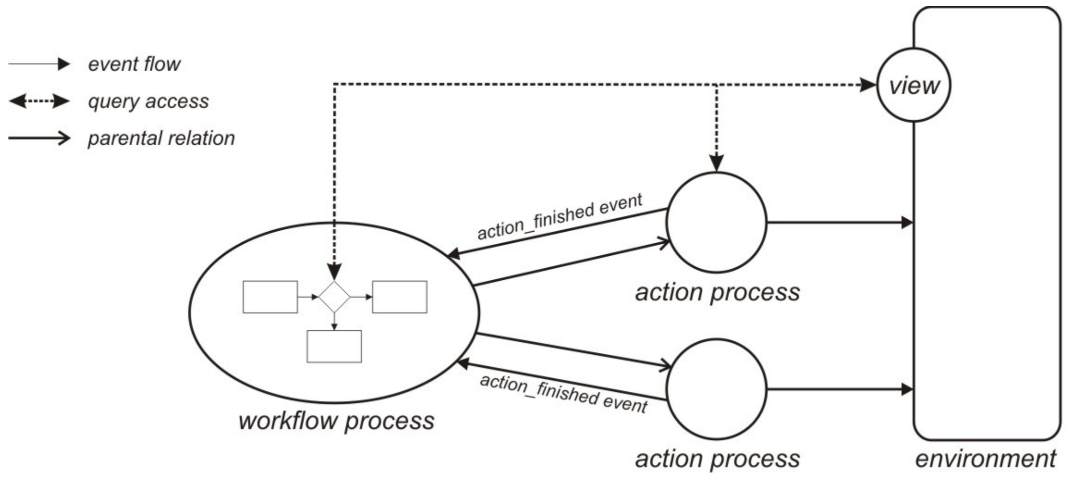

Figure 4.4 – Workflow and action processes

The workflow process needs to be informed when an action process is finished executing its action, so that it can continue with the workflow execution. In order to implement that, all action processes must output an *action_finished* event, informing the workflow process when they are finished.

By taking advantage that Process-DEVS also provides the notion of environment, it will be naturally used to represent the notion of environment state of the workflow processes. A specific environment view is provided by the environment to serve as the environment state, as the workflow process can perceive it. The action processes may, but are not limited to, perceive the environment through the same view. As any other kind of process, the action processes change act on the environment by sending events to it.

This way of representing workflows has two main advantages:

- By separating the workflow control logic from the execution of actions, a higher level of modularity is achieved.

- Representing both workflows and actions as processes allows hierarchical workflow composition. A workflow action may be a sub-workflow.

4.1.3 A Formal Workflow Model

This section introduces a formal workflow model in three stages. First, it introduces the simpler notion of workflow. Then, it defines the notion of execution states. When a workflow is executed, it produces a sequence of *execution states* as its nodes are triggered. Finally, it defines the notion of *workflow process* as a process in the sense of section 3.3.1.

Workflow Definition

A *workflow* is defined as a tuple $WF = \langle A, ES, N, E, entry, exit, type \rangle$, where

(1) A is the set of *action processes* that WF can execute
(2) ES is the set of *environment states* that WF can perceive
(3) N is the set of nodes of the *workflow graph* of WF
(4) $E \subseteq N \times N$ is the set of edges of the workflow graph of WF
(5) $entry \in N$ is the entry point of WF
(6) $exit \in N$ is the exit point of WF
(7) $type: N \rightarrow T$ is a function that assigns to each node n in N a *node type* in T

The set A define the actions WF can execute. These actions are actually action processes that are forked by the workflow process when an action is executed.

The situation in which the workflow process is embedded is the simulation environment, as defined in section 3.3.1. The set ES defines the states in which the workflow process can perceive the environment. The environment state is queried, for example, in the *exclusive choice* pattern (van der Aalst 2003), to determine which actions are executed next.

The workflow control structure is defined by a graph, whose vertices are nodes that belong to the set *N* and whose edges belong to the set *E*. The entry point of the workflow is the node in which the execution starts. Likewise, the exit point is the node where it finishes.

Two operators are defined to access the incoming and outgoing nodes of any given node in the workflow graph:

(8) $outgoing\_nodes(n) = \{ m \in N \mid (n, m) \in E \}$

(9) $incoming\_nodes(n) = \{ m \in N \mid (m, n) \in E \}$

Given a node $n \in N$, *type(n)* assumes one of the following values:

(10) *action(ap)*, where $ap \in A$, associates action process *ap* with node *n*. In this case, *n* is called an *action node* of *WF*. There must not be two different action nodes with the same action process:

$$(\forall n_1, n_2 \in N)\ (type(n_1) = type(n_2) = action(ap) \Rightarrow n_1 = n_2)$$

(11) *parallel_split*, which indicates that *n* is a *parallel split node* of *WF*.

(12) *syncronization*, which indicates that *n* is a *synchronization join node* of *WF*.

(13) *exclusive_choice(*ϕ*)*, which indicates that *n* is a *parallel split node* of *WF*, with choice function ϕ: $ES \rightarrow outgoing\_nodes(n)$.

(14) *simple_merge*, which indicates that *n* is a *simple merge join node* of *WF*.

As defined above, the workflow graph may have an arbitrary structure, which poses considerable difficulties when it comes to defining the notion of workflow process and its operational semantics. We therefore introduce the concept of well-formed workflows and, at the same time, a convenient notation to express them.

Let *A* be a set of action processes. The set of *well-formed workflow programs over A* and the set of *well-formed workflows over A* are inductively defined as follows:

(15) An action process a in A is a well-formed workflow program over A that defines the well-formed workflow

$WF = \langle\{act\}, ES, \{act\}, \emptyset, act, act, t\rangle$

where $t(act) = action(act)$

In the next definitions, let wf_1 and wf_2 be two well-formed workflow programs and let $WF_1 = \langle A_1, ES_1, N_1, E_1, entry_1, exit_1, type_1\rangle$ and $WF_2 = \langle A_2, ES_2, N_2, E_2, entry_2, exit_2, type_2\rangle$ be the well-formed workflows they define. Assume that $ES_1=ES_2$, that is, WF_1 and WF_2 have the same set of environment states. Define $ES=ES_1=ES_2$. Then:

(16) $wf_1 \rightarrow wf_2$ is a well-formed workflow program that defines the well-formed workflow

$WF = \langle A_1 \cup A_2, ES, N_1 \cup N_2, E_1 \cup E_2 \cup \{(exit_1, entry_2)\}, entry_1, exit_2, t\rangle$

where $t(n) = type_1(n)$ if $n \in N_1$

$= type_2(n)$ if $n \in N_2$

(17) $wf_1 \,//\, wf_2$ is a well-formed workflow program that defines the well-formed workflow

$WF = \langle A_1 \cup A_2, ES, N_1 \cup N_2 \cup \{sp, syn\}, E_1 \cup E_2 \cup \{(sp, entry_1), (sp, entry_2), (exit_1, syn), (exit_2, syn)\}, sp, syn, t\rangle$

where

$t(sp) = parallel\_split$

$t(syn) = synchronization$

$t(n) = type_1(n)$ if $n \in N_1$

$= type_2(n)$ if $n \in N_2$

(18) Let Φ: $ES \rightarrow \{true, false\}$ be a choice function on ES. Then, $\Phi\ ?\ wf_1 : wf_2$ is a well-formed workflow program that defines the well-formed workflow

$WF = \langle A_1 \cup A_2, ES, N_1 \cup N_2 \cup \{ec, sm\}, E_1 \cup E_2 \cup \{(ec, entry_1), (ec, entry_2), (exit_1, sm), (exit_2, sm)\}, ec, sm, t\rangle$

where

$t(ec) = exclusive\_choice(\phi)$, where ϕ: $ES \rightarrow \{first_1, first_2\}$ and

$$\begin{aligned} \phi(es) &= first_1 && \text{if } \Phi(es) = true \\ &= first_2 && \text{if } \Phi(es) = false \\ t(sm) &= simple\_merge && \\ t(n) &= type_1(n) && \text{if } n \in N_1 \\ &= type_2(n) && \text{if } n \in N_2 \end{aligned}$$

The well-formed workflows are informally depicted in Figure 4.5. Their entry and exit nodes are indicated by entering and leaving arrows respectively.

