A scenario language to orchestrate virtual world evolution

Article · January 2003

CITATIONS
24

READS
98

2 authors:

Frédéric Devillers
École Nationale d'Ingénieurs de Brest
7 PUBLICATIONS **97** CITATIONS
SEE PROFILE

Stéphane Donikian
Golaem
92 PUBLICATIONS **2,048** CITATIONS
SEE PROFILE

Eurographics/SIGGRAPH Symposium on Computer Animation (2003)
D. Breen, M. Lin (Editors)

A Scenario Language to orchestrate Virtual World Evolution

Frédéric Devillers[1] and Stéphane Donikian[2]

IRISA
Campus de Beaulieu
F-35042 Rennes, FRANCE
[1] INRIA, *fdeville@irisa.fr*
[2] CNRS, *donikian@irisa.fr*

Abstract

Behavioural animation techniques provide autonomous characters with the ability to react credibly in interactive simulations. The direction of these autonomous agents is inherently complex. Typically, simulations evolve according to reactive and cognitive behaviours of autonomous agents. The free flow of actions makes it difficult to precisely control the happening of desired events.
In this paper, we propose a scenario language designed to support direction of semi-autonomous characters. This language offers temporal management and character communication tools. It also allows parallelism between scenarios, and a form of competition for the reservation of characters. Seen from the computing angle, this language is generic: in other words, it doesn't make assumptions about the nature of the simulation. Lastly, this language allows a programmer to build scenarios in a variety of different styles ranging from highly directed cinema-like scripts to scenarios which will momentary finely tune free streams of actions.

Categories and Subject Descriptors (according to ACM CCS): I.3.6 [Computer Graphics]: Languages

1. Introduction

The goal of the behavioural model is to simulate autonomous entities like organisms and living beings. A behavioural entity has the following capabilities: perception of its environment, decision, action and communication [4]. The scenario component of a behavioural simulation carries out the responsibility for orchestrating the activities of semi-autonomous agents that populate the virtual environment. In common practice, these agents are programmed as independent entities that perceive the surrounding environment and under normal circumstances behave as autonomous agents. However, in most experiments and training runs [13], people want to create a predictable experience. This is accomplished through the direction of agent's behaviours. To facilitate coordination of activities, actors have to be built with interfaces through which they can receive instructions to modify their behaviours. Scenario processes control the evolution of the simulation by strategically placing objects in the environment and guiding the behaviour of actors to create desired situations. Scenario Authoring Tools are also important for digital crowds used as special effects for the cinema Industry. Jon Labrie, Chief Technology Officer at Weta, who was in charge of crowd animation for the film *The Lord of the Rings* said: *It turns out that it is at least as difficult to control an intelligent autonomous agent as it is to control a real actor. They don't necessarily do what you want them to do. So tools for choreography are just as important as anything else*[14]. Another domain in which scenario becomes very important is the interactive art experience in virtual or mixed reality [10, 20]: the problematic here is to provide a way for the artist to control the interactive experience lived by the user as his actions are not predictable.

To one point of view, a scenario can be treated as the behaviour of a disembodied object. This suggests that the same languages used to program the behaviours of simulated characters with material properties can be used to program the modules that manipulate the environment by creating, destroying or modifying characteristics of entities, and coor-

dinating the actions of other actors. A number of research teams working on simulation have converged on hierarchies of parallel state machines as a model for programming reactive behaviours [12, 16, 2, 8]. We think that the state machine model has also much to offer for scenario programming. The issue in this paper is to present how we intend to describe a scenario whose objective is to partially constrain the activity of semi-autonomous entities.

2. Related Works

ASAS, proposed by Reynolds in 1982 [18], is one of the first animation language based on actors and scripts, and its goal is to offer to the animator the ability to control an animated sequence by using a script. ASAS is based on LISP, and adds specific notions such as geometric and photometric characteristics, transformation operators and the data structure of an actor. Ridsdale et al. [19] proposed a system to animate characters in a theatrical environment. A scenario is decomposed in a sequence of scenes for which the scenarist has to define, initial and final location of actors, their goals and relational constraints. An expert system is used to determine actions that each actor will have to perform during the scene. If we address the problem of controlling semi-autonomous entities, the preceding descriptive languages cannot be of use as it is impossible to know in advance the behaviour of characters. In such a case, a simulation consists of decentralized autonomous agents that evolve in an environment and interact with each other and with the environment. In the case of a semi-autonomous entity model, instructions might be given by a coordinator. Different kinds of coordination can be implemented:

goal oriented: define goals to achieve during the animation. The main complexity consists in planning actions to be done by actors;

rules: observe actors evolving in the scene, and on a specific situation, start a script;

ambient: there is a main character (usually the user in the loop) and all actions of other actors are determined depending on his own behaviour and are executed in order to test and study his reaction;

sequence of actions: determine a set of action to be done and their time schedule.

In all those cases, a script cannot be exhaustive and virtual actors should be able to improvise or decide themselves actions to perform. A script will specify actions only on a few number of entities at each time and all other should evolve autonomously. On the basis of their behavioural model described with SCA control loops (reflexive behaviour) and PatNets (finite state machines), Badler et al. [2] propose to use higher levels of PatNets to define goals of actors and their schedule. In a similar way, J. Kearney et al. [5] use HCSMs (Hierarchical Concurrent State Machines) in their Hank simulator to describe ambient scenarios, acting on the traffic around the user driven car. Such scenarios are composed of triggers and beacons which are used to produce some specific events and directors which are responsible for choreographing entity behaviours to create specific situations. More recently, P. Willemsen [22] introduces SDL, a Scenario Description Language. It is an interpreted scripting language that interactively evaluates expressions entered by an experimenter and translates them into HCSM inserted and executed into a simulator. This language contains some instructions such as mathematical functions, conditions and four monitor statements (when, every, aslongas, whenever). Due to the interpretation choice, some limitations are imposed on the combination of instructions. We will use the same kind of monitoring instructions in our language but in a more complex way thanks to a compilation process.

