**Scenario Languages for Driving Simulation**

Joseph Kearney

Peter Willemsen

Department of Computer Science

University of Iowa

Iowa City, Iowa

United States

*kearney@cs.uiowa.edu, willemsn@cs.uiowa.edu*

Stéphane Donikian

Frédéric Devillers

IRISA

Campus de Beaulieu

Rennes, France

*donikian@irisa.fr, fdeville@irisa.fr*

## Résumé

Dans cet article, nous présentons les besoins nécessaires à la réalisation de langages de programmation de scénario et nous considérons une approche textuelle de spécification de ces langages. Par programme scénaristique, nous entendons le code qui permet de déterminer quels objets se trouvent dans l'environnement de simulation et comment le comportement de ces mêmes objets doit être coordonné afin de produire des situations prédictibles pour le conducteur du simulateur de conduite. La programmation de scénario est un élément important pour les applications qui utilisent la simulation pour l'entraînement des conducteurs et l'étude de leurs comportements.

## Abstract

In this paper we examine the requirements for scenario control programming languages and consider approaches for designing textual-based scenario specification languages. By scenario programs, we mean the code that determines what objects are in the simulation environment and how object behaviors are coordinated to produce predictable experiences for the driver in a ground vehicle simulator. Scenario programming remains a challenging problem for applications that use simulation for driver training or studies of driver behavior.

## Introduction

To one point of view, the code that controls a driving simulation scenario can be treated as the behavior of a disembodied object. This suggests that the same languages used to program the behaviors of simulated objects with material properties (such as vehicles, pedestrians, and traffic lights) can be used to program the modules that manipulate the environment by creating objects, destroying objects, and coordinating the actions of other objects. A number of groups working on simulation have converged on hierarchies of state machines as a model for programming object behaviors [1, 2, 3, 4]. The state machine model has much to offer for scenario programming. However, the demands of scenario control differ in many respects from the demands of behavior programming. By focusing on the specific needs of scenario control we believe it is possible to design simple, expressive scripting languages tailored to the needs of scenario programming.

## What is a Scenario?

The scenario component of a driving simulation carries the responsibility for orchestrating the activities of the semi-autonomous agents that populate the virtual driving environment. In common practice these agents are programmed as independent entities that perceive the surrounding environment and under normal circumstances behave as autonomous agents [2, 5, 6, 7]. Thus, vehicles travel on roads, avoid collisions with other vehicles, and obey traffic control devices. Pedestrians navigate on sidewalks and crosswalks, maintain a polite separation between other pedestrians, and watch out for vehicles. Traffic lights cycle through green, yellow, and red states.

The primary purpose of the behavior model for a material object is to produce input values for the virtual actuators that control the object's degrees of freedom. For a vehicle, the control values are typically steering angle and the position of the accelerator pedal. The control variables for an articulated object could be joint torques. For a traffic light, the control variable determines which light is illuminated.

Behavior models are complex, carefully crafted computational objects. A multi-faceted behavior, like driving, involves situation dependent responses, simultaneous consideration of many factors, and integration of information about road geometry, road logic, the activities of near-by objects. To manage complexity, behaviors are typically decomposed into sub-behaviors responsible for specific aspects of driving. Values returned by sub-behaviors must be integrated to determine the appropriate set of control variables. A number of researchers have adopted hierarchical state machines as a computational framework for programming behaviors.

When a driving simulation is populated with vehicles programmed to operate autonomously, they will create plausible, ambient traffic. However, in most experiments and training runs we want to create a predictable experience. This is accomplished through direction of object behaviors [5, 8, 9]. To facilitate coordination of activities, objects are built with interfaces through which they can receive instructions to modify their behaviors. For example, a vehicle could be directed to reduce its velocity, turn left at the next intersection, or change lanes. Scenario processes control the evolution of the

simulation by strategically placing objects in the environment and guiding the behaviors of objects to create desired situations. For example, a vehicle might be directed to encroach into the interactive driver's lane or abruptly stop in front of the driver.

The code that controls a scenario shares much in common with the behavior models of material objects. The principle activities of a scenario model are to create objects, destroy objects, and send directives to objects in order to create desired circumstances. Scenarios can be programmed in similar fashion to the behaviors of other entities, creating a disembodied agent that invisibly manipulates the environment. However, we've found that the state machine architectures used for programming the behaviors of material objects are unnecessarily complicated for scenario behaviors. Scenario behaviors never use many of the standard features of the state machines such as data flow channels for communicating continuous streams of input and output values, initialization functions for creating stable starting configurations, and pre-activity functions that prepare input for sub-state machines before they are executed.