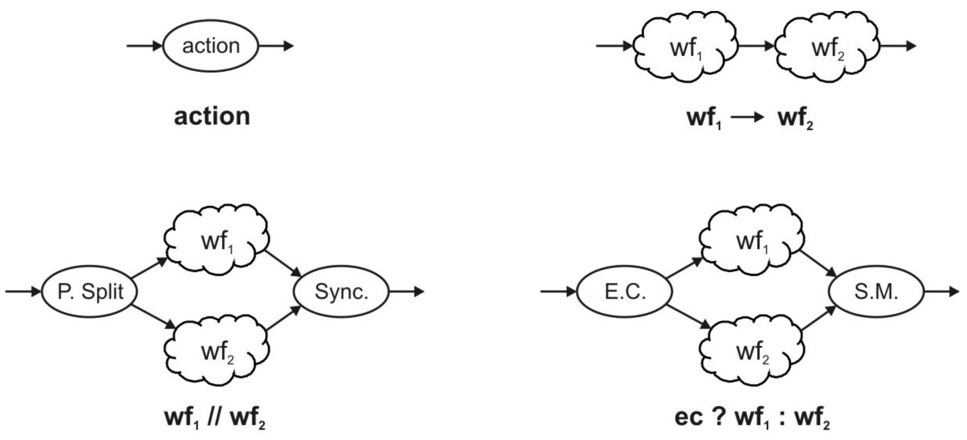

Figure 4.5 – Workflow definition operators and their graphical representation

As an example, the well-formed workflow illustrated in Figure 4.2 is defined by the expression:

(("Stop Oil Leak" → "Find Leak Causes") // (("Install Barriers" // "Detect Oil Type") → ("If Oil is Contained" ? "Recovery Procedure" : "Clean Coast Procedure"))) → "Evaluate Damage ..."

A well-formed workflow has the property informally stated as follows: no node in the workflow is reached by more than one execution thread, with the exception, of course, of *synchronization* nodes. This is easily verifiable because the only kind of node that forks execution threads is the *parallel split*, which is only produced by the parallel sub-graph composition operator, which always put a *synchronization* node where the two threads meet. This is necessary because it is an assumption of the simple merge pattern that none of its incoming branches is ever executed in parallel. Also, most workflow systems do not allow multiple concurrent execution instances of the same action (van der Aalst 2003).

From now on, we assume that all workflows are well-formed.

Workflow Execution States

Before defining the workflow process in the format of the Process-DEVS formalism, we shall define first the notion of *workflow execution states*, which will serve as basis for the definition of the workflow process.

The execution of a workflow is formally defined as a sequence of *execution states* and may yield different results according to the execution environment, which we shall refer to simply as *the environment*. During workflow execution, the environment may be altered by an external process. Therefore, the current environment state is also part of the workflow execution state.

Let $WF = \langle A, ES, N, E, entry, exit, type \rangle$ be a workflow. An *execution state* of WF is a triple $WS = \langle A_{exec}, Syn_{state}, \omega \rangle$, where

$A_{exec} \subseteq A$

is a set that contains all action nodes that are in execution in WS.

$Syn_{state}: Syncs \rightarrow 2^N$

is a function that defines the internal states of all *synchronization* nodes, where

$Syncs = \{ s \in N \mid type(s) = synchronization \}$ is such that

$(\forall s \in Syncs)\ (Syn_{state}(s) \subseteq incoming\_nodes(s))$

$\omega \in ES \cup \{not\_started\}$

is the environment state of WS.

The set A_{exec} keeps all action nodes whose corresponding processes have been forked but have not finished yet. Syn_{state} is responsible for informing the internal state of all *synchronization* nodes. These nodes need to keep an internal state because they only trigger their outgoing node when all of their incoming nodes have finished executing. Therefore, their internal state consists of a subset of their incoming nodes, informing which ones have already finished executing. This makes it possible to know precisely at which execution states the outgoing node of a synchronization node is triggered. Finally, ω is the current environment state, which is used to determine the right outgoing path of an exclusive choice node when it is triggered. The special value *not_started* is used in the initial execution state, when the workflow process has not yet started.

The *initial execution state* of *WF* is a state of the form $WS_0 = \langle \emptyset, Syn_0, not\_started \rangle$, where $Syn_0(s) = \emptyset$, for any synchronization node *s* of *WF*.

The execution state of a workflow changes when one of its nodes is triggered. When a node in a workflow is triggered, it may cause other subsequent nodes to be triggered in cascade. As an example, triggering a *parallel split* node causes its outgoing nodes to be triggered. Once the cascade of node triggering has finished, the workflow reaches a new execution state. The whole cascade of triggers fired by the initial node trigger is considered atomic and characterizes one single execution state transition.

In what follows, let $WF = \langle A, ES, N, E, entry, exit, type \rangle$ be a workflow and $WS = \langle A_{exec}, Syn_{state}, \omega \rangle$ be an execution state of *WF*.

The function $trigger\colon \boldsymbol{WS} \times N \times (N \cup \{nil\}) \rightarrow \boldsymbol{WS}$, where $\boldsymbol{WS}$ is the set of all states of *WF*, formalizes the effects of triggering a node. The function is recursive to represent the triggering cascade. Intuitively, if $trigger(WS, n, n_{prev}) = WS_{next}$, then *n* represents the node that is being triggered, n_{prev} is the incoming node that caused the trigger and WS_{next} is the next execution state. Note that n_{prev} may assume the special value *nil*. That happens when *n* is the entry point of the workflow and therefore has no incoming nodes.

The function *trigger* is defined according to type of *n* (note that equations (21) and (23) assume a specific cardinality of *outgoing_nodes(n)*, which is guaranteed by the constraints imposed on the workflow graph structure):

(19) If $type(n) = action(ap)$, then

$$trigger(WS, n, n_{prev}) = \langle A_{exec} \cup \{ap\}, Syn_{state}, \omega \rangle$$

(20) If $type(n) = parallel\_split$, then

$$trigger(WS, n, n_{prev}) = trigger(\ldots trigger(trigger(WS, n_1, n), n_2, n)\ldots, n_n, n)$$

where $outgoing\_nodes(n) = \{n_1, n_2, \ldots, n_n\}$

(21) If $type(n) = synchronization$, then

$$
\begin{aligned}
trigger(WS, n, n_{prev}) &= WS && \text{if } n = exit \\
&= chg\_sync\_state(WS, n, \psi) && \text{if } n \neq exit \wedge \psi \neq incoming\_nodes(n) \\
&= trigger(chg\_sync\_state(WS, n, \varnothing), n_{next}, n) && \text{otherwise}
\end{aligned}
$$

where

$$chg\_sync\_state(WS, n, s) = \langle A_{exec}, Syn_{state} \leftarrow (n, s), \omega \rangle$$

$$\psi = Syn_{state}(n) \cup \{n_{prev}\}$$

$$outgoing\_nodes(n) = \{n_{next}\}$$

(22) If $type(n) = exclusive\_choice(\phi)$, then

$$trigger(WS, n, n_{prev}) = trigger(WS, \phi(\omega), n)$$

(23) If $type(n) = simple\_merge$, then

$$
\begin{aligned}
trigger(WS, n, n_{prev}) &= WS && \text{if } n = exit \\
&= trigger(WS, n_{next}, n) && \text{if } n \neq exit
\end{aligned}
$$

where $outgoing\_nodes(n) = \{n_{next}\}$

Now that the semantics of node triggering is defined, it is possible to specify how a workflow is executed in the discrete-event simulation environment described in section 3.3.