In ViCrowd [15], crowds can be controlled at different levels and one of them consists in a script language where action, motion and behavioural rules are defined in order to specify the crowd behaviours. Improv is an authoring system [17] for scripting interactive actors in virtual worlds. Participants are represented by fully articulated human figures or avatars. Body movements of avatars are computed automatically by the system. The author needs to control the choices an actor makes and how the actors move their bodies. Authors specify individual actions in terms of how those actions cause changes over time to each degree of freedom (DOF) in the model. An Improv actor can be doing many things at once, and these simultaneous activities can interact in different ways. The author can place actions in different groups, and these groups are organized into a "back-to-front" order. Actions which are in the same group compete with each other and each action possesses some weight (global actions are located in the rear groups, and local ones in the front groups). Different scripts can run in parallel and can be ordered on the same temporal referential by using instructions like *wait* n *seconds*. In that case, start time of actions are scheduled. Non deterministic behaviours can also be expressed by specifying that an actor has to choose from a set of actions or scripts. Weights can be used to affect the probability of each item being chosen.

Usually scripts are applied to virtual actors which do not have real cognition and hence are unable to understand natural language [21]. Another key issue is the translation of scripts or scenarios written in a simplified form of natural language into something understandable by virtual actors. N. Badler et al. [3] proposed to control character actions by using sentences expressed in natural language. To support language understanding and to transform it into actions executed by autonomous characters, they have introduced the PAR model (Parametric Action Representation). A PAR gives a complete description of an action, and it is parameterized by a set of characteristics which can be given as adjuncts to the action verb in a sentence. One of the biggest problem in using some kind of natural language to specify scenarios is the reference to objects. In the current discourse, it is usual to

say "the object", because the author knows which object he is talking about, but in a scenario it is necessary to name objects or to refer to them by a designation operation. In some very directive scenarios, the animator will be able to give a name to all objects and there will be no ambiguity in the description, but in other cases like situations which produce events to start a goal oriented scenario, it is not possible to plan in advance which objects will be involved in it. Our objective is to specify a language which will allow dynamic selection of actors at run-time based on class and set operations. This language should be able to describe the four kinds of scenarios and mix them together. In this paper, we will focus on the structure of the scenario language and will illustrate its potential use on an example.

3. Scenario Authoring

3.1. Introduction

We have specified a scenario language which allows a description of scenarios in a hierarchical manner and their scheduling at simulation time. This language has its own syntax and is compounded of a set of specific instructions. The language contains classic instructions such as *if, switch, repeat until, random choice, wait* which are executed inside a time-step. It contains also more specific instructions (*wait-for, eachtime, aslongas, every*) that will spend more than one time-step to be finished and that can run in parallel during the execution of the scenario they belong to. All those instructions can be composed in a hierarchical manner. To manage actors, the language offers also specific instructions to specify the interface of an actor, to reserve and free actors. By using priorities, a scenario can take an actor from another one, which will be informed by a message. Concerning messages, each scenario can subscribe to those that are of interest to itself.

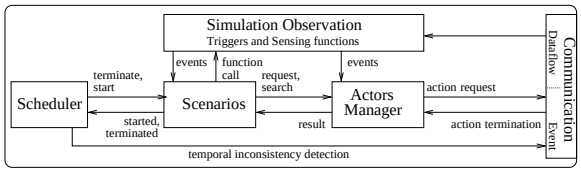

Figure 1: *The global architecture.*

A scenario can be decomposed into sub-scenarios and each scenario is corresponding to a set of tasks or actions, ordered on a global temporal referential. Tasks in a scenario can modify characteristics of actors, create them or ask an actor to perform a specific action. A scenario can start at a predefined time given by the author but can also be started when a situation occurs (conditional scenario). Some of those scheduling informations are stored in a dynamic execution graph. A scheduler uses it to start or terminate scenarios. To detect situations, triggers and sensing functions are managed by the simulation observation (cf figure 1). For example, a circular trigger will detect any object entering or exiting its area. Instead of creating a complete and self-contained language, we have chosen an open approach which consists in a full integration in C++. As explained below, the scenario language is built upon C++ and it is possible to introduce C++ code everywhere inside scenarios.

3.2. State Machine Approach

A state machine approach has been chosen to specify most of the instructions of the scenario language, and also their composition. Due to the expressivity of their graphical representation, we will present it to specify the semantics of those instructions. Each state machine is composed of states and transitions between states. As illustrated in figure 2, a transition between a state A and a state B is composed of two parts: a condition and a sequence of actions. If the symbol # is the condition part, it means that the condition is always true, while if # constitutes the action part, no action is associated to the transition. Each state machine of the system is either an atomic state machine, or a composite state machine. Some specific instructions can be used to start (*start(SM)*) and terminate (*terminate(SM)*) a sub-state machine SM and to know if a set of sub-state machines are terminated (*end(*$SM_1 \ldots SM_n$*)*).

Optional duration parameters can be attached to a state. The first one (d_{min}) is used to force the state machine to keep a state active for a minimum duration. This allows the modelling of delay: it is not possible to fire any transition until the time spent in the state become higher than this duration. On the contrary, a maximum duration d_{max} can also be specified. When this value is reached, an event *eod* is generated in order to warn the system that something that should have happened actually did not.

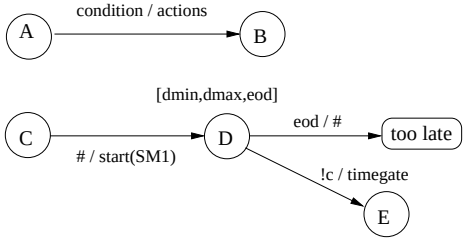

Figure 2: *the state machine model*

Our state machine model makes use of the superstep semantics of Statecharts [11], i.e. the ability to fire several following transitions at the same date, this being limited by a *timegate* (which prevents from firing the following transition at the same date). This state machine model has been implemented through a description language which is compiled and used to generate C++ code[8].