Additionally, our experience is that scenario behaviors have a different character than the behaviors of substantive objects. Scenario objects are principally concerned with sequencing activities to create a coherent experience for the interactive driver. The code is predominated by event detection, synchronization of activities, action sequencing, and message sending. In contrast, behaviors of material objects are shaped by the need to know how the immediate environment constrains behavior and determining how to balance simultaneous demands.

## Scenario Expression

What should a scenario programming language look like? One option is to design a language to explicitly define the components of the state machine: states, transitions, and an initial state. For example, a simple state machine composed of four states could be defined as follows:

```
sm1 <activity1>;
sm2 <activity2>;
sm3 <activity3>;
sm4 <activity4>;

transition sm1 sm2 <cond1>;
transition sm2 sm3 <cond2> and <cond3>;
transition sm2 sm4 <cond2> and not <cond3>;
initial state sm1;
```

Such literal representations of state machines are often verbose and difficult to read. The state machine code for simple activity sequences poorly reflects the relationships among activities. For example, compare the code above to the following script (written in procedurally-styled pseudo code):

```
DoInSequence
Do <activity1> until <cond1>;
Do <activity2> until <cond2>;
if <cond3>
then Do <activity3>
```

```
else Do <activity4>
end
```

The temporal order of activities is (we believe) clearer in the latter, more procedural-looking sequence. One of the common difficulties in programming state machines is that textual-based state machine code tends to have poor locality. As shown in the first example, control logic statements are usually separated from the context dependent activities to be performed when states are active. This leads to disjoint programs that are difficult to read. One solution is to use hybrid interfaces that mix textual code fragments and state diagrams. While such multi-faceted interfaces offer rich means of expression, we believe there is merit in exploring possibilities for text-only languages. They are simple to implement and in many situations purely textual code is much easier to compose and read than are multi-component specifications. Several driving simulators provide scripting languages for programming driving scenarios. van Wolffelaar and van Winsum [10] developed a simple and elegant language for programming scripted scenarios called the Scenario Specification Language. The SCANeR simulator [11] also includes sophisticated scripting facilities. Scenario scripting is also important component in virtual environments systems such as Alice [12].

## Design Issues

We envision a suite of languages each adapted to special purposes: A light-weight, interpreted language for run-time interaction; a medium weight compiled language for off-line specification of more complex scenarios with temporal dependencies; and a heavy weight compiled language with the full power of hierarchical state machines for creating libraries of intricate scenarios that can be instantiated in the lighter weight languages.

The research groups at Irisa and Iowa are collaboratively investigating scripting languages to explore a range of alternative designs. Both groups have already developed compiled state machine languages for behavior programming. At Irisa this language is based on Hierarchical Parallel Transition Systems (HPTS) [1] ; at Iowa it is based on Hierarchical Concurrent State Machines (HCSM) [3]. We are currently exploring specialized languages tailored to the needs of scenario programming. In the remainder of the paper, we discuss a range of issues from the basic look and feel of the language to event detection and activity scheduling. Throughout the paper, we give examples illustrating the design alternatives. The examples are also meant to convey the style of programming we envision for scenario programming. At the end of the paper, we present an extended example.

## Simple Activities

We want simple, compact ways to define state dependent activities. For example, in order to discourage tailgating we may want scenario vehicles to slow down whenever the driver approaches unacceptably close to another vehicle. The following code fragment illustrates the style of expression we seek.

```
whenever (driver following distance too small )
{
send <reduce speed> to lead vehicle
}
```

Our intention is that this statement will create a simulation process that regularly monitors the separation between the driver's vehicle and the vehicle in front of it (if there is one). Any time this separation distance is unacceptably small (where the computation of the appropriate separation distance should depend on the speed of the lead vehicle), the process will direct the vehicle in front of the driver to reduce its speed. The process will continue to send messages as long as the separation is too small causing the lead vehicle to progressively reduce its speed. Should the driver change lanes and later tailgate another vehicle, then the process will send directives to this vehicle to reduce its speed. Sometimes we may want to nest scenario statements to set a more complex context for an activity. For example, the following nested scenario statement is enclosed in an "aslongas" statement. The body of the "aslongas" statement is active until its guard becomes false. Thus, the following "whenever" statement will cease to influence the simulation as soon as the driver is not on a four-lane highway.

```
aslongas(driver.road is a four-lane highway)
  {
  whenever(driver in center lane and
       driver following distance small)
       {
       send <change lane right> to lead vehicle
       }
  }
```

As long as the driver remains on a four-lane highway, whenever the driver is in the center lane, vehicles will be directed to change lanes whenever the driver approaches them from behind.