Workflow Process

The *workflow process* keeps track of the workflow's execution state and forks action processes in order to simulate the actions described in the workflow. Therefore, the responsibility of the workflow process is to determine the time each action should be executed, according to the workflow control logic. The actual execution of the actions is delegated to the action processes, which, in turn, have the responsibility of notifying the workflow process of the exact time they finish their execution by sending an *action_finished* event to it.

When the workflow process is notified about the completion of an action, it computes the next execution state of the workflow and forks the corresponding action processes if some action was started by this execution state transition. These execution state transitions occur instantaneously with respect to simulation time. Therefore, it is assumed that an action starts at the same time instant its previous action finished. Just before computing a transition, the workflow process needs to update its internal perception of the environment state, so that it always considers the right environment state when computing a transition. When a transition produces an execution state *WS* with $A_{exec} = \varnothing$, the workflow process is finished. In this situation, there are no more executing actions and, therefore, no more actions will be started because the workflow process will not receive any more *action_finished* events that could possibly trigger them.

Let $WF = \langle A, ES, N, E, entry, exit, type \rangle$ be a workflow. Using the formalism introduced in section 3.3.1, the *workflow process* for *WF* is the tuple

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

where

(24) $S = \boldsymbol{WS} \times 2^A$, where $\boldsymbol{WS}$ is the set of all possible execution states of *WF*

The internal state of the workflow process has the form *(ws*, *new_acts)*, where *ws* is the current execution state and *new_acts* is the set of actions started in the last execution state transition.

(25) $X = \{action\_finished(ap) \mid ap \in A\}$ is the set of input events

(26) $Y = \{action\_finished(WP)\}$ is the (unitary) set of output events

(27) $E = ES$

(28) $P = A$

The definitions of these components are straightforward. The workflow process can receive events of the form *action_finished(ap)* from its children processes when they finish executing their actions. Likewise, considering that this workflow process can be a child of another workflow process, it should send an event of the same type when the workflow has finished its execution. The environment view of the workflow process is defined by the set of environment states that the workflow can perceive. Finally, the set of possible children processes is defined by the set of workflow actions.

The internal transition function δ_{int} is defined as follows:

(29) For each state $WS = \langle A_{exec}, Syn_{state}, \omega\rangle$ of *WF*,

$$\delta_{\text{int}}((WS, out), env) = finished \quad \text{if } \omega \neq not\_started$$
$$= (WS', A'_{exec}) \quad \text{if } \omega = not\_started$$

where $WS' = \langle A'_{exec}, Syn'_{state}, \omega'\rangle$ is the execution state of *WF* such that $WS' = trigger(\langle A_{exec}, Syn_{state}, env\rangle, entry, nil)$

Recalling the operational semantics defined in section 3.3.2, the internal transition function δint is called at the time a process is started. This function defines the first execution state transition of *WP*. It triggers the entry node of the workflow. When δ_{int} is called for the second time, which is characterized by $\omega \neq$ *not_started*, it finishes the process. During all workflow execution, only the external transition function δ_{ext} is used.

The external transition function δ_{ext} is defined as follows:

(30) For each state $WS = \langle A_{exec}, Syn_{state}, \omega\rangle$ of *WF*,

$$\delta_{\text{ext}}(((WS, out), e), env, action\_finished(ap)) = (WS', A'_{exec} - A_{exec})$$

where

$$WS' = (\langle A_{exec} - \{ap\}, Syn_{state}, env\rangle, \varnothing) \quad \text{if } ap = exit$$
$$= trigger(\langle A_{exec} - \{ap\}, Syn_{state}, env\rangle, n_{next}, ap) \quad \text{if } ap \neq exit$$

where $\{n_{next}\} = outgoing\_nodes(ap)$

This function is called when *WP* receives an *action_finished(ap)* event. When that happens, the action that has just finished is removed from the set of executing actions in *WS* and the environment state is updated to the current value *env*. If *ap* is not the exit point of the workflow, its outgoing node is triggered. Besides changing the execution state of the workflow, this node triggering may also cause one or more actions to start execution. The set of started actions is found by subtracting the sets of executing actions of the next execution state WS_{next} from *WS*.

(31) For each state $WS = \langle A_{exec}, Syn_{state}, \omega \rangle$ of WF,

$$\lambda((WS, new\_acts)) = \{action\_finished(WP)\} \quad \text{if } A_{exec} = \varnothing$$
$$= \varnothing \quad \text{if } A_{exec} \neq \varnothing$$

This function makes the workflow process send the *action_finished(WP)* event when the set of currently executing actions becomes empty, which is the condition for finishing the workflow process.

(32) For each state $WS = \langle A_{exec}, Syn_{state}, \omega \rangle$ of WF,

$$\rho((WS, new\_acts)) = (new\_acts, \varnothing)$$

The action processes corresponding to the actions started in the last execution state transition are forked.

(33) For each state $WS = \langle A_{exec}, Syn_{state}, \omega \rangle$ of WF,

$$ta((WS, new\_acts)) = \infty \quad \text{if } A_{exec} \neq \varnothing$$
$$= 0 \quad \text{if } A_{exec} = \varnothing$$

This function states that the internal transition function δ_{int} should be called for the second time (the first time is at the start of *WP*) only when the workflow has reached its finish condition $A_{exec} = \varnothing$.

This process models all the workflow control logic. When the workflow process is started, the δ_{int} function (definition (29)) is invoked. Then, as the workflow executes, the δ_{ext} function (definition (30)) is invoked multiple times until the workflow has finished its execution. Each of those calls produces a new execution state. The ρ function (definition (32)) informs which action processes are forked after each execution state transition. When the workflow has finished its execution, the δ_{int} function is invoked again to finish the process and the λ function (definition (31)) outputs the event *action_finished(WP)*. The *time-advance* function (definition (33)) assures that the δ_{int} function is only invoked for the second time when the workflow execution has finished (i.e. when $A_{exec} = \varnothing$).

4.1.4 Workflow Composition

As defined in the last section, a workflow is a form of process composition, where the definitions of multiple processes are combined into the definition of one larger process, namely the workflow process.

As a consequence of the way workflows were modeled, it is trivial to compose a workflow with sub-workflows. Since a workflow process is a process, it can be used as an action process of another workflow, just like any other kind of process. Hence, it is possible to compose workflows hierarchically. Note that, in order to allow this form of composition, it is essential that the workflow process outputs an *action_finished* event when the workflow execution has finished. In fact, its *output function* (definition (31)) does precisely that.

4.2 Cell Space Processes

It was mentioned in section 2.2.2 that cellular automata (CA) have been extensively used to model the dynamics of anthropic and natural phenomena. A large variety of those models can be found in the GIS literature. The ability to execute such models in a serious game framework goes in the line of integrating different formalisms and giving these games well founded simulational realism.

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). A *cell space* is a space representation where the space is partitioned in a discrete set of cells. A *cell* is an atomic unit of space which has a unique state at any given time. The idea of cell spaces is to provide a discrete space representation for modeling dynamic spatial phenomena.

4.2.1 The Modularity Problem of Cellular Automata

Recalling the definition presented in section 2.2.2, a *CA* is defined by a tuple $\langle C, S, N, T\rangle$, where C is the cell set, S is the state set, N is the neighborhood function and T is the state transition function. This monolithic structure contains both the representation of space and the representation of a dynamic phenomenon that happens in that space.

Despite its simplicity, this way of representing dynamic processes on cell spaces presents some limitations. Particularly, it does not allow external processes to interfere with it. For example, imagine a *CA* that models the dispersion of oil leaked into the sea. If a containment barrier is installed, it must interfere with the dispersion process. In order to model this phenomenon with the strict *CA* formalism, one has to combine both the logic of dispersion and the logic of containment in the transition function of the *CA*. This makes the whole process monolithic and therefore hurts the modularity and reusability of the model. The notion of cell space models (Batty 2005), although more flexible than strict *CA*, still is not capable of addressing that problem.

In order to accomplish that kind of modularity, it is necessary to break the logic of state transitions of cells. For this purpose, the principles of process-oriented simulation discussed in section 3.2.4 come in handy. The next section describes how it can help solving the modularity problem in the context of the Process-DEVS formalism.

