An executable model for an Intelligent Vehicle Control System

Florin Stoica

***Abstract*—**An abstract state machine (ASM) is a mathematical model of the system's evolving, runtime state. ASMs can be used to faithfully capture the abstract structure and step-wise behaviour of any discrete systems. We present a machine-executable model for an Intelligent Vehicle Control System, implemented in the specification language AsmL. Executable specifications are descriptions of how software components work. The mathematical background for the intelligent control of vehicles is represented by the stochastic automata. A stochastic automaton can perform a finite number of actions in a random environment. When a specific action is performed, the environment responds by producing an environment output that is stochastically related to the action. This response may be favourable or unfavourable. The aim is to design an automata system that can learn the best possible action based on the data received from on-board sensors or from the localization system of highway infrastructure. The reinforcement scheme presented is shown to satisfy all necessary and sufficient conditions for absolute expediency in a stationary environment. Some simulation results are presented, which prove that our algorithm converges to a solution faster than the one given in [7].The proposed model is verified through simulation in SpecExplorer tool from Microsoft Research.

***Keywords*—**Stochastic Learning Automata, Reinforcement Learning, ASMs, systems modeling.

I. Introduction

THE past and present research on vehicle control emphasizes the importance of new methodologies in order to obtain stable longitudinal and lateral control. In this paper, we consider stochastic learning automata as intelligent controller within our model for an Intelligent Vehicle Control System.

Specification and design in the software process are inextricably mixed. Formal specifications are expressed in a mathematical notation with precisely defined vocabulary, syntax and semantics. To create executable specifications, we need an industrial strength language. One such language has been developed at Microsoft Research. It is called AsmL (ASM Language). AsmL is a software specification language based on abstract state machines, a mathematical model of the system's evolving, runtime state. AsmL specifications may be run as a program, for instance, to simulate how a particular system will behave or to check the behavior of an implementation against its specification.

Manuscript received April 13, 2007; Revised received December 28, 2007

Florin Stoica is with the Department of Computer Science, Lucian Blaga University of Sibiu, Faculty of Sciences, Str. Dr. Ion Ratiu 5-7, 550012, Sibiu, ROMANIA (e-mail: florin.stoica@ulbsibiu.ro).

The meaning of these executable specifications comes in the form of an abstract state machine (ASM), a mathematical model of the discrete system's evolving, runtime state.

II. Gurevich Abstract State Machines

Gurevich abstract state machines, formerly known as *evolving algebras* or *ealgebras*, were introduced in [6]. We present here a self-contained introduction to ASMs.

A. States

The notion of ASM state is a variation of the notion of (first-order) structure in mathematical logic.

A *vocabulary* is a collection of *function symbols* and *relation symbols* (or *predicates*) each with a fixed arity. Symbols split into *dynamic* and *static*. Every vocabulary contains (static) logic symbols: nullary function names *true*, *false*, *undef*, the equality symbol, and the standard propositional connectives.

A *state* S of a given vocabulary V is a non-empty set X (the *superuniverse* of S), together with interpretations of the function symbols (the *basic functions* of S) and the predicates (the *basic relations* of S) in V over X.

A function (respectively relation) symbol of arity r is interpreted as a r-ary operation (respectively relation) over X. A nullary function symbol is interpreted as an element of X. The logic symbols are interpreted in the obvious way.

Let f be a relation symbol of arity r. We require that (the interpretation of) f is *true* or *false* for every r-tuple of elements of S. If f is unary, it can be viewed as a *universe*: the set of elements a for which $f(a)$ evaluates to *true*.

Let f be an r-ary basic function and $U_0,\ldots,U_r$ be universes. We say that f has *type* $U_1 \times \ldots \times U_r \rightarrow U_0$ in a given state if $f(x)$ is in the universe U_0 for every $x \in U_1 \times \ldots \times U_r$, and $f(x)$ has the value *undef* otherwise.

B. Updates

A state is viewed as a kind of memory. Dynamic functions are those that can change during computation. A *location* of a state S is a pair $l = (f, (x_1,\ldots, x_j))$ where f is a j-ary dynamic function (or relation) symbol in the vocabulary of S and $(x_1,\ldots, x_j)$ is a j-tuple of elements of S. The element $y = f(x_1,\ldots,x_j)$ is the *content* of that location.