## Event Management

The ordering of scenario activities frequently depends on situations and events that occur at unpredictable times during the simulation. Scenarios must monitor the simulation to detect when the prerequisite conditions required to initiate directed activities are satisfied. Sometimes the necessary conditions can be characterized as predicates of state variables. For example, in the tailgating example given above, a message is sent to the lead vehicle whenever the driver's following distance is too small. Such conditions can be expressed as boolean expressions of state variables obtained through queries to the simulation database.

Frequently, however, the prerequisite conditions depend on changes in state such a vehicle crossing a line on the road or entering an intersection. A history of state information must be maintained in order to detect events conditioned on changes in state variables. We introduce a sensor as an event monitor. Sensors are defined by a state change to be detected. The sensor automatically maintains the state memory needed to test for the change and creates a process to monitor the specified change condition. Whenever the sensor detects the change condition, an event is registered.

Scenario activities can be conditioned on these sensor events. In the following example, a sensor is created that triggers an event whenever the driver enters a particular region. The subsequent "whenever" statement causes a directive to be sent every time the driver enters the region.

```
on event <"enter", driver, region> trigger ENTRY_EVENT;
  whenever ( ENTRY_EVENT ) {
     send < message >;
  }
```

Sensors are parameterized by event type, object, and region. Regions represent intervals of roads or intersection areas. Regions may be static or dynamic. Dynamic regions are connected to moving objects. For example, a sensor may detect when the driver enters the blind spot of another vehicle. We imagine several different types of events will be useful: one-shot events that report only the first occurrence of an event, multi-shot events that report all occurrences, and time-staged-events that report the events only during a specified time periods.

## Abstraction

It is important to be able to encapsulate scenarios into parameterized modules that can be reused and combined with other modules. We call such a parameterized body of code a scenario. To illustrate modularization, we present a scenario that controls the timing of a traffic light. The traffic light behavior regulates the cyclic changes in the phase of the light. The traffic light behavior model is programmed to respond to a directive to be at a specific point in its cycle at a target time in the future. For example, Figure 1 illustrates how a traffic light would respond to a directive to be at the transition from yellow to red at a time 10 minutes from the current time. When the light receives such a directive it uniformly compresses or expands cycles between the current time and the target time, so that the light will be at the appropriate point in its cycle at the target time. This process maintains the normal progression of states and makes the smallest possible change to the normal cycle.

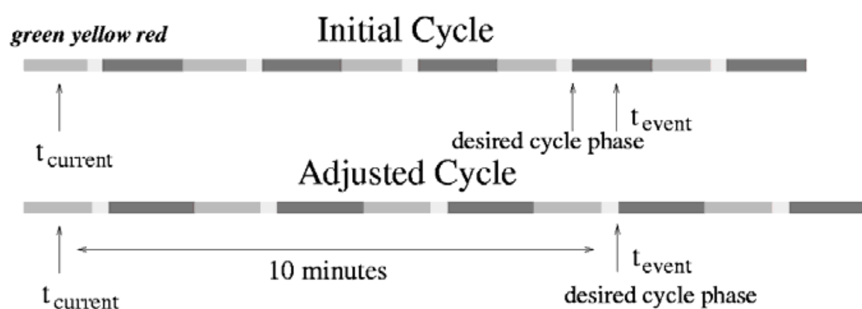



**Figure 1.** The basic behavior of the traffic light is to cycle between green, yellow, and red states. The behavior responds to a directive to synchronize the cyclic pattern so that at a time in the future ($t_{event}$) the light will be at a specific point in the cycle

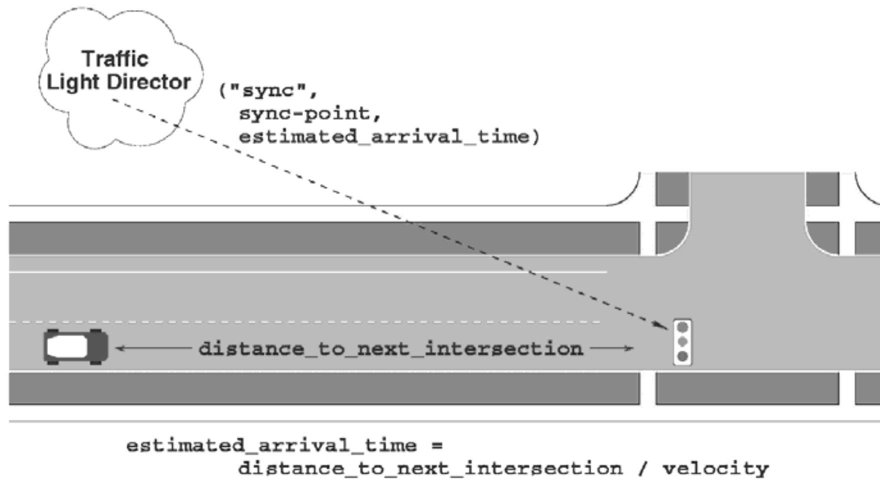
(labeled desired cycle phase). In the example shown the cycle duration is lengthened to insure that the light will be at the transition from yellow to red at time $t_{event}$.