3.3. Grammar of the Scenario Language

Let us introduce now the different instructions of the language and their semantics. We first introduce the low-level instructions which can be directly expressed in state machines. Then, hierarchy, scheduling and actor management issues are addressed. Concerning the grammar of the scenario language, keywords are written in bold, whereas italic typeface represents a non-terminal rule. A * stands for a $0..n$ repetition, while a $^+$ stands for a $1..n$ repetition. | is the alternative operator. A statement enclosed in {} is optional and [] are used to enclose alternatives.

Language ::= { *list_of_Actors* } { *list_of_Modules* } *scenario* (*scenario*)*
scenario ::= **scenario** *ID* **(** { *parameters* } **)** **{** *scenarioDef* **}**
scenarioDef ::= (*instance*)$^+$ *schedule* | *SeqElemTask*

The rule *list_of_Actors* is used to specify the C++ type of the Actor's Interface, while the rule *list_of_Modules* is used to specify which objects have to be created to receive actors. Each scenario is composed of a name (*ID*), a set of parameters (rule *parameters*) and a body *scenarioDef* inside brackets. Two alternatives are offered to describe the body part of a scenario (rule *scenarioDef* of the grammar):

- a meta-scenario composed by different sub-scenarios (named and parameterized) and a specific part to specify their scheduling.

instance ::= **instance** *ID* **of** *ID* **(***Params***);**
schedule ::= **schedule {** [*allenRelation* | *instantRelation*]$^+$ **};**

- a scenario consisting of a sequence of elementary instructions (rule *SeqElemTask*).

SeqElemTask ::= *elemTask* **;** ({ **wait (***timeExpr***),** } *elemTask* **;**)*
elemTask ::= [*Declaration* | *Implementation* | *Include* | *Destructor* | *if* | *switch* | *repeat* | *waitfor* |*asLongAs* | *eachtime* | *every* | *join* | *randomChoice* | *use* | *extraction* | *subscription* | *reserve* | *free* | *confiscation*]

All instructions in a sequence (rule *SeqElemTask*) are executed one after each other. An optional delay can be specified.

instruction A**; wait(**d_1**),**
instruction B**; wait(**d_2**),**
instruction C**;**

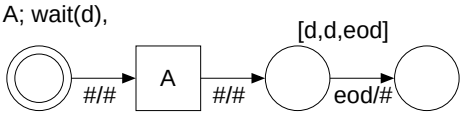

Figure 3: *translation of* ***A; wait(d),***

Inside a sequence of elementary tasks, most of the instructions are completely specified by state-machines. Their corresponding state-machines will be presented to offer a better understanding of their semantics. As first example, the figure 3 gives the translation into a state-machine of the instruction `A; wait(d),`.

3.4. Elementary Tasks

As the language is built upon C++, it is always possible to include C++ code everywhere inside scenarios by using four specific instructions: *include, declaration, implementation* and *destructor.*

Include ::= **include {***include_block***};**
Declaration ::= **declaration {***declaration_block***};**
Implementation ::= **implementation {***code_block***};**
Destructor ::= **destructor {***destructor_block***};**

The four rules are used to directly include specific C++ instructions. The *include_block* will be used to insert C++ #include instructions, while the *declaration_block* is used to declare C++ variables that could be used inside the scenario. The *code_block* is used to insert a sequence of instructions written in C++ and the *destructor_block* is used to give the specific C++ sequence of instructions that should be executed when the scenario will be terminated. Inside a scenario, it is possible to insert several *include, declaration* and *implementation* blocks, but only one *destructor* block is allowed per scenario. The content of those blocks is not analyzed during the parsing of the scenario language but will be analyzed during the C++ compilation.

The four following rules *if, switch, repeat* and *randomChoice* describe usual conditional expressions. We will describe them by giving examples. Instructions `if, repeat` and `switch` are executed inside a time-step and corresponds to their usual meaning. In those instructions and the following one `sub-sequence A` means any sequence of elementary instructions (rule *SeqElemTask*).

The `random choice` instruction allows to choose between different subsequences. Each of them is guarded by a condition and may have a weight (default weight is equal to one). The choice is made between all subsequences whose guarding condition is true, and by taking into account their relative weight.

```
if (c1) {                    switch (num_expr) {
    sub-sequence A1              case (1) {
}                                    sub-sequence C1
else {                           }
    sub-sequence A2              case (2 to  4) {
};                                   sub-sequence C2
                                 }
                                 else {
repeat {                             sub-sequence C3
    sub-sequence B1              }
} until( c );                };
```

Figure 4: *Instructions if, repeat and switch.*

The `waitfor` instruction is a very simple synchronization instruction: **waitfor(**condition**)**, and its effect is also very simple: *The execution of the current scenario is stopped until the condition becomes true.*

```
random choice {
    condition (cond1) {
        sub-sequence B1
    }
    condition (cond2) weight(5) {
        sub-sequence B2
    }
    ...
};
```

Figure 5: *Instruction random choice.*

The figure 6 shows the state machine for the `waitfor` instruction.

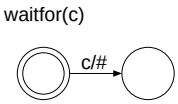

Figure 6: *translation of the waitfor instruction.*

The instruction `aslongas` allows to activate a sub-sequence and to execute it until the condition *c* becomes false:

```
aslongas (c) {
    sub-sequence A
};
```

As illustrated on figure 7, the meaning is: as long as condition *c* is true, run sub-sequence *A*. As *A* can be stopped at any time, two distinct state machines are necessary because of this termination condition (With one state machine, this could be possible only with one transition guarded by *(c==FALSE)* from each state of *A* to the final state.). The state machine on the left part of the figure 7 is in charge of starting a sub-state machine (`son`) and of controlling, at each time-step, the value of the condition *c*. When *c* becomes false, it terminates the current *A*. The second state machine is in charge of executing the sub-sequence *A*. If the end of *A* is reached while *c* is true, do *A* again at the next time-step. The `timegate` instruction avoids infinite looping if the state machine representing *A* has no duration.

