Intelligent autonomous agents in HLA virtual environments

Samir TORKI, Patrice TORGUET, Cédric SANZA, Jean-Pierrre JESSEL
Virtual Reality & Computer Graphics Research Group, IRIT
Université Paul Sabatier, 118 route de Narbonne
31062 Toulouse Cedex 4, France
Phone. (33) 05 61 55 74 01
Fax. (33) 05 61 55 62 58
E-mail:`{torki,torguet,sanza,jessel}@irit.fr`

Pierre SIRON
Information Modeling and Processing Department, ONERA-CERT
2, av. E. Belin, BP 4025
F-31055 Toulouse Cedex, France
E-mail:`Pierre.Siron@cert.fr`

Keywords :
High Level Architecture, Classifier systems, driving simulation, distributed simulation

ABSTRACT: *Many simulations of virtual environments involve realistically behaving autonomous agents to give users as much interaction capabilities as possible. In this kind of simulations, interoperable architectures such as the High Level Architecture enable us to create complex and lively environments from simple simulations of different kinds of entities. However, making simulations collaborate requires to give existing agents the ability to interact with the newly integrated ones. Such a task generally consists in redefining existing behaviours completely. Hence, making several even interoperable simulations collaborate, implies long and demanding developments while interoperability tries to prevent them. Such drawbacks are mainly due to the fact that most behavioural models are based on finite state machines and expert systems for which designers have to describe exhaustively all agents' behaviour.*

Classifier systems enable designers to describe agents' behaviours through goals ("what to do"*) instead of transitions or set of rules (*"how to do it"*). Then, agents learn how to reach those objectives using evolutionary learning algorithms. Such a modeling is more suited for the development of interoperable simulations. Indeed, adding new simulated agents only requires to add new goals to the existing ones.*

This paper presents the distributed driving simulation we built using interoperable subsimulations. At first, it presents how classifier systems make agents' behaviour easily evolutional in the context of interoperable simulations. Then, it shows how interoperability enables us to share the management of autonomous entities between several computers in a distributed way.

1 Introduction

Most simulations of virtual environments are built as modular sets of subsystems designed to manage every part of their running cycle [1][2]. They generally contain display, audio, haptic, physics model and autonomous entities simulation subsystems.

However, these are commonly designed as complex monolithic applications which fulfil specifications but are difficult to maintain. This is especially the case for autonomous entities simulation systems which simulate the agents users interact with.

This paper mainly focuses on the simulation of such autonomous entities to create traffic in driving simulations. Any traffic simulation has to be well designed as users' behaviours heavily depend on the way this system works. As many driving simulations are created for traffic safety purpose, users require vehicles involved in traffic to act in a realistic way.

In this context, an entity behaves realistically if users react in front of this entity in the same way as if they were in front of a human-controlled one. The worst thing to happen would be that users could exploit some of the model's weaknesses to overcome difficult situations.

Many techniques are used to create realistically behaving agents. The most efficient one consists in making several human-controlled agents participate to a multi-user simulation (e.g. Massively Multiplayer Online Game: MotorCity Online). Such an approach provides the most realistic behaviours possible, but it often requires too many users to simulate dense traffic. This is why most driving simulations involve autonomous entities.

In some cases, autonomous entities are scripted so that they can confront users to very particular situations. However, scripts can be used for only short simulation sequences as they need to define step by step what the entity has to do. Simulations in which users can drive in normal conditions require to define in a generic way how autonomous vehicles have to behave.

Many simulators' and especially the Iowa Driving Simulator's (IDS) autonomous entities were controlled by *finite state machines* (FSMs) [3]. In the IDS, such a modelling led to a very complicated and almost impossible to maintain FSM [4]. To solve those development and maintenance issues, *Hierarchical Concurrent State Machines* (HCSMs) [5] (FSMs in which states can be FSMs) were introduced in the IDS .

Even if FSMs have been widely used, several pedestrian and vehicle traffic simulations are based on boolean and fuzzy logic [6] expert systems. These are sets of *if-then* rules the system triggers according to its perception and to its internal state.

However, all these models require designers to define exhaustively how agents must behave. This leads to very difficult and very demanding to define models and tends to discourage designers from defining various behavioural models for each kind of entities. This generally implies a loss of realism for the simulation.