The scenario below takes as parameters a traffic light and a synchronization point. The scenario extrapolates the time at which the driver will arrive at the next intersection based on the driver's current velocity and the distance to the intersection. The traffic light is directed to adjust its cycle so that at the target time it will be at the synchronization point. Once the scenario is invoked, it will continue to send directives to the traffic light on each simulation step. Thus, the light will adjust it's cycle duration whenever there is a change in the speed of the driver's vehicle.

```
scenario Traffic_light_director (light, sync_point)
 {
  // Do forever
  whenever (TRUE)
   {
    // Compute driver's estimated time of arrival at the next intersection.
    estimated_arrival_time = driver.distance_to_next_intersection/ driver.velocity;
    // Send a "sync" message to the traffic light telling it be at the desired state
    // (sync_point) at the expected arrival time
    send <"sync", sync_point, estimated_arrival_time> to light;
   }
 }
```

The code includes database references to object attributes using a record-like syntax. For instance, driver.velocity refers to the current velocity of the driver's vehicle. These statements can be simply translated into existing queries in our simulators. To demonstrate scenario invocation, we present a scenario named Light_Sequencer that uses the Traffic_light_director to control the timing of a sequence of traffic lights. The Light_Sequencer scenario takes as input an integer n. Once invoked, the scenario will direct the traffic lights at the first n-1 intersections crossed by the driver to turn yellow just before the subject enters the intersection. At the nth intersection, the light is to turn red just before the subject enters the intersection. A new Traffic_light_director scenario is created for each successive intersection. As the driver approaches the intersection, the Traffic_light_director sends directives to the traffic light at the upcoming intersection (see Figure 2.) When the driver arrives at the intersection, the current Traffic_light_director is terminated. As the driver leaves the intersection a new Light_Sequencer scenario is created and the current scenario is terminated.

**Figure 2.** The Light_Sequencer scenario creates a succession of n




Traffic_Light_Director scenarios. Each Traffic_Light_Director directs the behavior of a single traffic light to synchronize its cycle to the approach of the driver. When the driver enters the intersection, the current Traffic_Light_Director is terminated.

```
scenario Light_Sequencer (i)
{
        // Determine the intersection the driver is approaching.
        intersection = NextIntersection( driver.road );
        // Determine the traffic light controlling the next
        // intersection towards which the driver is heading
        traffic_light = GetLight( driver.road, driver.lane );
        if (i > 1)
                sync_point = transition_to_yellow;
        else
                sync_point = transition_to_red;
        // Create a traffic light director.
        tld = create( Traffic_light_director( traffic_light, driver, sync_point ) );
        on one_shot_event <"enter", driver,  intersection> trigger ENTRY_EVENT;
        on one_shot_event <"exit", driver,  intersection> trigger EXIT_EVENT;
        // Monitor subject entry into an intersection and upon entry,
        // terminate the traffic_light director.
        when ( ENTRY_EVENT ) {
                // terminate the traffic light director
                destroy( tld );
        };
        // Monitor subject exit out of the intersection mask and
        // upon exit, create a new intersection manager, if necessary,
        // to handle the next intersection
        when ( EXIT_EVENT ) {
                if (i > 1)
                        create( Light_Sequencer( i-- ) );
        };
        // terminate self by exiting the prototype
        exit;
}
```

Note that this scenario makes no assumption about the driver's route through the environment. Traffic lights are dynamically accessed in response to the driver's unrestricted turning decisions. However, it is assumed that every intersection is regulated by a traffic light. Note also that the scenario recursively invokes itself.

## The Semantics of Scenario Invocation

What does it mean to invoke a scenario? We've considered three execution models that define different relationships between the fates of the calling and called scenarios. Depending on the circumstances, one or another method may be preferred by an author. All three models can be made available in a language provided that there is a clear syntactic distinction to identify which invocation method is to be used when a scenario is invoked.