An update of state S is a pair (l, y'), where l is a location $(f, (x_1,\ldots, x_j))$ of S and y' is an element of S; of course y' is *true* or *false* if *f* is a predicate. To fire the update (l, y'), replace the old value $y = f(x_1,\dots, x_j)$ at location l with the new value y' so that $f(x_1,\dots, x_j) = y'$ in the new state.

A set $Upd = \{(l_1, y'_1), ..., (l_n, y'_n)\}$ of updates is *consistent* if the locations are distinct. In other words, *Upd* is *inconsistent* if there are i, j such that $l_i = l_j$ but y'_i is distinct from y'_j. (Example: set-valued variables can be updated partially by inserting and removing individual set members; several such updates are *non-conflicting partial updates* if the set of updates is consistent, i.e. don't both insert and remove the same element).

C. Transition Rules

Expressions are defined inductively. If f is a j-ary function symbol and $e_1,...,e_j$ are expressions then $f(e_1,...,e_j)$ is an expression. (The base of induction is obtained when $j = 0$.) If f is a predicate then the expression is *Boolean*.

An *update rule R* has the form:

$f(e_1,..., e_j) := e_0$

where f is a j-ary dynamic function symbol and each e_i is an expression. (If f is a predicate then e_0 should be a *Boolean* expression). To execute R, fire the update (l, a_0) where $l = (f, (a_1,..., a_j))$ and each a_i is the value of e_i.

A *conditional rule R* has the form:

if e **then** R_1 **else** R_2

where e is a Boolean expression and R_1, R_2 are rules. To execute R, evaluate the guard e. If e is true, then execute R_1; otherwise execute R_2.

A *do-in-parallel rule R* has the form:

do in-parallel

R_1

R_2

where R_1, R_2 are rules. To execute R, execute rules R_1, R_2 simultaneously.

A *do-forall rule R* has the form:

forall $x \in$ *set_expr*

$R_1(x)$

where *set_expr* is a set expression, $R_1(x)$ is a rule and x does not occur freely in the expression *set_expr*. To execute R, execute all subrules $R_1(x)$ with x in *set_expr* at once.

A *choose rule R* has the form:

choose $x \in$ *set_expr*

$R_1(x)$

where $R_1(x)$ is a rule and x does not occur freely in the set expression *set_expr*. To execute R, choose any element x of *set_expr* and execute the subrule $R_1(x)$.

The behaviour of a machine (its run) can always be depicted as a sequence of states linked by state transitions. The run starts form initial state and can be seen as what happens when the control logic is applied to each state in turn:

$S_1 \Rightarrow S_2 \Rightarrow S_3 \Rightarrow \dots$

The machine's control logic behaves like a fix set of transition rules that say how state may evolve.

III. Stochastic Learning Automata

An *automaton* is a machine or control mechanism designed to automatically follow a predetermined sequence of operations or respond to encoded instructions. The term *stochastic* emphasizes the adaptive nature of the automaton we describe here. The automaton described here does not follow predetermined rules, but adapts to changes in its environment. This adaptation is the result of the *learning* process. Learning is defined as any permanent change in behavior as a result of past experience, and a learning system should therefore have the ability to improve its behavior with time, toward a final goal.

The stochastic automaton attempts a solution of the problem without any information on the optimal action (initially, equal probabilities are attached to all the actions). One action is selected at random, the response from the environment is observed, action probabilities are updated based on that response, and the procedure is repeated. A stochastic automaton acting as described to improve its performance is called a *learning automaton*. The algorithm that guarantees the desired learning process is called a *reinforcement scheme* [5].

Mathematically, the environment is defined by a triple $\{\alpha, c, \beta\}$ where $\alpha = \{\alpha_1, \alpha_2, ..., \alpha_r\}$ represents a finite set of actions being the input to the environment, $\beta = \{\beta_1, \beta_2\}$ represents a binary response set, and $c = \{c_1, c_2, ..., c_r\}$ is a set of penalty probabilities, where c_i is the probability that action α_i will result in an unfavourable response. Given that $\beta(n) = 0$ is a favourable outcome and $\beta(n) = 1$ is an unfavourable outcome at time instant $n\ (n = 0, 1, 2, ...)$, the element c_i of c is defined mathematically by:

$$c_i = P(\beta(n) = 1 \mid \alpha(n) = \alpha_i) \quad i = 1, 2, ..., r$$

The response values can be represented in three different models. In the P-model (described above), the response values are either 0 or 1, in the S-model the response values are continuous in the range (0, 1) and in the Q-model the values belong to a finite set of discrete values in the range (0, 1).