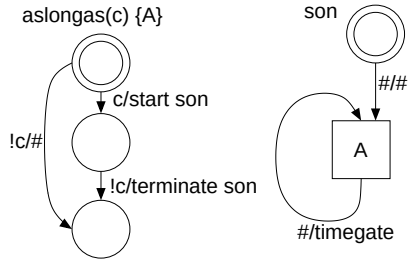

Figure 7: *translation of the aslongas instruction.*

The `eachtime` instruction is designed for reactive scenarios:

```
eachtime (c1) {
    sub-sequence A
} until (c2);
```

As shown in figure 8, the scenario continues just after the eachtime's launch, in parallel with the eachtime's loop (which runs via the *loop* state machine). Several instances of the sub-state machine *A* can coexist at the same time depending on the duration of those instances. The first instance of *A* is launched if *c1* is true when the initial state is reached, or when *c1* becomes true. But other instances are launched only if *c1* becomes false and then again true. The timegate instruction is used for the same reason as for the `aslongas`: only one *A* is launched during one time-step. The exit condition *until(c2)* is optional. When it is used, the termination of the *loop* state machine depends on *c2* but also on the termination of all the A_i. When *c2* is not used, there is no final state in the *loop* state machine. The termination of this state machine will then depend on the termination of its parent.

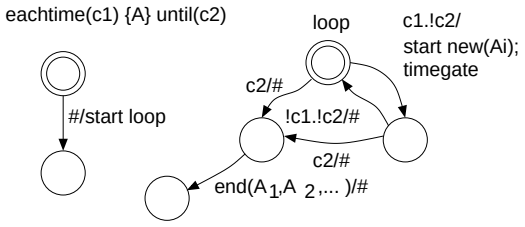

Figure 8: *translation of the eachtime instruction.*

The `every` instruction is used to launch an instance of a sub-scenario each time a delay is reached:

every (delay**) {**
    sub-sequence A
} until (c**);**

As for the `eachtime` instruction, a *loop* state machine is launched, and runs in parallel with the scenario (cf figure 9). The first instance of *A* is launched only after the first *d* delay. It is possible to use a termination condition *c*. If *c* occurs during a delay, no more *A* will be launched. Here again, dynamic state machines are used, because multiple instances of *A* can coexist at the same time, if the duration of *A* is longer than *d*.

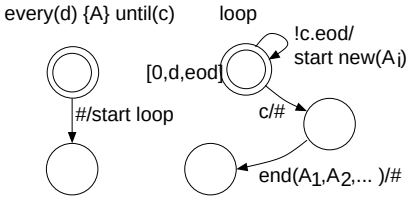

Figure 9: *translation of the every instruction*

The `join` instruction is used as a point of synchronization in a sequence of instructions. It permits to wait for the complete execution of all `eachtime` and `every` instructions currently running inside the current sequence. If the `join` instruction is used after an `eachtime` or an `every` instruction without an `until` condition, the part of the sequence following the `join` instruction will never be executed.

every (delay**) {**
    sub-sequence A
} until (c**);**
...
join;

Figure 10: *The join instruction.*

The `use` instruction (**use** sc_name**(**sc_p$_1$,..., sc_p$_n$**)**) permits to start the execution of an instance of the scenario *sc_name* as a sub-scenario of the current one. The current sequence of instruction is stopped until the end of the execution of the called sub-scenario. Due to the semantic of instructions `eachtime` and `every`, only a subpart of the scenario may be stopped and other parts can run in parallel with the sub-scenario. In the example shown in figure 11, depending on the evolution of the value of the condition c_2, the sub-scenario *scenarioA* can be either executed before, after or can overlap the execution of the *sub-sequence A*.

eachtime (c$_1$**) {**
    eachtime (c$_2$**) {**
        sub-sequence A
    };
    use scenarioA(param$_1$, ...**);**
    sub-sequence B
};
sub-sequence C

Figure 11: *Example of a call to a sub-scenario.*

To represent the sequence of instructions, their corresponding state-machines have to be concatenated by using transitions without condition nor action. A simplification algorithm is then used to reduce the number of states and transitions as illustrated by the example of figure 12. More complex simplifications can be operated but have not yet been implemented.

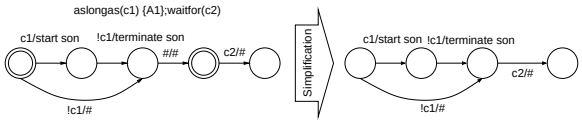

Figure 12: *An example of the composition and simplification of state-machines.*

3.5. Actor Management

An important part of scenario concerns the ability to dialog with autonomous characters. In our language, this is done via an interface of actor. For each object with which the scenario will have to interact, an interface has to be built. This interface describes the way the scenario can communicate with the actor. This actor can be an autonomous agent, but it can also be a camera, a button, a trigger or any reactive object. All actors will be member of the global set `ActorSet`, and will also belong to a specific type of actor. Different operations can be performed on sets (union, intersection and subtraction). We have also defined the `forall` instruction to execute of a sub-scenario *A* for all the components *x* of a set. A corresponding state machine *A(x)* is started for each instance of *x*. All these state machines will be launched at the same logic instant.