Another realism issue comes from the need of simulating dense traffic. Indeed, most simulations subsystems are controlled by a single computer [7] which can't handle the simulation of all the vehicles involved in a city. One technique used to solve such a problem in single-user simulations consists in providing as much computing power as possible to simulate vehicles in a small user-centred area [4].

However, such a solution is not suited for multi-user simulations. Indeed, if users were scattered all over the city, a single computer would have to simulate too many vehicles.

This paper mainly presents how we managed to create easier to define autonomous vehicles using *classifier systems* and how we distributed the management of these vehicles between several computers thanks to the *High Level Architecture* (HLA). First, it presents what classifier systems and the HLA are. Then, it shows how we integrated them in our simulation.

2 Classifier systems

Classifier systems [8] are adaptive systems aimed to learn sets of *if-then* rules according to rewards they receive from the environment. Most classifier systems are, like the αCSs [9] we developed, based on Holland's Learning Classifier Systems (LCS). This section presents what LCSs are, how they work and the main differences αCSs introduce.

2.1 Architecture

A classifier system functions as a perception-decision-action loop. At first, it perceives information from the environment with its sensors. Then, a decision unit defines what action the system should trigger according to what it sensed. Finally, the system executes that action using its effectors, which modifies the environment (Figure1).

Most of a classifier system's work is accomplished by its decision unit. It consists of a base of rules called *classifiers*. A classifier is a bit string which represents an *if-then* rule: it is composed of a condition and an action part. A classifier's condition part is made of $0, 1, \#$ bits with $\#$ representing the wildcard ('don't care') symbol ; its action part is a typical $0, 1$ bit string.

Classifier systems' goal consists in making good classifier sets emerge by keeping the most interesting rules alive and replacing the worst ones by potentially better ones. *Emergence* means that classifier systems can generate good rule-bases from a null or randomly initialized set of classifiers.

To differentiate interesting rules from bad ones, every classifier is given a strength which defines how useful it was when the system triggered it.

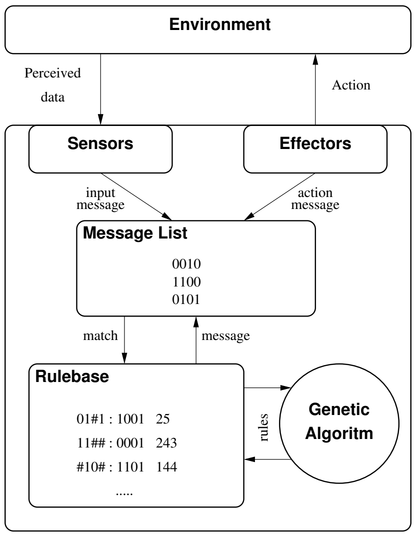

Figure 1: Holland's LCS

2.2 Running cycle

First, the system perceives information from its environment with its sensors. These information are converted into bit string (0,1 bits) messages stored in a message list.

Then the decision process starts; it is mainly divided into three steps:

- performance (or action selection) cycle,
- credit assignment cycle,
- rule discovery cycle.

2.2.1 Performance cycle

The performance cycle consists in choosing the most appropriate rules to trigger according to the messages contained in the message list.

First, all the rules matching at least one message (i.e. every non # bit of its condition part corresponds to the value of the same bit in the message) are selected to participate to a bid. The bid made by every bidding classifier is proportional to its strength. The winners are selected using a *roulette-wheel* selection in which the probability for a classifier to win is proportional to the bid it made.

Those winning classifiers are allowed to post their action part as messages into the message list. These can either be used by other classifiers during the following iteration (*inference cycle*) or sent to the system's effectors to perform the corresponding action and modify the environment.

2.2.2 Credit assignment cycle

The credit assignment cycle consists in updating classifiers' strength according to the consequences their activation implied. Several credit assignment techniques exist, but the most used one is the *Bucket Brigade Algorithm* [8] described in this section.

Once the system chose the classifiers it had to activate, it makes each of them pay the bid it made (this amount is removed from the classifier's strength) to those which helped its selection by posting a message matching its condition part.