The environment can further be split up in two types, stationary and nonstationary. In a stationary environment the penalty probabilities will never change. In a nonstationary environment the penalties will change over time.

In order to describe the reinforcement schemes, is defined $p(n)$, a vector of action probabilities:

$$p_i(n) = P(\alpha(n) = \alpha_i),\ i = \overline{1, r}$$

Updating action probabilities can be represented as follows:

$$p(n+1) = T[p(n), \alpha(n), \beta(n)]$$

where T is a mapping. This formula says the next action probability $p(n+1)$ is updated based on the current probability $p(n)$, the input from the environment and the resulting action. If $p(n+1)$ is a linear function of $p(n)$, the reinforcement scheme is said to be linear; otherwise it is termed nonlinear.

IV. Reinforcement Schemes

A. Performance Evaluation

A learning automaton generates a sequence of actions on the basis of its interaction with the environment. If the automaton is "learning" in the process, its performance must be superior to "intuitive" methods. In the following we will consider the simplest case, the P-model and stationary random environments.

Consider a stationary random environment with penalty probabilities

$\{c_1, c_2, ..., c_r\}$ where $c_i = P(\beta(n)=1 \mid \alpha(n)=\alpha_i)$.

We define a quantity $M(n)$ as the average penalty for a given action probability vector:

$$M(n) = P(\beta(n)=1 \mid p(n)) =$$
$$\sum_{i=1}^{r} P(\beta(n)=1 \mid \alpha(n)=\alpha_i) * P(\alpha(n)=\alpha_i) = \sum_{i=1}^{r} c_i p_i(n)$$

An automaton is absolutely expedient if the expected value of the average penalty at one iteration step is less than it was at the previous step *for all steps*: $M(n+1) < M(n)$ for all n [10].

Absolutely expedient learning schemes are presently the only class of schemes for which necessary and sufficient conditions of design are available. The algorithm we will present in this paper is derived from a nonlinear absolutely expedient reinforcement scheme presented by [7].

B. Absolutely expedient reinforcement schemes

The reinforcement scheme is the basis of the learning process for learning automata. The general solution for absolutely expedient schemes was found by Lakshmivarahan and Thathachar [5].

A learning automaton may send its action to multiple environments at the same time. In that case, the action of the automaton results in a vector of responses from environments (or "teachers"). In a stationary N-teacher P-model environment, if an automaton produced the action α_i and the environment responses are β_i^j $j=1,...,N$ at time instant n, then the vector of action probabilities $p(n)$ is updated as follows [7]:

$$p_i(n+1) = p_i(n) + \left[\frac{1}{N}\sum_{k=1}^{N}\beta_i^k\right] * \sum_{\substack{j=1\\ j\neq i}}^{r}\phi_j(p(n)) - \left[1-\frac{1}{N}\sum_{k=1}^{N}\beta_i^k\right] * \sum_{\substack{j=1\\ j\neq i}}^{r}\psi_j(p(n))$$

$$p_j(n+1) = p_j(n) - \left[\frac{1}{N}\sum_{k=1}^{N}\beta_i^k\right] * \phi_j(p(n)) + \left[1-\frac{1}{N}\sum_{k=1}^{N}\beta_i^k\right] * \psi_j(p(n)) \quad (1)$$

for all $j \neq i$ where the functions ϕ_i and ψ_i satisfy the following conditions:

$$\frac{\phi_1(p(n))}{p_1(n)} = ... = \frac{\phi_r(p(n))}{p_r(n)} = \lambda(p(n)) \quad (2)$$
$$\frac{\psi_1(p(n))}{p_1(n)} = ... = \frac{\psi_r(p(n))}{p_r(n)} = \mu(p(n))$$

$$p_i(n) + \sum_{\substack{j=1\\ j\neq i}}^{r}\phi_j(p(n)) > 0 \quad (3)$$

$$p_i(n) - \sum_{\substack{j=1\\ j\neq i}}^{r}\psi_j(p(n)) < 1 \quad (4)$$

$$p_j(n) + \psi_j(p(n)) > 0 \quad (5)$$

$$p_j(n) - \phi_j(p(n)) < 1 \quad (6)$$

for all $j \in \{1,...,r\} \setminus \{i\}$

The conditions (3)-(6) ensure that $0 < p_k < 1$, $k = \overline{1,r}$.