The first model we consider treats invocation as a transfer of control from the calling scenario to the invoked scenario (much like a procedure call). In this model, execution of the calling scenario is suspended until the invoked scenario terminates. Because there is a single, linear flow of control, this model is simple to understand and it is easy to track the execution of a scenario during a simulation.

The second and third models both treat scenario invocation as the spawning of a new process (similar to a fork.) As in a fork, execution in the calling scenario continues concurrently with execution of the invoked scenario. Because the sub-scenarios are parameterized, there can be many instances of sub-scenario running in parallel. For example, consider a sub-scenario that controls the flow of traffic through an intersection by directing the turn decisions of vehicles that drive through the intersection. A supervisory scenario could create separate instances of this scenario for a set of intersections.

We need to consider what happens to a spawned scenario when its parent (i.e. the scenario that spawned it) is terminated. We could choose to link the continued execution of the spawned scenario to its parent. Thus, if the parent terminates, all spawned scenarios are automatically terminated. The persistent coupling between scenarios and their creators leads to a tightly integrated flow of control.

Another possibility is to treat the spawned scenario as a completely independent computational process. In this model, termination of the parent has no influence on the continued execution of the spawned scenario. This methods lead to a highly distributed, free-form style of programming. Because all scenarios called in this manner continue to run they are individually terminated, care must be taken encode explicit terminations when they reach the end of their useful lifetime.

## Scheduling Scenarios

The scenario abstraction provides a means to encapsulate scenario activities that can be invoked as needed from other scenarios. The temporal relationships among sub-scenarios are often critically important. Using the first model of scenario invocation presented above (whereby control is transferred to the sub-scenarios), we can define a

series of sub-scenarios that are to follow one another in linear sequence. It is very useful to have mechanisms to define more complex temporal relationships.

For example, we may want to specify that scenario A must start after the scenario B has been activated. To define these less restrictive relations, we introduce the notion of a schedule. In general, a schedule specifies a graph over the sub-scenarios that details the order, as well as the concurrency of actions. The structure of a scenario is decomposed into a hierarchy of parallel sub-scenarios. Each sub-scenario is composed of a sequence of elementary tasks. The scheduling of elementary tasks is expressed by the concatenation of corresponding state-machines.

J. Allen [13] defined a logic model which describes all possible relative positionings of two temporal intervals along an axis. The Allen logic model includes thirteen elementary relations. The temporal interval over which it is active can be represented by its start instant and end instant. All temporal constraints between sub-scenarios can be expressed in the scenario by using Allen's and instant logic. The scheduling is logically consistent if there is at least one solution to order extremities of segments (circuit checking in graphs [14]).

In the following pseudo-code, we provide a simple example to illustrate a use of scenario scheduling. In order to highlight the scheduling structure, we omit code in the bodies of the sub-scenarios.

```
scenario main () {
  scenario action1 () {...}
  scenario action2 () {...}
  scenario action3 () {...}
  schedule {
    beginning of action1 equals beginning of main;
    beginning of action2 equals beginning of main;
    action2 meets action3;
  }
}
```

In this example three actions are related to each other by how the sub-scenarios interact. Both action1 and action2 start at the same time as their parent. However, action3 cannot start until action2 has finished.

## Putting it Together

The research groups at Irisa and University of Iowa are working together to investigate the design of scenario languages. We have deliberately chosen to take different approaches in order to experiment with a variety of methodologies. Irisa is developing a compiled scenario language with scheduling based on Allen logic. Iowa is developing an interpreted languages that will permit authors to interactively enter directives while a simulation is running. In addition to developing languages, we are exploring a variety of scenario programming styles. By creating a spectrum of languages, we hope to foster experimentation with a diverse approaches to modeling scenario control.

We conclude by outlining the features of a compiled scenario language being developed at Irisa. A complete definition of the language can be found in [5]. Our

objective is to have a strong specification of the scenario, to compile it into a set of HPTS. 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 referent. 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 the scheduling information is stored in a dynamic execution graph. The scheduler (cf figure 3) uses this graph to start or terminate the scenario state machines. To detect situations, triggers and sensing functions are managed by the simulation observation.