4.2.2 Separating Behavior from Cell Space

In order to model cell space phenomena in Process-DEVS, it is necessary to break the monolithic representation of cellular automata into two parts: the *cell space (CS)* and the *cell space process (CSP).* The *CS* represents the physical aspect of the *CA* and is, therefore, modeled as part of the simulation environment. The *CSP* represents the behavioral aspect of the *CA* and is modeled as a process in the sense of section 3.3.1. The idea is that the *CSP* periodically perceives the state of the *CS* through an environment view and generates one or more events manifesting its intentions of changing the *CS* state. When these events reach the environment, they cause a state transition in the *CS*. This procedure is depicted in Figure 4.6.

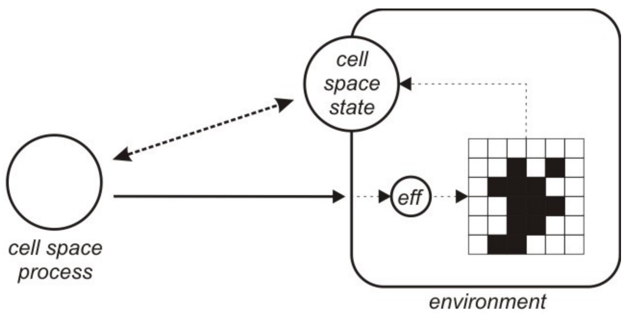

Figure 4.6 – Breaking a CA into physical state (environment) and behavior (process)

The cell space is formally defined as

$$CS = \langle C, S, I, eff \rangle$$

where

C is the set of cells

S is the set of cell states

I is the *intention set*

eff: $\Phi \times 2^{(C \times I)} \rightarrow \Phi$ is the *effect function*, where $\Phi = \{\ \varphi: C \rightarrow S\ \}$

The set C contains all cells from the CS. At each point in time, every cell must have a definite state from the set S. The *cell space state* is the state of the entire CS, and it is defined by a function φ: $C \rightarrow S$. For each cell $c \in C$, its state is given by $\varphi(c)$. The set Φ, containing all possible cell space states, defines the environment view of the CSP, which is the way the CSP perceives the CS.

The I and eff properties define the way the CSP acts on the CS. The set I defines the possible intentions the CSP can manifest for any given cell. Each time the CSP intends to change the CS state, it manifests its intentions by sending an event $its \in 2^{(C \times I)}$. That event consists of a set of pairs (c, i) in $C \times I$. Each pair indicates an intention i for a cell c. Once the its event reaches the environment, it causes a state transition on the CS. The effect function defines the next cell space state as $eff(\varphi_{curr}, its)$, where φ_{curr} is the current cell space state.

Let $CS = \langle C, S, I, eff \rangle$ be a cell space. Let N, B, Δt be *cell space process parameters*, where

N: $C \rightarrow C^{|N|}$, where $|N|$ is the neighborhood size and $(\forall c \in C)(c \notin N(c))$ is the *neighborhood function*

B: $S \times S^{|N|} \rightarrow 2^{I}$ is the *behavior function*

Δt is the time period

The *cell space process* for *CS* with parameters *N*, *B*, Δt is defined as

$$CSP[N, B, \Delta t] = \langle S, X, Y, E, P, \delta_{int}, \delta_{ext}, \lambda, \rho, ta \rangle$$

where

$S = 2^{C \times I}$

$X = \varnothing$

$Y = 2^{C \times I}$

$E = \Phi$

$P = \varnothing$

$\delta_{int}(s, \varphi) = \bigcup_{c \in C} \{(c,i) \mid i \in B(S_t(c), (S_t(n_1), S_t(n_2), \dots))$ and $N(c)=\{n_1, n_2, \dots \}\}$

$\delta_{ext}((s, e), evt, \varphi) = s$

$\lambda(s) = \{s\}$

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

$ta(s) = \Delta t$

The neighborhood function *N* is defined exactly as in the basic *CA*, described in section 2.2.2. The behavior function *B* defines the *CSP*'s intentions for a cell, given the state of that cell and its neighbors. For notation simplicity, the term $|N|$ is used to denote the neighborhood size, which is assumed constant for all cells, as in the basic *CA* definition. Finally, Δt defines the periodicity the *CSP* sends its intentions to the *CS*.

Considering the *CSP* dynamics in the Process-DEVS formalism, the *CSP* periodically manifests its intentions by taking as input a *CS* state φ. The time advance function *ta* is a constant function that always outputs Δt. This will cause the internal transition function δ_{int} to be invoked every Δt time units. This function takes as input the *CS* state and generates a set of intentions. These intentions are stored in the internal state of the *CSP*. The output function λ makes every set of intentions produced by δ_{int} be sent as an event to the environment. Note that the output function λ returns the internal state itself, which is possible in this case because $S = Y$.

The *CS* and the *CSP* interact in a cyclic way. In each cycle, the *CSP* reads the *CS* state to compute its intentions, which are sent to the environment as an event. When the environment receives this event, a new state is computed for the *CS*. This cyclic procedure produces a series of cell space states φ_0, φ_1, φ_2, ... , which is given by the recurrence relation $\varphi_{t+1} = eff(\varphi_t, \delta_{int}(s, \varphi_t))$.

Besides increasing modularity, separating a cellular dynamic model into a *CS* and *CSP* does not cause a loss expressivity power when compared to traditional *CA*. The following theorem proves that.

Theorem 1: For any $CA = \langle C, S, N, T\rangle$, one can define an equivalent *CS*-*CSP* pair that produces the same sequence of cell space states.

Proof: Let us define a cell space $CS_{ca} = \langle C, S, S, eff\rangle$, where

$$eff(\varphi_t, itts) = \varphi_{t+1} \mid \varphi_{t+1}(c) = s, \qquad if\ \exists!(c, s) \in itts$$
$$= \varphi_t(c), \qquad otherwise.$$

Additionally, let us define a cell space process $CSP_{ca}[N, bhv, 1]$, where

$bhv(\varphi_t(c), (\varphi_t(n_1), \varphi_t(n_2), \ldots)) = \{\ T(\varphi_t(c), (\varphi_t(n_1), \varphi_t(n_2), \ldots))\ \}$.

Lemma: The CS_{ca}-CSP_{ca} pair produces the same sequence of cell space states as *CA*.

Proof:

Let $CS_{ca}=\langle C, S, I, eff\rangle$ and $CSP_{ca}[N, bhv, 1]=\langle S, X, Y, E, P, \delta_{int}, \delta_{ext}, \lambda, \rho, ta\rangle$

Given that CS_{ca} is at state φ_t, the next state φ_{t+1} is given by

$\varphi_{t+1} = eff(\varphi_t, \delta_{int}(s, \varphi_t))$

Since the CSP_{ca} behavior function *bhv* always returns a unitary set, it follows from the definition of δ_{int} for *CSP*'s that $\delta_{int}(s, \varphi_t)$ always contains exactly one pair (c, s) for each cell $c \in C$, therefore, it follows from the definition of the *eff* function:

$\varphi_{t+1}(c) = s \mid (c, s) \in \delta_{int}(s, \varphi_t)$

From the definition of δ_{int} and *bhv*:

$\varphi_{t+1}(c) = T(\varphi_t(c), (\varphi_t(n_1), \varphi_t(n_2), \ldots))$, where $N(c)= \{\ n_1, n_2, \ldots\ \}$

This is exactly the same recurrent relation that defines the state sequence of *CA*, as defined in section 2.2.2.

The dissociation between physical state and behavior increases the modularity of a cell-based simulation. This modularity brings reuse benefits, since the same *CSP* can be used with different *CS* and vice-versa. Another modularization benefit of this approach is the clear separation between the behavior implementation and the internal data structures of the *CS*. This separation is somehow imposed by Process-DEVS and is in accordance to the principles of *process oriented simulation*, discussed in section 3.2.4.