Theorem If the functions $\lambda(p(n))$ and $\mu(p(n))$ satisfy the following conditions:

$$\lambda(p(n)) \le 0$$
$$\mu(p(n)) \le 0 \quad (7)$$
$$\lambda(p(n)) + \mu(p(n)) < 0$$

then the automaton with the reinforcement scheme in (1) is absolutely expedient in a stationary environment.

The proof of this theorem can be found in [9]

V. A New Nonlinear Reinforcement Scheme

Because the above theorem is also valid for a single-teacher model, we can define a single environment response that is a function f of many teacher outputs.

Thus, we can update the above algorithm as follows:

$$p_i(n+1) = p_i(n) + f*(-\theta*H(n))*[1-p_i(n)] - (1-f)*(-\theta)*[1-p_i(n)]$$

$$p_j(n+1) = p_j(n) - f*(-\theta*H(n))*p_j(n) + (1-f)*(-\theta)*p_j(n) \quad (8)$$

for all $j \neq i$, i.e.:

$$\psi_k(p(n)) = -\theta * p_k(n)$$

$\phi_k(p(n)) = -\theta * H(n) * p_k(n)$

where the learning parameter θ is a real value which satisfy: $0 < \theta < 1$.

The function H is defined as:

$$H(n) = \min\left\{1;\ \max\left\{\min\left\{\frac{p_i(n)}{\theta(1-p_i(n))} - \varepsilon,\ \left(\frac{1-p_j(n)}{\theta * p_j(n)} - \varepsilon\right)_{\substack{j=\overline{1,r}\\ j\neq i}}\right\};0\right\}\right\}$$

Parameter ε is an arbitrarily small positive real number.

Our reinforcement scheme differs from the one given in [7] by the definition of these two functions: H and ϕ_k.

We will show that are satisfied all the conditions of the reinforcement scheme (1).

From (2) we have:

$$\frac{\phi_k(p(n))}{p_k(n)} = \frac{-\theta * H(n) * p_k(n)}{p_k(n)} = -\theta * H(n) = \lambda(p(n))$$

$$\frac{\psi_k(p(n))}{p_k(n)} = \frac{-\theta * p_k(n)}{p_k(n)} = -\theta = \mu(p(n))$$

The rest of the conditions translate to the following:

Condition (3):

$$p_i(n) + \sum_{\substack{j=1\\ j\neq i}}^{r} \phi_j(p(n)) > 0 \Leftrightarrow$$

$$p_i(n) - \theta * H(n) * (1 - p_i(n)) > 0 \Leftrightarrow$$

$$\theta * H(n) * (1 - p_i(n)) < p_i(n) \Leftrightarrow H(n) < \frac{p_i(n)}{\theta * (1 - p_i(n))}$$

This condition is satisfied by the definition of the function $H(n)$.

Condition (4):

$$p_i(n) - \sum_{\substack{j=1\\ j\neq i}}^{r} \psi_j(p(n)) < 1 \Leftrightarrow p_i(n) + \theta * (1 - p_i(n)) < 1$$

But $p_i(n) + \theta * (1 - p_i(n)) < p_i(n) + 1 - p_i(n) = 1$ since $0 < \theta < 1$

Condition (5):

$p_j(n) + \psi_j(p(n)) > 0 \Leftrightarrow p_j(n) - \theta * p_j(n) > 0$ for all $j \in \{1,\ldots,r\} \setminus \{i\}$

But $p_j(n) - \theta * p_j(n) = p_j(n) * (1 - \theta) > 0$ since $0 < \theta < 1$ and $0 < p_j(n) < 1$ for all $j \in \{1,\ldots,r\} \setminus \{i\}$

Condition (6):

$p_j(n) - \phi_j(p(n)) < 1 \Leftrightarrow p_j(n) + \theta * H(n) * p_j(n) < 1$ for all $j \in \{1,\ldots,r\} \setminus \{i\}$

We have:

$p_j(n) + \theta * H(n) * p_j(n) < 1 \Leftrightarrow H(n) < \dfrac{1 - p_j(n)}{\theta * p_j(n)}$ for all $j \in \{1,\ldots,r\} \setminus \{i\}$.

This condition is satisfied by the definition of the function $H(n)$.

With all conditions of the equations (1) satisfied, we conclude that the reinforcement scheme is a candidate for absolute expediency.