When a classifier posts an action to an effector, the system can receive a reward from the environment. This reward can either be positive if that action was useful (i.e. fulfils the agent's objectives) or negative if it produced a bad result.

This reward is added to that classifier's strength. Thus, bad classifiers tend to have a low strength while good ones get stronger. It is worth noticing that a classifier's strength corresponds to a prediction of the income it should receive if it gets triggered by the system.

The credit assignment cycle also makes rules' strength evolve so that never or very rarely selected classifiers become weaker through taxation mechanisms in which:

- Every rule not matching any message pays a life tax : unused or very rarely used rules (those rules correspond to states of the environment the entity never met and 'should' not meet) become weaker.
- Every bidding classifier pays a tax so that very general[1] but weak classifiers become weaker as they pay that tax but get no income.

During the credit assignment cycle, a classifier's strength evolves as:

$$S_i \leftarrow \begin{cases} (1 - T_{life})\cdot S_i, \text{ if it does not bid,} \\ (1 - T_{life} - T_{bid})\cdot S_i, \text{ if it does not win,} \\ (1 - T_{life} - C_{bid})\cdot S_i + R, \text{ if it wins.} \end{cases}$$

where:

T_{life}	= life tax,
T_{bid}	= bidding tax,
C_{bid}	= bidding coefficient,
S_i	= i^{th} classifier's strength,
R	= reward received.

The performance and credit assignment cycles make the system activate the rules which will potentially give it the

[1]A classifier is more general than an other if it has more #s in its condition part.

best income while the others become weaker and tend to be less used. However, this is not enough for an adaptive system as the rules would remain the same during the whole simulation.

2.2.3 Rule discovery cycle

To make their classifier population evolve and new classifiers emerge, classifier systems involve genetic algorithms (GAs) [10]. Such GAs apply three genetic operators on the classifier population:

- *selection* : it consists in selecting the fit classifiers which will remain in the following classifier generation (i.e. the population after the GA is being run). Selection is generally made using a *roulette-wheel* selection in which the probability for a classifier to be selected is proportionnal to its strength.
- *crossover* : parent classifiers are selected and subparts of these classifiers are exchanged (Figure2). The *crossover* operator is aimed to create two children classifiers which combine good features from both parents.

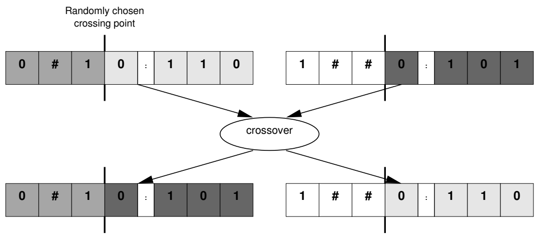

Figure 2: Crossover operator

- *mutation* : it consists in flipping the value of a randomly chosen bit on a randomly selected classifier (Figure3). Mutation is aimed to make new potentially good features emerge.

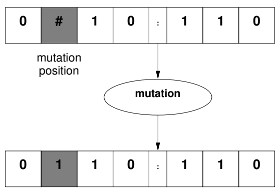

Figure 3: Mutation operator

GAs' goal is to make potentially better classifiers from the existing ones.In many classifier systems (LCSs, ZCSs [11], αCSs...), classifiers' strength is both used as a reward prediction in the action selection cycle and as a fitness function used by the genetic algorithm. During its execution, the GA makes fit rules reproduce and crossover while bad ones are muted to make potentially better classifiers emerge.

A fourth genetic operator, the *covering* operator, was introduced in the context of classifier systems. It is triggered when an environmental message does not match any classifier in the rulebase : a new classifier matching that message (the condition part is the message in which some positions are replaced by #s and the action part is generated randomly) replaces a weak classifier in the rulebase (Figure4).

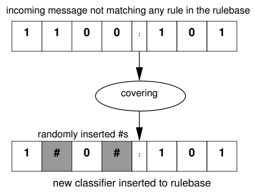
incoming message not matching any rule in the rulebase
1 1 0 0 : 1 0 1
covering
randomly inserted #s
1 # 0 # : 1 0 1
new classifier inserted to rulebase

Figure 4: Covering operator

2.3 αCSs

Our simulation involves vehicles controlled by the αCSs we developed. αCSs are very similar to Holland's LCSs as they globally work in the same way. However, they differ from LCSs as they are designed for real time simulations and provide multiobjective capabilities [9]. Indeed, αCSs don't have any message list in order to avoid variable response times due to inference cycles: at the end of the performance cycle, the winning classifier directly sends its action part to effectors.