The separation between behavior and physical representation is also an important step towards integrating *CS*-based dynamic models with other modeling formalisms. This is achieved by having multiple processes interacting with the same *CS*, as in some agent oriented simulations. The next section discusses the possibility of using multiple *CSP*'s with the same *CS*. A full example of how a *CSP* can be integrated with other kinds of processes in a modular way is given in chapter 5.

4.2.3 Composition of Cell Space Processes

In order to meet the requirement of realistic simulation models, cell space processes *(CSP)* tend to become more complex. However, it is not desirable that complex phenomena be modeled by monolithic complex *CSP*'s. Instead, it would be much better if they were defined by a composition of simpler *CSP*'s. This kind of modularity has three main benefits: (1) it facilitates model reuse; (2) it makes models more intelligible; (3) it makes models easier to change and maintain. All these features are important in the context of serious games design.

In order to exemplify the problem, let us consider the case of an emergency situation where some amount of oil has leaked into the sea. In this specific example, the oil position is modeled as a *CS*, where each cell has a real number property, indicating the amount of oil in it. There are several factors that may alter the oil configuration in the *CS*, such as the leak itself, dispersion on water, evaporation, containment by barriers, recovery by pumps at sea, coast hitting, coast cleaning procedures, and so on. All these factors could be modeled as a single monolithic *CSP*. However, it is more desirable that each of them is modeled as an individual *CSP* and then composed to produce the overall behavior.

As an illustrative example, consider the case of integrating two *CSP*'s that represent two of the above factors: CSP_{disp}, which models the oil dispersion, and CSP_{cont}, which models the oil containment caused by containment barriers. For compatibility issues, we shall assume that both *CSP*'s use the same time step Δt and both are started at the same time instant. Therefore, they always produce events at the same time instants.

CSP_{disp} models the oil dispersion by producing intentions of the form $(c, move(a, d))$, where c is the cell where the oil is moving from, a is a positive real indicating the amount of oil and $d \in \{N, NE, E, SE, S, SW, W, NW\}$ is the direction that indicates which of the eight cells in the Moore neighborhood will receive the oil. The function $dest$: $C \times \{N, NE, E, SE, S, SW, W, NW\} \rightarrow C$ computes the destination cell from the origin cell and a direction. Each intention causes the amount of oil at cell c to be decreased by a, and the amount of oil in cell $dest(c, d)$ to be increased by the same amount.

CSP_{cont} models the oil containment. It produces intentions of the form $(c, block)$, where c is a cell which is blocked by a containment barrier and, therefore, should not receive any amount of oil.

The problem in this example is how to compose these two processes in a way they produce the correct oil behavior while keeping them independent and, if possible, unaware of each other.

Parallel Composition

The simplest way to compose these two processes is to arrange them in a parallel pattern, as illustrated in Figure 4.7. This way, each *CSP* updates the *CS* independently, one after the other. Since both *CSP*'s send events to the environment at the same time instants, the tie breaking function of the simulation will determine which one gets executed first. The resulting series of *CS* states is determined as in Figure 4.7. The first *CSP* reads the *CS* state φ_t and applies a transition to the cell space, leaving it in an intermediate state φ_{int}. The second *CSP* reads this intermediate state and generates another transition that will finally produce the next state φ_{t+1}.

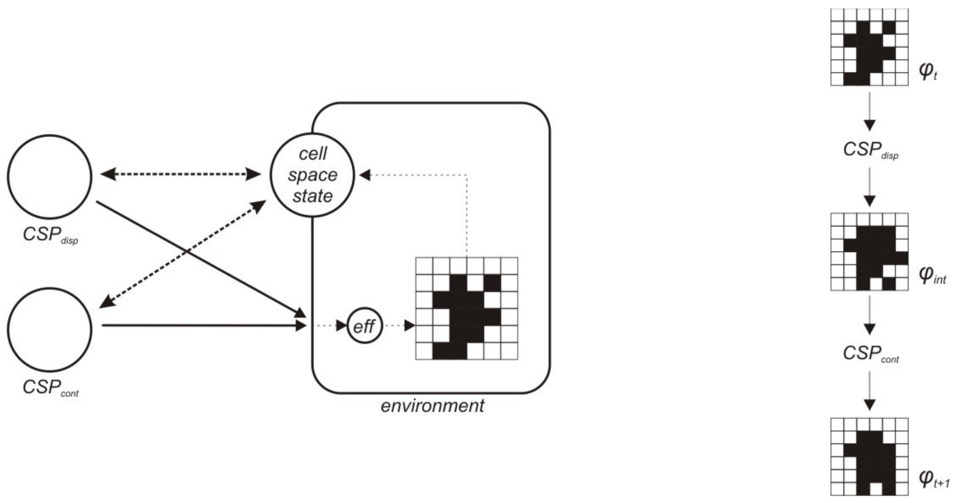

Figure 4.7 – Parallel composition of CSP's

This simple form of *CSP* composition has three limitations:

- There is no guarantee that a third process will not interfere with the intermediate *CS* state φ_{int}.
- If the events of the two *CSP*'s interfere with each other, it may not be possible for the second *CSP* to undo the effects of the first *CSP* because the initial *CS* state φ_t was lost in the transition to the intermediate *CS* state φ_{int}.
- This form of composition is not closed. Given two *CSP*'s composed in parallel, it may not be possible to define one single *CSP* that will produce the same effects.

Considering this simple parallel composition of CSP_{disp} and CSP_{cont}, and assuming that CSP_{disp} alters the environment before CSP_{cont} in each cycle, the weakness of this composition pattern is felt immediately. For instance, consider that, in a given cycle, CSP_{disp} generates an intention *(c*, *move(a*, *d))* and, on the same cycle, CSP_{cont} generates an intention *(dest(c*, *d)*, *block)*. These intentions are conflicting, since CSP_{disp} wants to move oil into a cell that is blocked by CSP_{cont}. In this case, since CSP_{disp} has a higher priority, the oil would move to cell *dest(c*, *d)*, causing an inconsistent intermediate state. That may generate a considerable problem because a third process might access this intermediate state. Another problem is that this inconsistency must be resolved when CSP_{cont}

generates the event *(dest(c, d), block)*. The best solution would be to move the oil back to its original cell, but that is not possible because the initial state is no longer known, and it is impossible to determine the origin cell.

This brief example illustrates two of the problems identified for this parallel form of *CSP* composition. The third problem is that it is not closed. This is easily proved by the following counter-example:

Consider a $CS = \langle C, S, I, eff \rangle$ where:

$C = \{c\}$ is the set of cells, where c is the only cell in this cell space

$S = \Re$ is the state set. The cell c has a real number defining its internal state

$I = \{increase\}$ is the intention set with only one possible intention

$eff(\varphi, \varnothing) = \varphi$

$eff(\varphi, (c, increase)) = (\varphi \leftarrow (c, \varphi(c)+1))$

This *CS* has a single cell *c*, which has a real number as its state, and accepts a single intention *increase*. When received, this intention increases the cell state by one.

Consider also a $CSP[N, B, \Delta t]$, where $N(c) = []$, $B(s, ns) = \{increase\}$ and $\Delta t = 1$, where *[]* represents a tuple of size zero. This *CSP* always outputs the intention *increase*, regardless of the previous *CS* state.

Now construct a simulation with the just defined *CS* as part of the environment and two exact copies of this *CSP*, composed in parallel, as in Figure 4.7. It is easy to check that, at each time step, the state of *c* will be increased by two. However, it is impossible to write a single *CSP* that will cause the state of *c* to be increased by two because the *CS* definition only allows one possible intention *increase*, which increases the value of *c* by only one.