extraction ::= **forall** *ID* { **type** *ID* } **in** *ID*
    {*code***}**
reserve ::= **reserve (** *actorReserve*
    (**,** *actorReserve*)* **)**
    { success {*code***}**
    failure {*code***} }**
actorReserve ::= [**from** *ID* **to** *ID* { **count (***code***)** }
    | **actor** *ID*]
    { **priority(***code***)** }

free ::= **free** [[**(***set***)**] | [**actor** *ID*]]
confiscation ::= **on confiscation(***parameters***)** **{***code***}**
subscription ::= **on** *ID***(***parameters***)** **from** { **actor** *ID* } **type** *ID* **{***code***}**
parameters ::= *parameter* (**,** *parameter*)*
parameter ::= (*T_ID* | ***** | **&** | **::**)+

As several scenarios can run at the same time, we have defined a reservation mechanism which allows a scenario to be sure that it will be the only one to dialog with a set of actors. This is done via the `reserve` instruction. It is both possible to reserve specific actors (`actor` *ID*) or a certain number of actors belonging to a set (`from` *originSet* `to` *targetSet* { `count` **(***number***)** }). An optional priority allows a scenario to specify its level of interest for an actor or a group of actors. As some of the actors can already be used by other scenarios, this reservation may either be successful (execution of the `success block`) or not (execution of the `failure block`). Due to the priority specification, a scenario has the ability to take an actor to another one: the `confiscation` instruction allows a scenario to be informed of this confiscation. The `free` instruction allows a scenario to release an actor or a set of actors. For example, in the following scenario (*crampGroup*), some actors are selected inside the set *_visitors* to participate to the scenario. Eachtime another scenario confiscates one of those actors, it is suppressed from the set.

```
scenario crampGroup(
  float xGoal, float zGoal,
  ActorSet * ptrVisitorSet,
  int requested_number,
  int _priority
)
{
include {
  #include "ActorSet.h"
};
declaration {
  ActorSet _visitors;
  ActorSet _reserve;
  int _number;
  bool _inProgress;
};
c++ {
  _number = ptrVisitorSet.size();
  if (_number > requested_number)
    { _number = requested_number; }
  _visitors = *ptrVisitorSet;
  _inProgress = true;
};

reserve (
  from _visitors to _reserve
  count(_number) priority(_priority)
)
{
  success { }
  failure {
    cout«"Warning !!! Problem during the
    initial reservation of actors"«endl;
    exit(0) //termination;
  }
};

forall visitor in _reserve
  { ptrVisitorSet -> erase(dynamic_cast
    <actortype *>(visitor)); };

on confiscation(actortype *
                confiscated_actor)
{
  _reserve.erase(dynamic_cast<actortype *>
                 (confiscated_actor));
  _number = _number - 1;
};

aslongas (_number>0 && _inProgress)
{
  use moveGroup(&_reserve,xGoal,zGoal);
  c++ { _inProgress = false; };
};
free(_reserve);
}
```

The instruction `on` *EventName*`(`*transmitter, other parameters*`)` `from` { `actor` *ID* } `type` *ID* {*code*} is executed when the scenario receive an event *EventName*, and by this way allows a scenario to receive information from actors. For example, in the following scenario, a click on a button will generate the call to a specific sub-scenario *SecondaryAction*.

```
scenario scenarioClick(
  ActorSet * ptrBadGuy,
  ButtonActor * button
)
{
  include {
    #include "ButtonActor.h"
    #include "ActorSet.h"
  };
  declaration { bool _click; };

  on Leftclick(ButtonActorGenerator *
               transmitter)
  from actor button type ButtonActor
  { _click = true; };

  c++ { _click = false; };
  eachtime(_click)
  {
    c++ { _click = false; };
    use SecondaryAction(ptrBadGuy);
  };
  join;
}
```

3.6. Scheduling

At the lower level of granularity, a scenario is composed of a sequence of elementary tasks. The scheduling of elementary tasks is expressed by the concatenation of corresponding state machines. The structure of a scenario can also be specified as a hierarchy of parallel sub-scenarios and scheduling constraints (temporal relations between them). Such kind of scenario will be written by using the following syntax:

instance ::= **instance** *ID* **of** *ID* **(***Params***);**
schedule ::= **schedule {**
[*allenRel* | *instantRel*]$^+$ **};**
allenRel ::= *ID* [
equals | **before** | **after** | **meets** |
met by | **overlaps** |
overlapped by | **during** |
contains | **starts** | **started by** |
finishes | **finished by**
] *ID* **;**
instantRel ::= [**start** | **end**] **of** *ID*
instantRel0
[**start** | **end**] **of** *ID* **;**
instantRel0 ::= [{ **(** *duration* **)** } [**before** | **after**] |
equals]

Each scenario can be represented by a temporal interval with its initial and final instants. All temporal constraints between sub-scenarios are expressed in a scenario by using Allen's and instant logic. J. Allen [1] has defined a logic model which permits to describe all possible relative positioning of two temporal intervals along an axis (cf figure 13). In the instant logic, the relative positioning of two instants along the temporal axis is either `before`, `after` or `equals`.

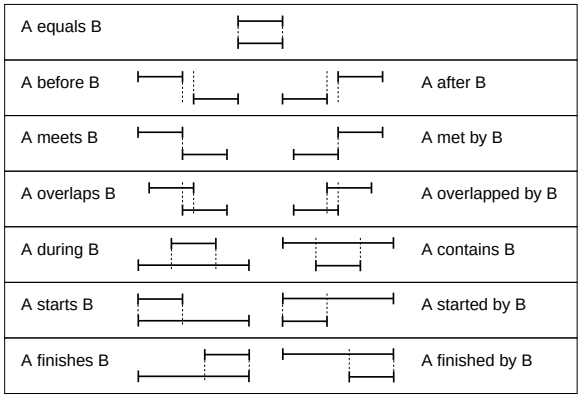

Figure 13: *Allen's relations between two intervals.*

The example of figure 14 illustrates the use of those instructions. Each Allen's relation can be expressed as constraints on the four extremities of the two intervals. For those temporal relations (Allen and instant), we did not allow to specify disjunction but only conjunctions of them, as it is

scenario name(parameters) **{**
instance sc_1 **of** $subScenario_1$(parameters) **;**
instance sc_2 **of** $subScenario_2$(parameters) **;**
instance sc_3 **of** $subScenario_3$(parameters) **;**
schedule {
sc_2 **starts** sc_1**;**
sc_3 **finishes** sc_1**;**
end of sc_2 **(**12.0**) before start of** sc_3**;**
};
}

Figure 14: *Example of a scheduled scenario.*

a NP hard problem to find a solution to the description in the general case. From the set of relations a directed valuated graph is constructed whose nodes are extremities of intervals. This description is logically consistent if there is at least one solution to order extremities of segments. The consistency of the description can be checked by searching for circuits during a unique graph traversal.