αCSs' most important difference with other systems comes from their multiobjective capabilities. Indeed, classifier systems were most often used for quite simple problems such as making an object follow a target. In such problems, defining the agent's fitness is fairly simple as it consists in minimizing an easy to determine function such as the distance between those two objects.

For more complex and real-life problems, defining an agent's objectives using a single fitness function becomes inextricable. Indeed, this function would be either too precise, leading to the emergence of very few different solutions, or not precise enough, requiring more time to make good solutions emerge. To make it easier for designers, αCSs allow them to define an agent's goals through several functions representing different sub-objectives. As in most cases, objectives don't have the same priority (e.g. avoiding collisions is more important than driving fast), αCSs also provide a mechanism of fitness prioritization.

Hence, creating αCS based autonomous agents consists in defining their sensors, their effectors and their objectives using as many fitness functions as needed.

3 The High Level Architecture

The HLA was developed by the Defense Modeling and Simulation Office (DMSO) of the Department of Defense (DoD) to meet the needs of defense-related projects. It became a standard (IEEE 1516) for the creation of interoperable and reusable distributed simulations.

Interoperability makes possible the creation of complex simulations from heterogeneous sub-simulations implemented using different programming languages, or running on heterogeneous systems.

Thus, interoperability brings several advantages for:

- Team collaboration: it allows teams making different technical choices to gather their work into one single application.
- Maintenance: interoperable applications don't require to throw away and to reimplement existing code in case of technical changes as it can be coupled to the new one.
- Performance: most appropriate technical choices can be made for each subpart of the application, making it possible to reduce the number of compromises to be done.

Reusability means that it is possible to reuse some parts of the simulation inside another one; this prevents from having to implement the same program twice for any application running it. This section presents HLA's features and the VIPER platform we built upon it.

3.1 HLA simulation architecture

An HLA simulation is called a *federation*. It is generally made of a set of sub-simulations called *federates* which can be computer controlled (e.g. simulation of autonomous vehicles) or human controlled simulations (e.g. simulation of user driven vehicles). A federation can also contain passive federates such as data collectors or display systems which just gather information from the simulation. Those federates do not participate to the simulation as actors: they do not send information to other federates.

To make the federation work, federates collaborate by exchanging information. HLA imposes federates to communicate in an object-oriented way: federates can share objects which become vehicles for information exchange. To access these objects, federates have to pass through a distributed system: the *RunTime Infrastructure* (*RTI*).

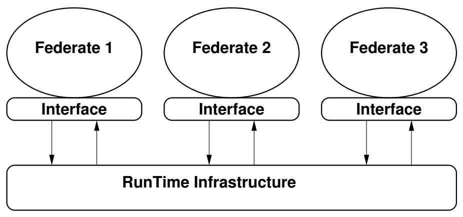

Figure 5: HLA Federation

3.2 VIPER

As a standard, HLA is designed to fit as much as possible with every simulation's needs: it generally provides very generic mechanisms which have to be refined according to the field in which they're used. In the scope of our research, we migrated the VIPER[2] [12] platform upon HLA.

3.2.1 Virtual environment model

VIPER is aimed for the design of every application based on a virtual environment which can be modelled by exchanges (symbolized by stimuli) between entities, in a virtual universe (Figure6).

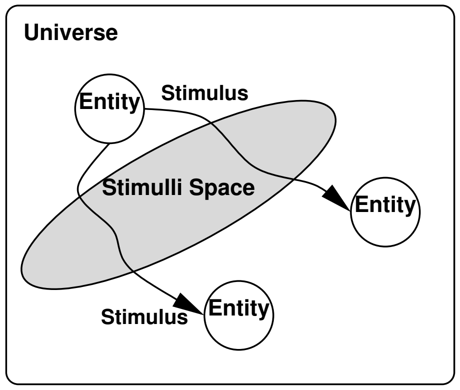

Figure 6: VIPER Simulation

The entity paradigm allows uniform management of virtual worlds scenery, virtual objects and avatars (entities which behave as interfaces between users [13], applications [14][15] or robots and the virtual universe). Entities are autonomous and own a set of attributes and behaviours. They are conceptually grouped into families (a set of entities which own the same attributes and behaviours).