Composition with a Conflict Resolver

In order to overcome the problems with pure parallel composition of *CSP*'s, we propose the use of a *conflict resolver (CR)*, as depicted in Figure 4.8. In order to formally define *CR*, we use the same notation for lists introduced in section

3.3.1 (for the definition of I/O processes). The *CR* is a process, defined as $CR[n, rf] = \langle S, X, Y, E, P, \delta_{int}, \delta_{ext}, \lambda, \rho, ta \rangle$, where the parameters are

n is the number of *CSP*'s in the composition

$rf: ITT^n \rightarrow ITT$ is the *conflict resolver function*, where $ITT = 2^{C \times I}$

and the Process-DEVS process properties are

$S = ITT^* \times \{0, 1, \ldots , n\}$, where ITT^* is a list of elements of ITT

$X = Y = ITT$

$E = \Phi$

$P = \varnothing$

$\delta_{int}((itts, i), \varphi) = ([], 0)$

$\delta_{ext}(((itts, i), e), \varphi, itts_{new}) = ([itts_{new} \mid itts], i+1)$

$$\lambda(([itt_1, itt_2, \ldots , itt_i], i)) = \begin{cases} rf(itt_1, itt_2, \ldots , itt_i) & \text{if } i = n \\ \varnothing & \text{if } i \neq n \end{cases}$$

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

$$ta((itts, i)) = \begin{cases} 0 & \text{if } i = n \\ \infty & \text{if } i \neq n \end{cases}$$

This process stores the intentions issued by different *CSP*'s. When it has received the intentions from all *n CSP*'s, it applies the *conflict resolver function* to determine the final intentions of this set of *CSP*'s. Since the *CR* processes needs to receive *n* intention events before computing the final result, it is important that all *CSP*'s work at the same frequency (i.e. all of them must have the same Δt). Hence, it is guaranteed that, in a sequence of *n* received events, there will be one from each *CSP*.

In this form of composition, the *CSP*'s are totally unaware of the *CR*. This helps keeping a high level of modularity. All of them take as input the same *CS* state to produce their intentions. No intermediate *CS* states are produced. Therefore, they act as a single *CSP* from the point of view of the *CS*, which receives a set of intentions every Δt time units.

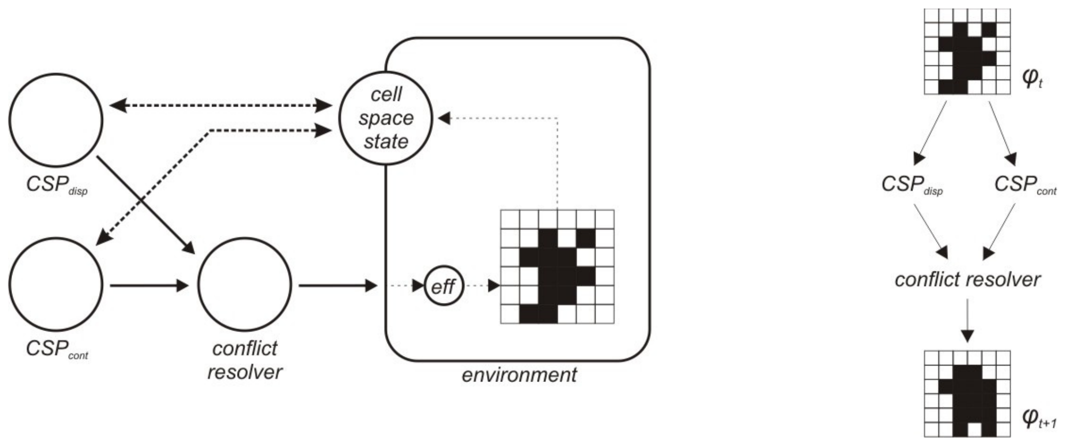

Figure 4.8 – Composition of CSP's with a Conflict Resolver

The use of a *CR* provides a way of solving the previously mentioned oil dispersion and containment problem, while keeping the logic of both *CSP*'s separate. This *CR* is defined as *CR[2, oil_cr]*, where

$$oil\_cr(itts_{disp}, itts_{cont}) =$$
$$\{(c_{from}, move(a, d)) \in itts_{disp} \mid (\forall(c_{to}, block) \in itts_{cont})(c_{to} \neq dest(c_{from}, d))\}$$

where $itts_{disp}$ and $itts_{cont}$ are the intentions generated by the oil dispersion and oil containment processes respectively. This *CR* will act as an intention filter and will let pass only the oil move intentions that do not attempt to put oil on blocked cells. We have therefore created the abstraction of a *CSP* that handles both dispersion and containment of oil and that is defined by composition of two individual *CSP*'s, which are totally unaware of each other.

In addition to allowing a higher degree of modularity, the composition of *CSP*'s via *CR* is also *closed*. This means that, for any composition of *n CSP*'s, it is possible to write one single *CSP* with an equivalent behavior. This is easily verifiable:

Theorem 2: For any set of *CSP*'s $\{CSP_1[N_1, B_1, \Delta t], CSP_2[N_2, B_2, \Delta t], \ldots, CSP_n[N_n, B_n, \Delta t]\}$ composed with a conflict resolver *CR[n, crf]*, it is possible to write a single $CSP_{comp}[N_{comp}, B_{comp}, \Delta t]$ with equivalent behavior.

Proof:

Let us define

$N_{comp}(c) = (nc_{11}, nc_{12}, \ldots, nc_{21}, nc_{22}, \ldots, nc_{n1}, nc_{n2}, \ldots)$

where $N_i(c) = (nc_{i1}, nc_{i2}, \ldots)$

$B_{comp}(s_c, (s_{11}, s_{12}, \ldots, s_{21}, s_{22}, \ldots, s_{n1}, s_{n2}, \ldots)) =$

$crf(B_1(s_c, (s_{11}, s_{12}, \ldots)), B_2(s_c, (s_{21}, s_{22}, \ldots)), \ldots, B_n(s_c, (s_{n1}, s_{n2}, \ldots)))$

where all states needed as inputs to all *n* behavior functions are also inputs to B_{comp}. This is easily verified because, by definition, N_{comp} contains all cells of any of the *n* neighborhood functions.

For any *CS* state *S*, *CSP*$_{comp}$ outputs the same intentions as the composition of the individual *CSP*'s using *CR* as conflict resolver. This is so because the behavior function of *CSP*$_{comp}$ receives as input precisely the intentions of each individual *CSP*, and outputs those intentions with conflicts resolved by the *crf* function, which is precisely the definition of how the conflict resolver works.

C*losure under composition* is indeed an interesting property of any simulation formalism that strives for modularity and reuse (Zeigler et al. 2000). Besides the reuse of sub-models, it also allows cascading composition in several levels of abstraction. In fact, the composition of *CSP*'s with a *CR* has solved all three problems identified with pure parallel composition. Therefore, it should be seen as a more reliable way of *CSP* composition.

4.3 Multi-Agent Systems

The discussion of section 3.2.2 concluded that agents should be modeled as specialized simulation elements. Agents are commonly modeled as cognitive entities which sense their surrounding environment through *sensors* and act on it through their *actuators*. In the middle of this process is the reasoning phase, which can be further divided into smaller pieces, such as in the Jason toolkit described in section 2.3.1. Most multi-agent simulation toolkits provide some degree of modularization. Therefore, when implementing agents on top of Process-DEVS, it is highly desirable to keep this modularity.

However, there is no consensus on what should be the internal elements of the agent reasoning process. Therefore, the next sections will describe simply a framework for modeling the basic notions of sensing, acting and reasoning. More detailed structures can be implemented on top of that.

Multi-agent simulations are often modeled in *discrete time* formalisms (Theodoropoulos et al. 2009). However, this is often pointed as a limitation (Michel et al. 2009). Therefore, the proposed framework will keep the more flexible *discrete event* paradigm of Process-DEVS.

4.3.1 Modular Agent Architecture

Agents interact with their environment. The interaction cycle is often composed of three main steps: sensing, reasoning and acting. Usually, multi-agent simulation frameworks decompose the reasoning stage into more detailed parts. However, there is no consensus on which is the right way of doing so. The main reason for this is the different kinds of behaviors intended for agents. Sometimes agents have a very simple reactive behavior and sometimes they are required to reason logically, remember facts, formulate beliefs and try to achieve goals. Therefore, since agent reasoning is not the focus of this work, we shall consider it a black box that receives information from sensors and sends its intentions to actuators.