For each scheduling block inside a scenario a corresponding scheduler will use the graph structure at run-time to schedule sub-scenarios execution. However, errors can occur at run-time if some scenarios do not respect constraints imposed in the partial order. This will be detected by the scheduler which will destroy such inconsistent sub-scenarios: the global consistency of a scenario description remains under the control and responsibility of the programmer.

3.7. Conclusion

We have developed a specific compiler for our scenario language. Each scenario is compiled and transformed into an equivalent C++ code, and by linking it with precompiled libraries, a scenario manager object is obtained that can be used in a simulation application[7].

4. Scenario Example

Different scenarios have been developed to illustrate the capabilities of our scenario language[6]. One of them is presented; it takes place inside a virtual museum, visited by a group of autonomous agents. In this scenario, actions of the user will determine the execution of different sub-scenarios. The goal of the user consists in finding and taking a statuette on its pedestal and coming back to put it on a second pedestal. To perform his task the user uses a hand made peripheral composed of two mouses and a magnetic sensor put on board. A yellow ray represents its direction of navigation. A blue ray, whose orientation is controlled by the magnetic sensor (a flock of bird with 6 degrees of freedom) will be used by the user to select and move the statuette inside the museum. Different buttons on the peripheral allows the user to go straight, and forward, to turn left and right and to

take and release the object selected by the blue ray. During the displacement of the user in the museum, three groups of visitors (*Group1, Group2* and *Group3*) will be controlled by different scenarios to cramp his navigation. Fourteen sub-scenarios have been specified and are scheduled as follows:

```
schedule {
  start of robot equals start of visit;
  start of startGroup1 equals start
    of visit;
  start of startGroup2 equals start
    of visit;
  start of supervision1 equals start
    of visit;
  supervision1 meets crampGroup1a;
  supervision1 meets crampGroup2a;
  supervision1 meets supervision2;
  supervision2 meets supervision3;
  supervision2 meets crampGroup3a;
  crampGroup3a meets crampGroup3b;
  supervision3 meets crampGroup3b;
  supervision3 meets supervision4;
  supervision4 meets crampGroup1b;
  supervision4 meets crampGroup2b;
};
```

Figure 15 illustrates the obtained scheduling between the different sub-scenarios. The *robot* scenario controls the robot displacement, it starts at the beginning of the main scenario and terminates when the robot has finished to perform its activity. The *startGroup1* and *startGroup2* scenarios ask two groups of visitors to go on the way of the user. *startGroup*[*1, 2*] and *crampGroup*[*1a, 1b, 2a, 2b, 3a, 3b*] are different instances of the same scenario *crampGroup* described above. Those scenarios are used to cramp the user displacement depending on its own location inside the virtual museum. *supervision*[*1, 2, 3*] are different instances of the scenario *supervision* given below. They are used to track the displacement of the user inside the museum. A rectangular trigger is used and when the user enters the corresponding area the scenario is ended. Due to the temporal relations between both families of scenarios (*crampGroup* and *supervision*), the user journey indirectly control the displacement of the different groups of visitors. For example, the scenario *supervision3* terminates when the user is near the statuette. By its ending, the scenario *crampGroup3b* is started and asks the third group to go around the user (cf image 5 of figure 16).

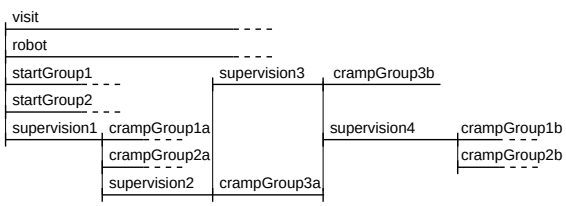

Figure 15: *Scheduling of the sub-scenarios*

```
scenario supervision(
  MuseumTriggerActor * manager,
  float xCenter, float zCenter,
  float width, float length
)
{
  include {
     #include "MuseumTriggerActor.h" };
  declaration { bool _enter; };
//----- init
  c++ {
    _enter = false;
    manager -> trackPosition(xCenter,zCenter,
                             width,length);
  };
//----- waiting for the user
  on enterArea( MuseumTriggerActor * sender,
                float x, float z)
  from actor trigger type MuseumTriggerActor
    { _enter = true; };
//----- block the end of the scenario
//----- until the user enters the area
  waitfor(_enter);
//----- end of the tracking
  destructor { manager -> stopTracking(); };
}
```

The scenario *moveGroup* presented now is used as sub-scenario of the *crampGroup* one to control the displacement of a group of visitors to a new location (action *moveTo* sent to each member of the set *_input*).

```
scenario moveGroup
(
  ActorSet * input,
  float xGoal, float zGoal
)
{
include {
  #include "ActorSet.h"
  #include "MapUser.h"
};

declaration {
  int _number;
  int _cpt;
  ActorSet _input;
};

// ------ init
c++ {
  _number = input->size();
  _cpt = 0;
  _input = *input;
};

// ------ displacement
forall humano type ptrMapUser in _input
{
   humano -> moveTo(xGoal,zGoal);
};
```

```
// ------ counting arrived actors
on reachedposition(GMapUser * sender,
                   float x,float z)
from type MapUser
{
  actortype * ancestor = dynamic_cast
               <actortype *>(sender);
  if (_input.find(ancestor) != _input.end())
  {
    if ( (pow((xGoal - x),2) +
          pow((zGoal - z),2)) > (2*2))
    {
      sender -> moveTo(xGoal,zGoal);
    }
    else { _cpt++; }
  }
};
// ------- waiting fo the end
waitfor(_cpt == _number);
}
```

Images of figures 16 illustrate different steps of the scenario execution.