The purpose of this structure is to simplify the definition of distribution schemes. Autonomous entities lead to a perfect encapsulation of the behaviour and state of an entity, and therefore facilitates distribution of entities: such an entity

[2]VIrtuality Programming EnviRonment

can execute its behaviour on any site communicating with other entities through well defined stimuli.

Interactions between entities, modelled by the stimulus paradigm (phenomenon or event perceptible by an entity), cross media, called stimuli spaces, which allow communications between many entities simultaneously. Each stimuli space is in fact a projection of the environment along a specific type of stimulus (3D shape space, sound space…). An entity receives perceptible stimuli (visible shapes, near sounds…) through sensors and acts on its environment through effectors (producing new stimuli). Sensors and effectors also manage interactions with the real world (e.g. a dataglove and a tracker sensor tracking user movements).

Each entity owns a set of behaviour components which modify its internal state (the set of its attributes) and command actions to its effectors. Behaviour components are triggered by sensors (there is a time sensor which allows timed behaviours) or by other components.

3.2.2 Architecture

VIPER's architecture is composed of 3 layers:

- the *virtual environment specification* for the definition of entities' behaviour and the components required for inter-entity communication (sensors and effectors).
- the *virtual environment distribution specification* to define the virtual environment and to specify how entities are distributed.
- the *distributed platform* for data distribution: VIPER can be plugged on existing data distribution systems such as PVM (Parallel Virtual Machine) or HLA.

VIPER provides two programming levels:

- High level programming: the virtual environment specification layer hides the distributed aspect of the application. The developer only has to define the entities, their sensors, effectors, stimuli and behaviour and VIPER manages the distributed aspect of the simulation.
- Low level programming: the virtual environment distribution specification layer enables the programmer to define the way data are distributed within the simulation so that he can optimize it in terms of network load.

Such an architecture also lets the designer choose the most appropriate data distribution system without reimplementing VIPER's upper layers. In its early version, VIPER was developed using PVM, we made it evolve to HLA as it is more suitable for distributed simulations. Indeed, HLA allowed us to define a VIPER entity as a federate (it is also possible to have several entities in one single federate) and stimuli as HLA objects or interactions. VIPER is currently based on CERTI, the ONERA's RTI.

3.3 CERTI

CERTI is an OpenSource RTI developed at ONERA. It provides almost all HLA specified set of services used by simulators to interoperate (such as object management, time management, optimization services, etc.). Using these services helps simulation reuse and interoperability. CERTI's global architecture is presented in Figure7 : it is made of two processes (RTIA and RTIG) and a library (libRTI). The RTIA (RTI Ambassador), which runs on each federate computer, allows concurrent sending/receiving of network messages while the federate application is running. The RTIG (RTIGateway) is a server which relays messages between RTIA (however several RTIG may cooperate in order to distribute the load of messages and to avoid a bottleneck in a system). The RTIG is also able to filter messages in order not to send unwanted messages to any federate and to manage global application security through firewall crossing and network messages encrypting.

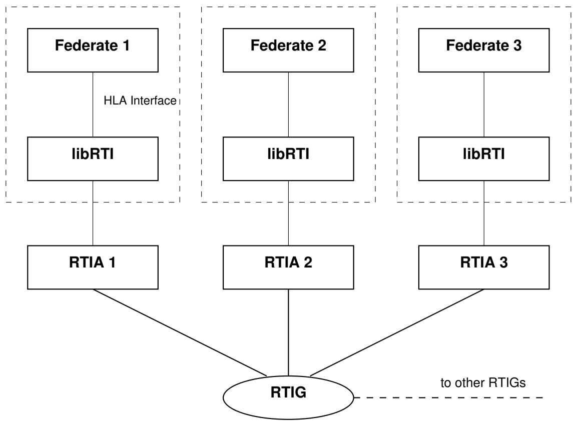

Figure 7: CERTI

4 Classifier systems in an HLA based simulation

The prototype of the driving simulation we are developing is based on a VRML city (Figure8) .