Furthermore, the functions λ and μ for our nonlinear scheme satisfy the following:

$$\lambda(p(n)) = -\theta * H(n) \leq 0$$

$$\mu(p(n)) = -\theta \leq 0$$

$$\lambda(p(n)) + \mu(p(n)) = -\theta * (1 + H(n)) < 0$$

because $0 < \theta < 1$ and $0 \leq H(n) \leq 1$

In conclusion, we state the algorithm given in equations (8) is absolutely expedient in a stationary environment**.**

VI. Simulation Results

A. Problem formulation

Reinforcement learning is justified if it is easier to implement the reinforcement function than the desired behavior, or if the behavior generated presents desirable emergent properties (like generalization, robustness, redundancy, adaptability) which cannot be directly built. This last reason is certainly the best motivation for the use of reinforcement learning in autonomous robotics.

To show that our algorithm converges to a solution faster than the one given in [7], let us consider a simple example. Figure 1 illustrates a grid world in which a robot navigates. Shaded cells represent barriers. The current position of the robot is marked by a circle.

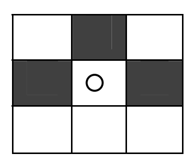

Fig. 1 A grid world for robot navigation

Navigation is done using four actions $\alpha = \{N, S, E, W\}$, the actions denoting the four possible movements along the coordinate directions [8].

B. Comparative results

We compared two reinforcement schemes using these four actions and two different initial conditions. The data shown in Table 1 are the results of two different initial conditions where in first case all probabilities are initially the same and in second case the optimal action initially has a small probability value (0.0005), with only one action receiving reward (i.e., optimal action).

	Average number of steps to reach p_{opt}=0.9999			
	4 actions with $p_i(0)=1/4,\ i=\overline{1,4}$		4 actions with $p_{opt}(0)=0.0005$, $p_{i\neq opt}=0.9995/3$	
θ	Alg. From [7]	New alg.	Alg. From [7]	New alg.
0.01	644.84	633.96	921.20	905.18
0.03	222.45	216.69	332.89	324.59
0.05	136.99	130.41	202.96	198.25
0.1	70.81	63.09	105.12	99.20
0.2	34.73	30.90	53.35	50.93
0.5	11.06	9.75	22.50	19.43

Table 1 Convergence rates for a single optimal action of a 4-action automaton in a stationary environment (200 runs for each parameter set)

Comparing values from corresponding columns, we conclude that our algorithm converges to a solution faster than the one given in [7].

VII. Using Stochastic Learning Automata for Intelligent Vehicle Control

The task of creating intelligent systems that we can rely on is not trivial. In this section, we present a method for intelligent vehicle control, having as theoretical background Stochastic Learning Automata. We visualize the planning layer of an intelligent vehicle as an automaton (or automata group) in a nonstationary environment. We attempt to find a way to make intelligent decisions here, having as objectives conformance with traffic parameters imposed by the highway infrastructure (management system and global control), and improved safety by minimizing crash risk. The aim here is to design an automata system that can learn the best possible action based on the data received from on-board sensors, of from roadside-to-vehicle communications. For our model, we assume that an intelligent vehicle is capable of two sets of lateral and longitudinal actions. Lateral actions are LEFT (shift to left lane), RIGHT (shift to right lane) and LINE_OK (stay in current lane). Longitudinal actions are ACC (accelerate), DEC (decelerate) and SPEED_OK (keep current speed). An autonomous vehicle must be able to "sense" the environment around itself. Therefore, we assume that there are four different sensors modules on board the vehicle (the headway module, two side modules and a speed module), in order to detect the presence of a vehicle traveling in front of the vehicle or in the immediately adjacent lane and to know the current speed of the vehicle. These sensor modules evaluate the information received from the on-board sensors or from the highway infrastructure in the light of the current automata actions, and send a response to the automata.

The response from physical environment is a combination of outputs from the sensor modules. Because an input parameter for the decision blocks is the action chosen by the stochastic automaton, is necessary to use two distinct functions F_1 and F_2 for mapping the outputs of decision blocks in inputs for the two learning automata, namely the longitudinal automaton and respectively the lateral automaton.