1. The red area represents the target area of the statuette. The blue robot on the right, is on the way to take its own statuette. Some autonomous agents can be seen on the background. Their behaviour consist in being in the way of the visitor during his search of the statuette.

2. The user has started to move in the direction of the statuette. In reaction, scenarios *crampGroup1a* and *crampGroup2a* asked two groups to move in its direction: the visitor will have to avoid them.

3. After having avoided those groups, a third one is moving in its direction (scenario *crampGroup3a*). The statuette is now clearly in the field of view of the visitor.

4. The visitor takes the statuette by using the blue ray.

5. At this moment, when he turns around, the user can see a group of autonomous agents just around him (scenario *crampGroup3b*). The red arrow shows the location of the robot who is moving slowly in the direction of the other statuette.

6. The robot takes its own statuette.

7. On the way back, the two first groups will cross again the visitor path (scenarios *crampGroup1b* and *crampGroup2b*).

8. The visitor leaves the statuette at its target location. This is the end of the manipulation and of the scenario.

5. Conclusion

In this paper we have presented the architecture of a scenario authoring system, based on a high level description language. Internal representation is based on the use of Allen's logic and of rewriting rules to produce hierarchical state machines. Dedicated instructions of the language concerning Scheduling and Actor Management issues has also been presented. Using C++ code inside the scenario provides the ability to describe a large variety of scenaristic elements without being limited to a specific description language. The virtual museum example illustrates one of the advantage of our approach: the temporal synchronization of different scenarios allows to reach the objective without having to write one big scenario integrating all features. Some of the scenarios are generic and can be used for several kinds of application, like the *supervision* scenario to trigger an area or the *moveGroup* scenario asking a group of characters to reach a specific location. By using this scenario language in common with triggers inside applications integrating controllable autonomous characters[7], it is very easy to describe three of the four categories of scenario given in section 2: rules, ambient and sequence of actions. Concerning the goal oriented family of scenario it is necessary to add an action planning functionality which is not part of our scenario language, but it can easily be combined with.

Nevertheless, our scenario language approach, combining the use of dedicated instructions in common with C++ parts, restricts the use of scenario language to programmers. Despite the benefits of the language approach described earlier, the description of behaviour remains quite difficult for people who are not computer scientists. Therefore we are working on a higher-level specification language, in order to allow behavioural specialists to specify and test their models in an interactive simulation. Work in progress concerns the connection of our scenario programming language with the XML output of a scenario authoring tool[9]. This authoring tool allows scenarists to describe scenarios of interactive fiction in natural language. A syntactic and spatio-temporal semantic analysis of phrases corresponding to actions is performed and the result consists in a decomposition of each action phrase into several parameters (like the nature and manner of the action, the subject and the object of the action) and in a specification of their relative positioning on a common temporal axis.

References

1. J. Allen. An interval based representation of temporal knowledge. In *Proceedings of the seventh International Joint Conference on Artificial Intelligence*, pages 221–226, Aug. 1981.

2. N. Badler, B. Reich, and B. Webber. Towards personalities for animated agents with reactive and planning behaviors. *Lecture Notes in Artificial Intelligence, Cre-*

ating Personalities for synthetic actors, (1195):43–57, 1997.

3. R. Bindiganavale, W. Schuler, J. Allbeck, N. Badler, A. Joshi, and M. Palmer. Dynamically altering agent behaviors using natural language instructions. In C. Sierra, M. Gini, and J. Rosenschein, editors, *International Conference on Autonomous Agents*, pages 293–300, Barcelona, Spain, June 2000. ACM Press.
4. B. Blumberg and T. Galyean. Multi-level direction of autonomous creatures for real-time virtual environments. In *Siggraph*, pages 47–54, Los Angeles, California, U.S.A., Aug. 1995. ACM.
5. J. Cremer, J. Kearney, and P. Willemsen. Directable behavior models for virtual driving scenarios. *Transactions of the Society for Computer Simulation International*, 14(2), June 1997.
6. F. Devillers. *Langage de scénario pour des acteurs semi-autonomes*. PhD thesis, University of Rennes I, Sept. 2001.
7. F. Devillers, S. Donikian, F. Lamarche, and J. Taille. A programming environment for behavioural animation. *Journal of Visualization and Computer Animation*, 13:263–274, 2002.
8. S. Donikian. HPTS: a behaviour modelling language for autonomous agents. In *Fifth International Conference on Autonomous Agents*, Montreal, Canada, May 2001. ACM Press.
9. S. Donikian. DraMachina: an authoring tool to write interactive fictions. In S. Gobel, N. Braun, and U. Spierling, editors, *TIDSE'03, first International Conference on Technologies for Interactive Digital Storytelling and Entertainment*, Darmstadt, Germany, Mar. 2003. Fraunhofer IRB Verlag.
10. S. Göbel, T. Smithers, and M. Schnaider. art-e-fact: a generic platform for the creation of interactive art experience in mixed reality. *Computer Graphik Topics*, 14:18–20, Apr. 2002.
11. D. Harel. Statecharts : a visual formalism for complex systems. *Science of Computer Programming*, 8(3):231–275, 1987.
12. J. Kearney, J. Cremer, and S. Hansen. Motion control through communicating, hierarchical state machines. In G. Hegron and O. Fahlander, editors, *Fifth Eurographics Workshop on Animation and Simulation*, Oslo, Norway, Sept. 1994.
13. J. Kearney, P. Willemsen, S. Donikian, and F. Devillers. Scenario languages for driving simulations. In *DSC'99*, Paris, France, July 1999.
14. S. Lehane. Digital crowds: game-style ai controls adapted for behavioral animation. *Film & Video*, 19(7):27, July 2002.
15. S. R. Musse. *Human Crowd Modelling with Various Levels of Behaviour Control*. PhD thesis, EPFL, Lausanne, Suisse, Jan. 2000.
16. H. Noser and D. Thalmann. Sensor based synthetic actors in a tennis game simulation. In *Computer Graphics International'97*, pages 189–198, Hasselt, Belgium, June 1997. IEEE Computer Society Press.
17. K. Perlin and A. Goldberg. Improv: A system for scripting interactive actors in virtual worlds. In *SIGGRAPH'96*, 1996.
18. C. Reynolds. Computer animation with scripts and actors. *Computer Graphics*, 16:289–296, July 1982.
19. G. Ridsdale and T. Calvert. Animating microworlds from scripts and relational constraints. In N. Magnenat-Thalmann and D. Thalmann, editors, *Computer Animation '90 (Second workshop on Computer Animation)*, pages 107–118. Springer-Verlag, Apr. 1990.
20. U. Spierling, D. Grasbon, N. Braun, and I. Iurgel. Setting the scene: playing digital director in interactive storytelling and creation. *Computer & Graphics*, 26(1):31–44, Feb. 2002.
21. D. Wavish and D. Connah. Virtual actors that can perform scripts and improvise roles. In *Autonomous Agents'97*, 1997.
22. P. Willemsen. *Behavior and Scenario Modeling for Real-time Virtual Environments*. PhD thesis, Department of Computer Science, University of Iowa, 2000.