The first part of our work consisted in defining agents' sensors, effectors and behavioural models according to the structure of the street network.

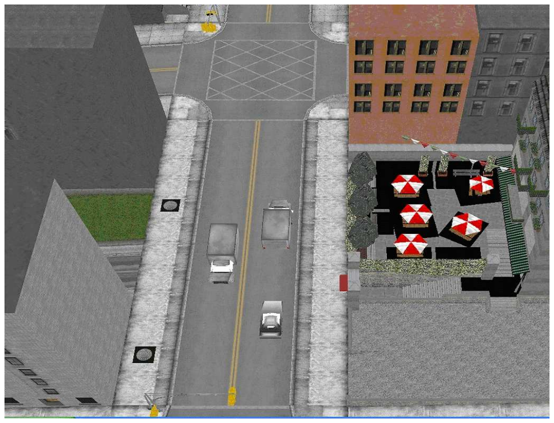

Figure 8: Autonomous vehicles in the city

4.1 Autonomous agents' definition

The simulation involves αCS based autonomous vehicles which have 4 sensors and 4 effectors. αCSs enabled us to define their objectives through 3 quite simple fitness functions (Figure9).

sensors
car ahead very near (VN) car ahead near (N) car ahead quite far (QF) car ahead approaching (AP) on the rightmost lane (R)
effectors
go ahead (GA) accelerate (AC) brake (BR) change lane (SH)
goals
avoid collisions with other vehicles drive as fast as possible remain on the rightmost lane

Figure 9: Agents' components

Vehicles' goals correspond to three fitness functions:

$$fitness1 = \begin{cases} 0\text{, if no collision} \\ -100\text{, if collides.} \end{cases}$$
$$fitness2 = speed - minSpeed.$$
$$fitness3 = \begin{cases} -5\text{, if shifting to leftmost lane} \\ 0\text{, else.} \end{cases}$$

This simple model made several behaviours emerge from a randomly initialized rulebase as shown in Figure10 and Figure11. Classifiers' strength is very important to analyze those behaviours : strong classifiers are more likely

	N	QF	AP	VN	R	action	strength
1	#	0	#	0	#	GA	99.999985
2	0	0	#	0	#	AC	97.334869
	#	0	1	0	#	SH	45.854927
3	#	0	#	0	#	AC	99.999985
4	0	0	#	0	#	AC	99.990540
	#	0	1	0	#	SH	81.616341
	#	0	#	0	#	AC	99.999184
	0	0	1	0	1	SH	99.999985
5	#	0	#	0	#	AC	99.999985
	0	0	#	0	#	AC	99.999985
	#	0	1	0	1	SH	80.798553
6	0	0	#	0	#	AC	99.654404
	0	#	1	#	#	BR	45.834044
7	#	0	#	0	#	AC	99.930191
8	#	0	1	0	#	AC	99.891861
	#	#	1	#	#	SH	30.834044
	0	0	#	0	#	AC	93.321449
	#	0	#	0	#	SH	79.918678
9	#	#	1	#	#	SH	25.097435
	#	#	#	0	#	AC	99.654404
	0	0	#	0	#	AC	99.587669

Figure 10: Significant classifiers (strength > 30)

vehicles	condition	action
1,7	¬vehicle ahead	go ahead
2,4,8,9	¬ vehicle ahead approaching	accelerate shift lane
3	¬ vehicle ahead	accelerate
5	¬ vehicle ahead approaching∧on rightmost lane	accelerate shift lane
6	¬ vehicle ahead ¬ vehicle near∧approaching	accelerate brake

Figure 11: Textual description of obtained behaviours

to send their action part to the effectors than weaker one. Thus, for *vehicle 2*, if a vehicle is approaching, the second classifier will be triggered (approaching bit at 1) while in other conditions the first one will be more likely chosen ($97.334869 \gg 45.854927$).

It is worth noticing that contrary to commonly used static models (finite state machines, expert systems...) for which designers and developers have to define manually all possible behaviours, classifier systems only require to define the objectives once, to get a set of behaviours.