Reasoning, sensing and acting are modeled by processes, which are named respectively, *reasoning processes*, *sensor processes* and *actuator processes*, as depicted in Figure 4.9. Sensor processes read the state of its environment and produces events representing *sensations*. Reasoning processes take these sensations as input and produce *intentions*. The intentions are sent to actuator processes, which will perform the actions and alter the environment.

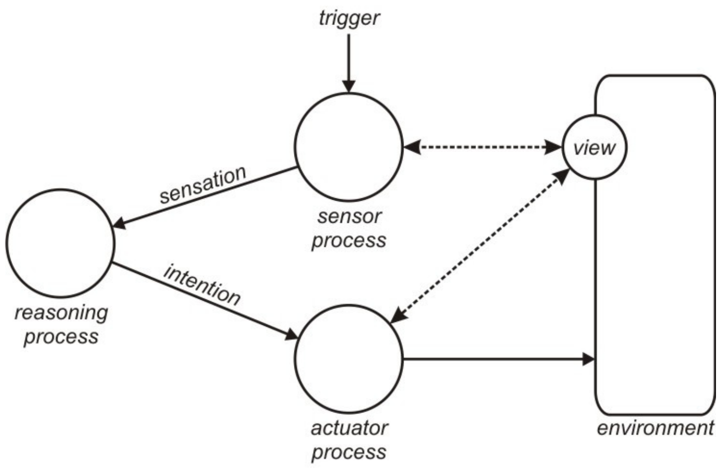

Figure 4.9 – An agent with its behavior decomposed in sensor, reasoning and actuator processes

The sensor process is defined as

$$P_{sen}[E, Y, \sigma] = \langle S, X, Y, E, P, \delta_{int}, \delta_{ext}, \lambda, \rho, ta\rangle$$

where

E is the set of states in which the sensor can perceive the environment

Y is the set of sensations, which is the same as the output set

σ: $E \rightarrow 2^Y$ is the function that, given an environment state, returns a set of corresponding sensations

$S = 2^Y$

$X = \{trigger\}$

$P = \varnothing$

$\delta_{\text{int}}(s, env) = \varnothing$

$\delta_{\text{ext}}((s, e), env, trigger) = \sigma(env)$

$$\lambda(s) = \begin{cases} \varnothing & \text{if } s = \varnothing \\ s & \text{if } s \neq \varnothing \end{cases}$$

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

$$ta(s) = \begin{cases} \infty & \text{if } s = \varnothing \\ 0 & \text{if } s \neq \varnothing \end{cases}$$

In its internal state, the sensor process stores the last perceived environment state. Whenever it is triggered, it invokes the σ function to produce a set of sensations, which are output in the same time instant as it was triggered. The sensor process is triggered when it receives the *trigger* event. Any process can send a trigger event to the sensor process. For example, consider a proximity sensor which periodically checks the distance of a given object to the agent. Whenever this distance is less than a threshold radius, it produces a sensation of nearby object presence. In this case, there might be a clock process, which only behavior is to periodically send a trigger event to the sensor process.

This triggering mechanism allows one to build a simulation where sensations are only computed when they are really needed. For instance, an agent may be in such a situation where the sensations of a particular sensor do not affect its decisions. In that case, the simulation may be modeled in such a way that a specific sensor is not triggered in such situations, thereby avoiding unnecessary processing.

The sensations produced by sensor processes are used as input by the reasoning process, which shall not be defined formally because there are many different ways to model the internal reasoning of agents. The important fact here is that the reasoning process produces intentions, which are sent to the actuator processes. The role of an actuator process is to take intentions and produce the actual actions that are performed in the environment. The actuator process is defined as

$$P_{act}[Y, E, A] = \langle S, X, Y, E, P, \delta_{int}, \delta_{ext}, \lambda, \rho, ta \rangle$$

where

Y is the set of events that this actuator may generate to the environment

E is the set of states in which the actuator can perceive the environment

$A = \{it_1, \dots, it_n\}$ where $it_i: E \rightarrow 2^Y$ is the set of intentions

$S = 2^Y$

$X = A$

$P = \emptyset$

$\delta_{int}(s, env) = \emptyset$

$\delta_{ext}((s, e), env, it) = s \cup it(env)$

$$\lambda(s) = \begin{array}{ll} \emptyset & if\ s = \emptyset \\ s & if\ s \neq \emptyset \end{array}$$

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

$$ta(s) = \begin{array}{ll} \infty & if\ s = \emptyset \\ 0 & if\ s \neq \emptyset \end{array}$$

The behavior of actuator process is very similar to the behavior of the sensor process, only with the flow of events in the opposite direction. It is triggered when it receives an intention event. When that happens, it computes the proper events to send to the environment and sends them at the same time instant. As the sensor process, it uses a transient internal state to implement that behavior.

Each intention defines a function that takes as input the environment state. This is necessary so that the actuator may check the preconditions that need to hold before a particular intention can be materialized into a concrete environment state change. For example, an agent cannot move to a location which is occupied by another agent, even if it intends to do so. Another reason for checking the environment is that the effects of an intention may depend on external factors. As an example, consider an agent whose movement is affected by the wind. Each time it issues a move intention on a given direction, the actual displacement of its position will depend on the wind direction and intensity.

The use of actuators, as well as sensors, helps isolating the core logic of agent reasoning. This way of modularizing the behavior of an agent allows one to express the agent reasoning only in terms of sensations and intentions, without worrying about the interaction with the environment. That does not mean that the agent reasoning cannot be further modularized. It has been presented here as a unique black box process, but it could very well be decomposed into a set of more specialized processes.

This modular architecture, besides allowing reuse of agent parts, can also help improving simulation performance by sensor and actuator sharing among agents. For example, if there are lots of agents that need to sense the same environment data with the same frequency, they can share a single sensor process. This is accomplished simply by changing the process coupling structure in the simulation, without touching the definition of the sensor process behavior. This is also true for the actuators.

4.3.2 Simulation of Multi-Agent Systems

A multi-agent system is a system in which multiple agents interact with each other. They interact either through the environment or directly, via message exchanging. Considering the interaction through the environment, there is no difficulty in composing a set of agents in a multi-agent system. It is only necessary to put them in the same simulation, sensing and acting on the same environment. On the other hand, in order to allow message exchanging, it is necessary to provide some means of taking a message from an agent to another. There are many ways to accomplish this. Figure 4.10 depicts the simplest case, where the agents exchange messages directly, by sending events to each other.

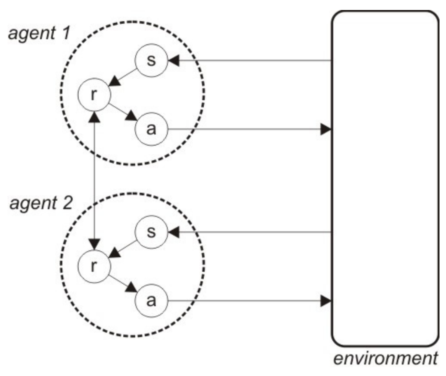

Figure 4.10 – A multi-agent system simulation

Another possible way of message exchanging is to provide each agent a mailbox as part of the environment state. This has the advantage of keeping agents uncoupled, i.e. not sending events directly to one another, which makes it easier to implement systems where agents are created and destroyed frequently. However, this way of designing a multi-agent system can be counter-intuitive, as messages are usually not thought of as part of the physical environment.

A more structured message exchanging mechanism can be implemented by designing a specialized process or set of processes to handle message routing between agents. However, since there is no consensus on how messages should be exchanged in multi-agent systems, it is difficult, and maybe impossible, to point a generic solution that will fit all cases. More likely, the best design will depend on the objectives of the target system.

4.4 An Informal Discussion on Process Patterns

In general, modularity solutions lead to structure issues. Indeed, modularization is the process of dividing objects into smaller pieces, which should then be structured somehow to produce the desired result. This chapter has presented ways to implement some popular dynamic modeling formalisms on top of Process-DEVS, with emphasis on process modularity and composition. Based on those cases, this section describes some generic solutions for process composition that may also apply to a number of other different cases.