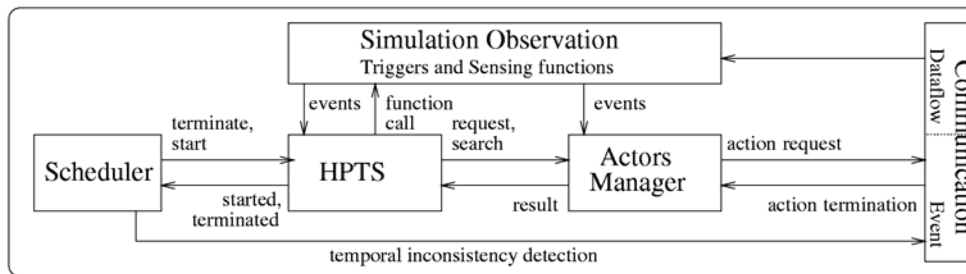



**Figure 3.** Architecture of the Scenario Manager.

Tasks in a scenario can both modify characteristics of dynamic entities or create, suppress and modify their activity or their motivations. As an entity may have different activities running in parallel, it is necessary to organize these activities for each kind of entity by levels of priority and indicate those which can be concurrent. This information will be used by the actor manager, to determine whether it is possible for an entity to perform a specific task, within its current activity, or if it is necessary either to suspend or terminate some current tasks or to wait for their termination. An extension of HPTS is used to describe most of the language instructions. A complete example is shown in appendix A to illustrate the features of this scenario language.

## References

**[1]** S. Donikian and E. Rutten. *Reactivity, concurrency, data-flow and hierarchical preemption for behavioural animation*. In E.H. Blake R.C. Veltkamp, editors, Programming Paradigms in Graphics'95, Eurographics Collection. Springer-Verlag, 1995.

**[2]** R. Sukthankar. *Situational Awareness for Tactical Driving*. PhD Thesis, Robotics Institute, Carnegie Mellon University, January 1997.

**[3]** J. Cremer, J. Kearney and Y. Papelis. *HCSM: A Framework for Behavior and Scenario Control in Virtual Environments*. In ACM Transactions on Modeling and Computer Simulation, 5(3), pages 242-267, July 1995.

**[4]** N. Badler, B. Webber, W. Becket, C. Geib, M. Moore, C. Pelachaud, B. Reich and M. Stone. *Planning and parallel transition networks: Animation's new frontiers*. In S. Y. Shin and T. L. Kunii, editors, Computer Graphics and Applications: Proc. Pacific Graphics '95, pages 101-117, River Edge, NJ. 1995.

**[5]** S. Donikian. *Towards scenario authoring for semi-autonomous entities*. In S.P. Mudur, J.L. Encarnacao, and J. Rossignac, editors, International Conference on Visual Computing (ICVC99), Goa, India, February 1999.

**[6]** J. Cremer, J. Kearney and P. Willemsen. *Directable Behavior Models for Virtual Driving Scenarios*. Transactions of The Society for Computer Simulation 14(2), pages 87-96, 1997.

**[7]** K. Perlin and A. Goldberg. *Improv: a system for scripting interactive actors in virtual worlds*. In SIGGRAPH'96, 1996.

**[8]** B.M. Blumberg and T.A. Galyean. *Multi-level direction of autonomous creatures for real-time virtual environments*. In Siggraph, pages 47-54, Los Angeles, California, U.S.A., August 1995. ACM.

**[9]** O. Alloyer, E. Bonakdarian, J. Cremer, J. Kearney and P. Willemsen. *Embedding Scenarios in Ambient Traffic*. In Proceedings of the Driving Simulation Conference (DSC 97), Lyon, France, 1997. Pages 75-84.

**[10]** P. van Wolffelaar and W. van Winsum. *Traffic Modeling and Driving Simulation - an Integrated Approach*. In Proceedings of the Driving Simulation Conference (DSC 95), Sophia Antipolis, France, 1995. Pages 235-244.

**[11]** J-M. Kelada and J.-M. Piroird. *Traffic generation and scenario control in the SCANeR simulator*. In Proceedings of the Driving Simulation Conference (DSC 95), Sophia Antipolis, France, 1995. Pages 224—234.

**[12]** R. Pausch, T. Burnette, A.C. Capehart, M. Conway, D. Cosgrove, R. DeLine, J. Durbin, R. Gossweiler, S. Koga and J. White. *Alice: A Rapid Prototyping System for 3D Graphics*. In IEEE Computer Graphics and Applications, 15(3), May 1995, pages 8-11.

**[13]** J.F. Allen. *An interval based representation of temporal knowledge*. In Proceedings of the seventh International Joint Conference on Artificial Intelligence, pages 221-226, August 1981.

**[14]** S. Donikian and G. Hegron. *Constraint management in a declarative design method for 3d scene sketch modeling*. In V. Saraswat and P. Van Hentenryck, editors, Principles and Practice of Constraint Programming, pages 427-443. MIT Press, 1995.