After updating the action probability vectors in both learning automata, using the nonlinear reinforcement scheme presented in section 5, the outputs from stochastic automata are transmitted to the regulation layer. The regulation layer handles the actions received from the two automata in a distinct manner, using for each of them a regulation buffer. If an action received was rewarded, it will be introduced in the regulation buffer of the corresponding automaton, else in buffer will be introduced a certain value which denotes a penalized action by the physical environment. The regulation layer does not carry out the action chosen immediately; instead, it carries out an action only if it is recommended k times consecutively by the automaton, where k is the length of the regulation buffer. After an action is executed, the action probability vector is initialized to $\frac{1}{r}$, where r is the number of actions. When an action is executed, regulation buffer is initialized also.

VIII. Sensor Modules

The four teacher modules mentioned above are decision blocks that calculate the response (reward/penalty), based on the last chosen action of automaton. Table 2 describes the output of decision blocks for side sensors.

As seen in Table 2, a penalty response is received from the left sensor module when the action is LEFT and there is a vehicle in the left or the vehicle is already traveling on the leftmost lane. There is a similar situation for the right sensor module.

The Headway (Frontal) Module is defined as shown in Table 3. If there is a vehicle at a close distance (< admissible distance), a penalty response is sent to the automaton for actions LINE_OK, SPEED_OK and ACC. All other actions (LEFT, RIGHT, DEC) are encouraged, because they may serve to avoid a collision.

The Speed Module compares the actual speed with the desired speed, and based on the action chosen send a feedback to the longitudinal automaton.

The reward response indicated by 0* (from the Headway Sensor Module) is different than the normal reward response, indicated by 0: this reward response has a higher priority and must override a possible penalty from other modules.

	Left/Right Sensor Module	
Actions	Vehicle in sensor range or no adjacent lane	No vehicle in sensor range and adjacent lane exists
LINE_OK	0/0	0/0
LEFT	1/0	0/0
RIGHT	0/1	0/0

Table 2 Outputs from the Left/Right Sensor Module

	Headway Sensor Module

Actions	Vehicle in range (at a close frontal distance)	No vehicle in range
LINE_OK	1	0
LEFT	0	0
RIGHT	0	0
SPEED_OK	1	0
ACC	1	0
DEC	0*	0

Table 3 Outputs from the Headway Module

	Speed Sensor Module		
Actions	Speed: too slow	Acceptable speed	Speed: too fast
SPEED_OK	1	0	1
ACC	0	0	1
DEC	1	0	0

Table 4 Outputs from the Speed Module

IX. An AsmL model for Intelligent Vehicle Control

In this section is described an AsmL program-model for Intelligent Vehicle Control. In Figure 2 is showed the class diagram of our AsmL model.

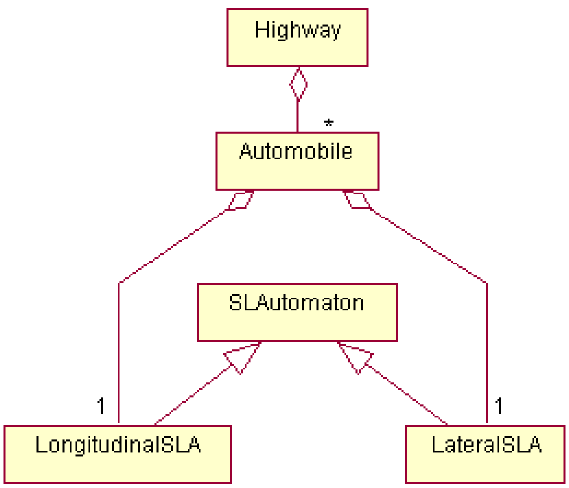

Fig. 2 The class diagram of the AsmL model

From this model are given detailed descriptions of the sensor modules and their outputs, definitions of functions for mapping the outputs of decision blocks in inputs for the two learning automata, namely the longitudinal automaton and respectively the lateral automaton, the learning process which are using the reinforcement scheme from section 5 and the selection of the action to be executed, according to the policy imposed through the regulation buffers.