The solutions are presented in the form of *process patterns*, by analogy with object-oriented design patterns (Gamma et al. 1995). These patterns will describe a number of ways in which to structure processes in a simulation to solve a particular problem while keeping a high degree of modularity and flexibility. These patterns are designed for Process-DEVS, but some of the ideas are generic enough so that they can be applied to other simulation frameworks as well.

The patterns in this section are not formalized as in the previous sections. Indeed, since these patterns are expected to be applicable to simulations in general, it would be necessary to have a broader experience with Process-DEVS on various simulation fields. At this point, it is not clear to which extent the patterns should restrict the nature of the processes.

Chapter 5 describes examples and discusses the benefits of the presented patterns in the field of emergency simulation.

4.4.1 Parallel Pattern

The *parallel pattern* is the most straightforward pattern. It consists of isolated processes making changes to the environment without any direct interaction between them, as depicted in Figure 4.11. This pattern has been extensively used for multi-agent systems simulation, where agents interact only through the environment, such as insect colonies (Bonabeau et al. 1999) and pedestrian behavior (Bandini et al. 2009).

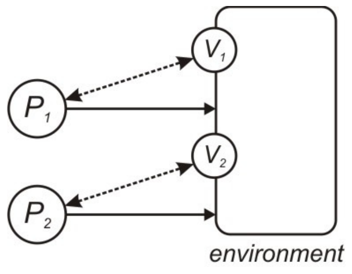

Figure 4.11 – Parallel pattern

The main benefit of this pattern is that the processes do not directly depend on each other. There is no direct communication between them. Therefore it is easier to reuse any of them in another simulation with a different structure.

This pattern can be used only if the logic of the modeled phenomenon can be broken into simple and independent processes. Note that the processes do not necessarily represent the behavior of physically separate entities. They may also represent different aspects of the behavior of a single physical entity. For example, consider an oil slick on the surface of the sea. This oil moves according to the weather conditions and it also gets recovered by pumps placed on recovery boats. These two aspects, namely the dispersion and recovery, can be modeled as completely independent processes.

4.4.2 Interference Pattern

Special care should be taken when using the parallel pattern. Some conceptual problems may arise if some behavior logic of the processes is put into the environment only to eliminate direct communication between them.

Consider for example the case of two characters moving through the environment. One of them is stronger than the other. When both want to move to the same place, the stronger prevails and the weaker is pushed to some adjacent place. One could model this situation with the parallel pattern where each character sends events to the environment when they wish to move. In this case, the environment should treat the case of collision and apply the rule of the stronger. This design is not so good because some of the logic that controls the position of the characters was modeled inside the environment.

However, it is still possible to keep these two interfering processes separate and unaware of each other with the *interference pattern*. This pattern attempts to increase the modularity of processes that directly alter the environment by sending events to it. The idea is to treat the output of the interfering processes as intentions instead of effects on the environment. If the intentions of two or more processes conflict with each other, a resolver process will define which effects should be applied. This is accomplished by redirecting the outputs to this resolver process as depicted in Figure 4.12. Hence, the events are intercepted by the resolver, which treats them as mere intentions. It then applies the interference logic, produces the resulting effects and sends them as events to the environment.

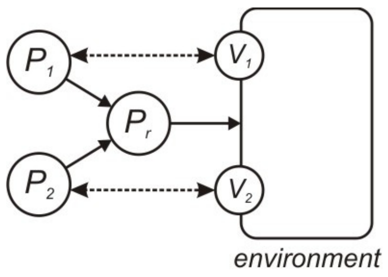

Figure 4.12 – Interference pattern

In the example of the two moving characters, both processes would send their movement intentions as events to the resolver. Whenever the resolver detects that both want to move to the same place, it treats the conflict and sends the correct movement effects to the environment. Thus, the environment is kept unaware of any logic specific to the movement of the characters.

The benefit of this pattern is that, although the processes interfere with each other, they are still kept totally unaware of each other. The only element that is aware of all interfering processes and the interference logic is the resolver process. For the other processes, no dependencies between them are introduced and they can be reused at will. Therefore, this pattern is useful when two or more processes with complex behavior interfere with each other in the way they alter the environment.

The interference pattern was already used in section 4.2.3 to model the composition of cell space processes. Here the pattern is made generic to any kind of process. In fact, a cell space process can be composed with different kinds of processes using the interference pattern.

The trick of this pattern is to transform an *acting behavior* into an *intentional behavior*. The events output by the processes represent not actions that alter the environment state, but intentions that are sent to the resolver process. Once the conflicts are treated, the resolver process outputs the actions to the environment. This concept is also used in multi-agent system simulation frameworks, where the acting behavior of an agent is separated from its intentional behavior by *actuators*, as shown in section 4.3.1.

4.4.3 Composite Pattern

If a reasonably complex process cannot be broken into fully independent parts, the *composite pattern* may be used to break it into less complex interdependent parts. The idea is to model complex behavior in hierarchical form and distribute its tasks among a hierarchy of processes. One root process represents, at the highest level of abstraction, a whole set of processes to the external world. However, it does not implement all the complex behavior, but rather delegates lower level tasks to its children. Figure 4.13 illustrates the pattern. One simple and common example is a process that controls a moving object in the environment. Among other behavioral aspects, the controller process is responsible for implementing the object's motion. Once it decides that the object should move to a given location through a specific trajectory, it forks a *move process* that will actually change its position from time to time until the destiny is reached. Hence, the parent process can focus on higher level primitives and the lower level details of displacement are handled by its child, the move process.

Note that the structure of the hierarchy of processes is not rigid. It may change with time. In the just mentioned example, when the object has reached its destination, the move process can be finished.

The workflow process defined in section 4.1.3 is a good example of this pattern. In that case, the parent process represents the workflow itself and it delegates the execution of the actions to children processes. The workflow process simply defines which processes to fork and when, by following the workflow logic.

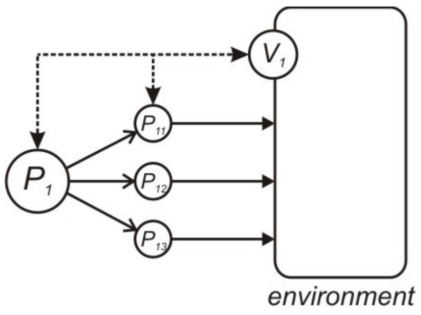

Figure 4.13 – Composite pattern

The benefit of this pattern is modularization and reuse. Although we cannot say that the processes in the hierarchy are totally independent, they certainly help in breaking a complex dynamic model in smaller and simpler sub models. The reuse of those simpler sub models is achieved when composing another complex behavior. For example, in the case mentioned above, the move process can be reused by another controller process that also implements the motion of some other kind of object.

This way of composing a complex behavior is somewhat different from that of the coupled DEVS model described in section 2.2.1. In that case, composition is achieved through aggregation, while here it is achieved through process forking. It is not clear which one is better, if any. This comparison could be the object of future work. However, child forking is certainly more flexible because its structure is mutable and defined only at the time the simulation is actually running.

4.5 Summary

This chapter formalized a number of ways to implement some common dynamic modeling formalisms on top of Process-DEVS. Section 4.1 presented a way in which workflows can be mapped to Process-DEVS. In fact, in the way workflow processes were defined, they represent a form of process composition where the workflow logic defines which processes are created and when. Section 4.2 discussed the issue of modularity in the domain of cell space processes and presented a formalism for dealing with it on top of Process-DEVS. It also showed how to compose cell space processes out of smaller pieces. Section 4.3 presented a formal framework in which it is possible to model multi-agent systems on top of Process-DEVS, with support for sensors and actuators.

The solutions presented in the first three sections of this chapter suggested some patterns in which to structure processes with interesting modularity properties. Section 4.4 informally discussed these patterns, leading to interesting conclusions that still need further experiments to be fully validated.

This chapter illustrated the capability of POS and Process-DEVS of serving as basis for modeling the dynamics of serious games in a modular way and integrating different existing formalisms.