Classifier system based agents can also adapt dynamically their behaviour to the situations they face during the simulation. Thus, if a vehicle hits another one, the classifier liable for that collision receives a very negative payment and tends not to be re-activated.

Classifier systems also make our simulation easily evolvable as adding new kinds of agents only requires to add some elements to the existing ones: contrary to static behavioural models, there is no need to throw previous work away. Indeed, such a task just consists in adding to existing agents:

- sensors to detect the newly integrated ones (e.g. detecting pedestrian and pedestrian crossings),
- effectors to interact with them (e.g. horn),
- goal they have to reach face to face with them (e.g. not to hit pedestrians).

The second part of our work consists in creating a multi-user simulation and in distributing the management of those autonomous agents between several computers using VIPER.

4.2 Distributed features

The way VIPER is designed makes it particularly suited for the simulation of autonomous and human-driven agents coexisting in a virtual environment. Our simulation involves autonomous vehicles driven by classifier systems and user-controlled vehicles. Those vehicles can be modelled as VIPER entities corresponding to HLA federates.

This simulation was developed using VIPER's top level layer (virtual environment specification) to create our entities. This required to define for each type of entities their sensors and effectors emitting and receiving stimuli containing vehicles' speed and position. These information are then used by the behavioural model to define what the entity has to do. Autonomous entities' behaviour is based on αCSs while driven vehicles' behaviour consists in converting users' key strokes into actions.

It is worth noticing that those entities have no sensors. This is due to the fact that information concerning the simulation are brought to users through a display subsystem which corresponds to a passive federate. It gathers information concerning vehicles' positions and orientations and modifies the scenegraph (i.e. a graph in which nodes are 3D transforms and leaves are geometry) to put every vehicle at the right position.

With such a modelling, the reusability HLA provides enabled us to:

- create autonomous entities managed in a distributed way by a set of computers (e.g. clusters) by launching several instances of autonomous vehicle entities on different computers.
- create a multi-user simulation by launching several instances of human-controlled vehicle federates on different computers.

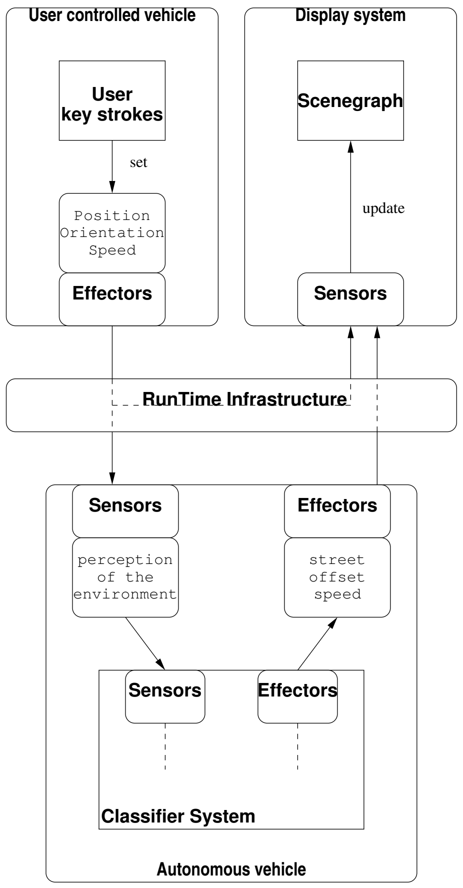

Figure 12: General architecture

5 Conclusion

Combining a VIPER based distributed architecture with evolutionary learning systems such as classifier systems enabled us to create an easily extensible distributed simulation.

Indeed, contrary to simulations involving hard coded behavioural models, αCSs only require extra sensors, effectors and objective functions, which is generally easier than redesigning a new rule base or a new finite state machine. Moreover, using classifier systems allow designers to define *"what the agent has to do"* and not *"how it has to do it"*. Such a modelling also gives the possibility to make different behaviours emerge from a single definition, which leads to more realistic simulations.

The VIPER platform, made the creation of a distributed multi-user simulation possible from quite easy to define components. Indeed, it manages all the distributed part of the work and only requires the definition of entities' sensors, effectors, stimuli and behaviours.

Finally, the use of an interoperable architecture such as the HLA should make our simulation capable of being connected to other HLA simulations based on different technical choices.