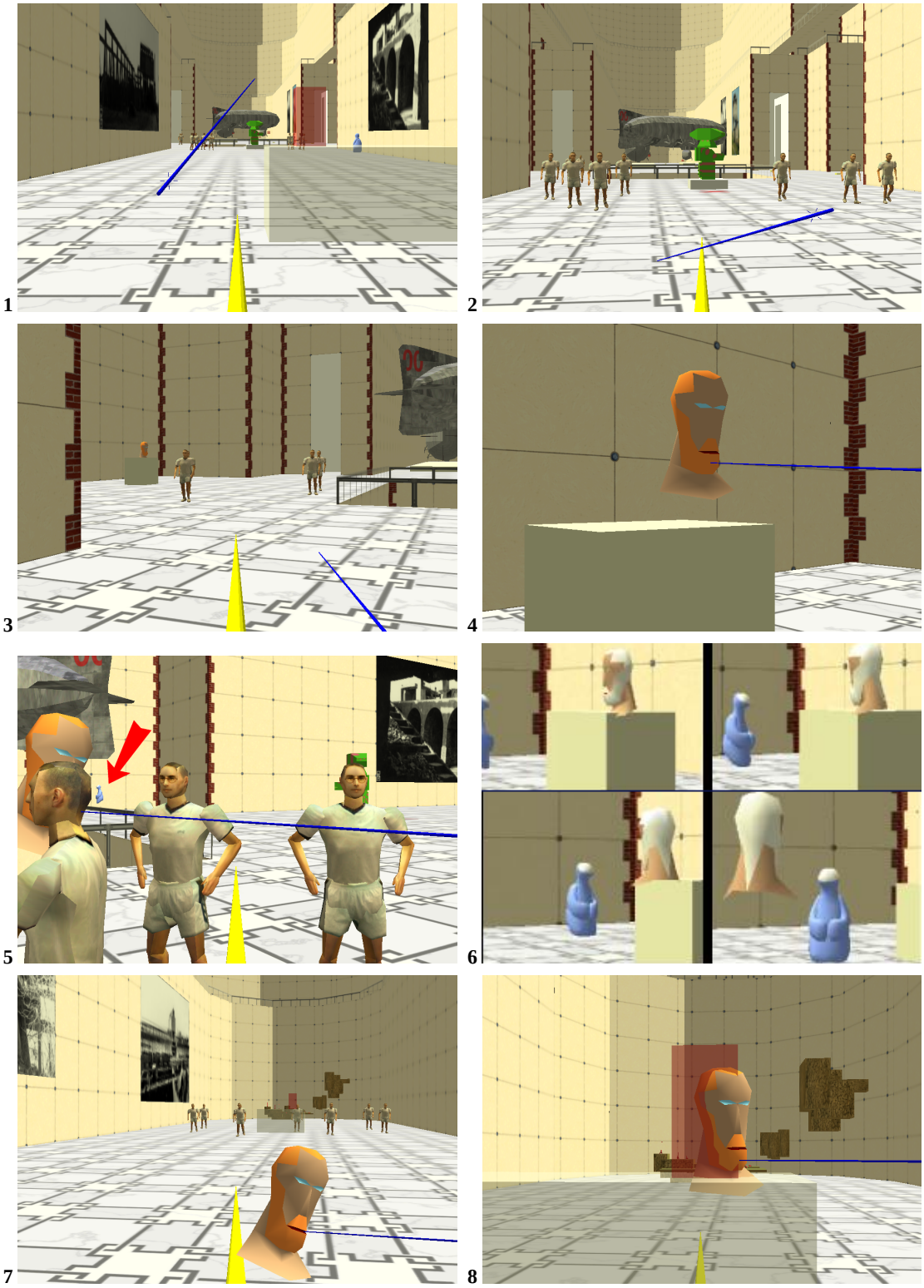

Figure 16: *Snapshots taken from an interactive experiment.*