## Appendix A.

### A Scenario Case Study

The objective of this scenario is to analyze the behavior of a car driver in an overtaking situation (cf figure 4). The green car (controlled by an interactive driver) is to enter a four-lane divided highway. The traffic densities in the two lanes are regulated by the scenario (ambientGenerator scenario). The right lane is to have low density in order make it easy for the green car to merge into traffic. When the green car passes through the check line CL2 (reducespeed scenario), the car in front of it will be asked to reduce its speed to force the green car to change lanes and overtake it. In the left lane, the flow is generated with a variable headway. The first vehicle of the fleet has the lowest headway, and the headway progressively increases for each follower (inc_headway scenario). We must observe (observer scenario) the configuration when the driver of the green car decides to overtake (distance to the front and back vehicles on the left lane).

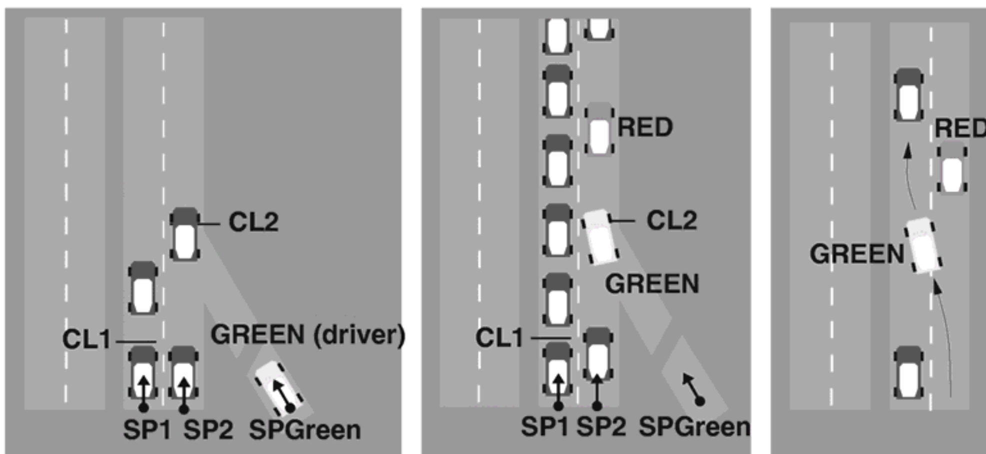



**Figure 4.** Three steps of the Overtaking scenario (beginning, intermediate step, end).

We now present this scenario step by step.

### Init scenario

```
scenario init()
  {
    c = new <drivableCar,"green">;
    send <"init",SPGreen> to c;
    waitfor(c.speed>0);
    send <"init",c> to global;
  }
```

This scenario creates the human driven vehicle and waits for the use of the vehicle by the real driver (speed >0). The "waitfor" instruction is a very simple synchronization instruction. The execution of the scenario is suspended until the condition becomes true.

**Ambient Generator**

```
scenario ambientGenerator( startPoint location,
  float paramFreq,  //vehicles per hours
  string name) //first part of the name for the cars
  {
  int i=0; // cars counter
  float desiredFreq=paramFreq;
  float currentFreq=paramFreq;
  float oldFreq=paramFreq;
  time tEnd=date; //ending time of the current flow variation
  time tStart=date; //start time of the current flow variation
  time deltaT = 0;
  time lastGen=date-(1/currentFreq):0:0:0; // last vehicle creation date
  bool end = FALSE;

  on <"end"> { end = TRUE; } // the end boolean is used by repeat, to terminate
  on <"newFreq",desiredFreq,deltaT> { oldFreq = currentFreq;
       deltaFreq = desiredFreq - oldFreq; tStart=date; tEnd=date+deltaT;}

  repeat {
    if (date>=tEnd) //the desired frequency is already reached
      {currentFreq = desiredFreq;}
    else
      {
      currentFreq = oldFreq + (date.h - tStart.h)*(desiredFreq - oldFreq)/(deltaT);
      }
    if (currentFreq>0) {
      if (date.h > (last.h + 1/currentFreq)) {
        X = new <car,concat(name,str(i))>;
        send <"location",location> to X;
        send <"start"> to X;
        i = i+1;
        lastGen=date;
      }
    }
  } until(end);
}
```

This scenario focuses on traffic generation at the start point. It generates cars at a specified frequency (currentFreq), from an initial location (location). It is possible to modify the frequency, by a "newFreq" event. This event gives information on the new desired frequency (desiredFreq) and on the delay (deltaT) to reach it.