We are also currently working on a city abstract model in which we will be able to integrate new features (pedestrian crossings, road signs, traffic lights...) agents will be able to perceive and interact with.

We will also work on persistent simulations and massively multi-entities simulations. We also plan to add database connectivity and to use HLA Data Distribution Management filtering techniques in VIPER's virtual environment distribution specification layer.

References

[1] Woon-Sung Lee. Driving simulation for vehicle driving simulation for vehicle control system development control system development. Technical report, Vehicle Control Lab. School of Mechnical & Automotive Engineering Kookmin University, Mar. 1998.

[2] National Highway Traffic Safety Administration. National Advanced Driving Simulator, Jun. 2002.

[3] Freeman J. The Iowa Driving Simulator: An Implementation and Application Overview. *SAE World Congress, Paper #950174*, 1995.

[4] Cremer J. The Software Architecture of Scenario Control in the Iowa Driving Simulator. *Proceedings of the Fourth Conference on Computer Generated Forces and Behavioral Representation*, pages 373–381, May 1994.

[5] Cremer J.,Kearney J., and Papelis. Y. HCSM: A Framework for Behavior and Scenario Control in Virtual Environments. *ACM Transactions of Modeling and Computer Simulation*, pages 242–267, Jul. 1995.

[6] Al-Shihabi T. Toward More Realistic Behavior Models for Autonomous Vehicles in Driving Simulators. *Transportation Research Record No. 1843*, pages 41–49, 2003.

[7] Johansson M. A Survey of Driving Simulator and their Suitability for Test Volvo Cars. Technical report, Department of Machine and Vehicle System, 2002.

[8] Holland J. Adaptation. In R. Rosen and F. M. Snell, editors, *Progress in Theoretical Biology*. New York: Plenum, 1976.

[9] Sanza C. *Evolution d'entités virtuelles coopératives par systèmes de classifieurs*. PhD thesis, Université Paul Sabatier, Jun. 2001.

[10] Holland J. Adaptation in Natural and Artificial Systems. *The University of Michigan Press*, 1975.

[11] Wilson S. ZCS: A zeroth level classifier system. *Evolutionary Computation*, 2(1):1–18, 1994. http://prediction-dynamics.com/.

[12] Torguet P. *VIPER: Un modèle de calcul réparti pour la gestion d'environnements virtuels*. PhD thesis, Université Paul Sabatier, Feb. 1998.

[13] Mouli R., Duthen Y., Caubet R. In VitrAm (In Vitro Animats, a behavioural simulation model). *2nd IEEE International Workshop RO-MAN'93*, Nov. 1993.

[14] Snowdon D.N.,West A.J. The AVIARY VR-System. A Prototype Implementation. *6^{th} ERCIM workshop*, Jun. 1994.

[15] Carlsson C. and Hagsand O. DIVE - a Platform for Multi-User Virtual Environments. *Computer & Graphics*, 17(6), Nov. 1993.

6 Author Biographies

SAMIR TORKI is graduate from a french computer engineering school. He is now a PhD student in the Computer Graphics and Virtual Reality team at IRIT laboratory. His research interests include virtual environments, distributed systems and behavioural simulation.

PATRICE TORGUET is an Associate Professor in the Paul Sabatier University, in Toulouse, France. His research interests include virtual environments, distributed systems and interactive simulations. He worked in several European Union funded research projects including CAVALCADE (a Cooperative Virtual Prototyping tool) and IMAGE (Interoperability of Aeronautical Simulation Applications).

CÉDRIC SANZA is an Associate Professor in the Paul Sabatier University, in Toulouse, France. His research focuses on autonomous entities including genetic algorithms, classifier systems, anticipation and crowd simulations.

JEAN-PIERRE JESSEL is professor, senior researcher and co-manager of the Computer Graphics and Virtual Reality team at IRIT laboratory. His current research interests include virtual reality (applications to virtual prototyping and collaborative working), distributed collaborative simulations, 3D interaction and character animation.

PIERRE SIRON was graduate from a french engineer school of computer science in 1980, and received his doctorate in 1984. Then he is a Research Engineer at ONERA and he works in parallel and distributed systems and computer security. He is a member of the Design and Validation of Computer Systems research unit.