**ReduceSpeed Scenario**

This scenario memorizes the last vehicle which passes through the check line and ends when it is the green one. An instruction "reduce your speed" is then sent to the preceding vehicle to force the driver of the green car to decide to overtake it.

```
scenario reduceSpeed()
{
  X is a car;
  desiredCar is a car;
  bool newcar=FALSE;
  bool end=FALSE;

  on  <"cross",X> from CL2 { newcar = TRUE;}

  eachtime (newcar)
   {
     newcar = FALSE;
     if (X == green)
       {
        desired_speed = green.speed-20;
        send <"reducespeed",desired_speed> to desiredCar;
        end = TRUE;
       }
     else
       {
        desiredCar = X;
       }
   };
  waitfor(end)
  send <"foundRed",desiredCar> to global
}
```

**inc_headway Scenario**

```
scenario inc_headway(float headwaymin)
{
  float hw; //current headway
  bool newcar = FALSE;

  on <"cross",X> from CL1 { newcar = TRUE;}

  hw = headwaymin;
  eachtime(newcar)
  {
    newcar = FALSE;
    hw = hw+0.05;
    send <"headway",hw> to X;
  }
  waitfor(FALSE); // no self-termination
}
```

In this scenario, each time a car passes through the line CL1, the scenario increases the value of the headway and then assigns this headway to the car. As a consequence, a fleet of vehicles with increasing inter-vehicle distances is created.

**greenObserver Scenario**

```
scenario greenObserver(car c) {

  dynamic float alpha = c.orientation;
  dynamic float xt = c.x+cos(alpha+pi/2)*d;
  dynamic float yt = c.y+sin(alpha+pi/2)*d;

  zone is a rectangularTrigger = createTrigger(xt,yt,alpha,10,3);
  // zone is now a trigger, dynamically positioned on the left side of the c car
  // it will be used to detect when the green car will overtake the c car.

  on <"enters",X> from zone { someoneHere =TRUE };
  eachtime(someoneHere)
  {
    if(X==green) { greenHere = TRUE };
  };
  waitfor(greenHere);

  send <"the driver has overtaken ",red.name> to output;
  send <"preceding car ",green.front.name> to output;
  send <"following car ",green.rear.name> to output;
}
```

This scenario is concerned with the actions of the real driver. When the driver decides to initiate the overtaking maneuver, a message is sent to the output channel in order to record data for later playback and analysis. For example, the distance to rear and front vehicles on the left lane could be recorded. The "dynamic" variables are special variables which will be automatically recomputed at each consultation if necessary. In this example, this permits a simple trigger to be attached to a vehicle in motion.

**Global Scenario**

The global scenario will contain or use all the preceding scenarios. The schedule instructions, by using Allen relations [13], define automatically when to start each scenario [5] (cf figure 5). For example, when the *reduceSpeed* scenario terminates, *inc_headway* and *greenObserver* scenarios will be automatically started due to the temporal constraints.

```
scenario global(
  lineTrigger CL1, lineTrigger CL2,
  startPoint SP1,startPoint SP2, startPoint SPGreen )
{
  boolean end=FALSE;
  green is a drivableCar;
  red is a car;

  on <"init",green> from init {}
  on <"foundRed",red> from reduceSpeed {}
  on <"end"> from greenObserver { end = TRUE; }

  scenario ambient1() { use ambientGenerator(SP1, 720, "carleft");}
  scenario ambient2() { use ambientGenerator(SP2, 360, "carright");}
  scenario init(...) {...}
  scenario reduceSpeed (...) {...}
  scenario greenObserver (...) {...}
  scenario inc_headway (...) {...}

  schedule {
    ambient1 equals global;
    ambient2 starts global;
    init starts global;
    init meets reduceSpeed ;
    reduceSpeed meets greenObserver;
    inc_headway equals greenObserver;
    greenOnserver finishes global;
    ambient2 meets greenObserver;
  }
}
```

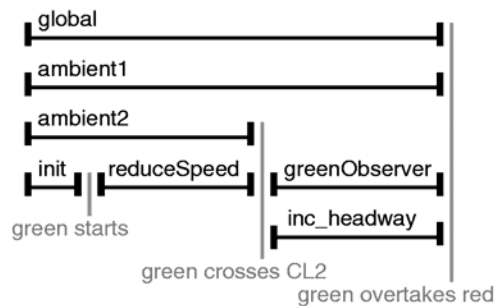



**Figure 5.** Schedule of the example.

## Acknowledgements


This research was supported in part by National Science Foundation grant IRI-9506624, Ford Motor Company and INRIA. We also want to thank Dragos and Ioana Lungeanu for their seminar project in interpreted scripting that demonstrated the interesting possibilities presented by interactive scenario languages.