For the longitudinal automaton, the environment response has the following form:

```
function reward(action as Integer) as Double
  var combine as Integer
  step
     combine :=  (max x | x in
     {speedModule(action),frontModule(action)})
  step
     if (combine = 2) combine := 0
  step
     return combine as Double
```

The speed module and the headway (frontal) module are specified as follows:

```
function frontModule(action as Integer) as Integer
        match action
          SPEED_OK:
             return auto.frontSensor()
          ACC:
             return auto.frontSensor()
          DEC:
             if (auto.frontSensor()=1)
                return 2
             else
                return 0
          _:
             return 0
function speedModule(action as Integer) as Integer
         match action
           SPEED_OK:
              if (auto.speedSensor() <> 0)
                 return 1
              else
                 return 0
           DEC:
              if (auto.speedSensor() = -1)
                 return 1
              else
                 return 0
           ACC:
              if (auto.speedSensor() = 1)
                 return 1
              else
                 return 0
           _:
              return 0
```

The *frontSensor()* method of the class *Automobile* are using the highway infrastructure in order to obtain the current position of headway vehicle, and return 1 (penalty) if there is such a vehicle at a lower distance than the minimum admissible distance, respectively 0 (reward) in other case.

```
function frontSensor() as Integer
   if (h.inFront(me))
      return 1
   else
      return 0
```

where *h* is the *Highway* object which are supervising the traffic. The *inFront()* method of class *Highway* must detect if there is an vehicle in front of the driven vehicle, at a distance lower than the minimum admissible distance:

```
function inFront(auto as Automobile) as Boolean
        if exists a in cars where
           (a.getLane() = auto.getLane())
           and (a.getX() - auto.getX() < front_dist)
           and (a.getX() - auto.getX() > 0.0)
```

```
        return true
    else
        return false
```

where *cars* represents the set of all vehicles which are running on the highway.

The learning process of the longitudinal automaton is described by the following method:

```
procedure learning()
    var i as Integer = 0
    var f as Double = 0.0
    var h as Double = 0.0
    var doIt as Boolean = false
    // choose an action
    step
      i := getAction()
    // compute environment response
    step
      f := reward(i)
    step for k = 1 to HISTORY-1
      regulation_layer(k-1):=regulation_layer(k)
    step
      if (f = 0)
        regulation_layer(HISTORY-1) := i
      else
        // ignore the action
        regulation_layer(HISTORY-1) := -1
      doIt:=true
    step for k = 0 to HISTORY - 1
      if (regulation_layer(k)<>i)
        doIt:=false
    step
      if (doIt)
        init()
        match i
          ACC:
            auto.setCurrentSpeed(
            auto.getCurrentSpeed()+delta)
          DEC:
            if (auto.getCurrentSpeed() > delta)
                auto.setCurrentSpeed(
                auto.getCurrentSpeed()-delta)
    step
      h := H(i)
    // update action probabilities
    // according to the our reinforcement scheme
    step
      p(i):=p(i)+f*(-t*h)*(1.0-p(i))-
              (1.0-f)*(-t)*(1.0-p(i))
    step for j=0 to ACTIONS-1
      if (j <> i)
        p(j):=p(j)-f*(-t*h)*p(j)+(1.0-f)*(-t)*p(j)
```

The function *H* of the nonlinear reinforcement scheme is specified as follows:

```
function H(i as Integer) as Double
  var h as Double = 0.0
  step
        h := p(i)/(t*(1.0-p(i)))-eps
  step for j=0 to ACTIONS-1
        if (j <> i)
          h := (min x | x in {h, (1.0-p(j))/(t*p(j))-eps })
  step
        h := (max x | x in {h, 0.0})
  step
        h := (min x | x in {h, 1.0})
  step
       return h
```

X. Simulation using scenarios

Spec Explorer is a software development tool for model-based specification and testing. Spec Explorer can help software development teams to detect errors in the design, specification and implementation of their systems [14].

The core idea behind Spec Explorer is to encode a system's intended behavior (its specification) in machine-executable form (as an AsmL "model program" [13]) which capture the relevant states of the system and show the constraints that a correct implementation must follow. The goal is to specify from a chosen viewpoint what the system must do, what it may do and what it must not do.

Also, Spec Explorer is used to explore the possible runs of the specification-program to validate designs, in other words, to see that no incorrect scenarios arise as a consequence of the design and that required scenarios are possible.

Discrepancies between actual and expected results are called *conformance failures* and may indicate any of the following: implementation bug, modeling error, specification error or design error**.**

The output of the exploration feature consists of possible runs of the model program that it discovers. Spec Explorer represents this data as a finite-state machine (FSM)**.** The nodes of the FSM are the states of the model program before and after the invocation of a top-level method (an *action*). *Actions* are the top-level methods that cause transition of the system from one state to another. *Scenario actions* represent sequences of subactions given programmatically. In the typical case, we use a scenario action to drive the system into a desired initial state.

In our model, there is a *scenario action* Main():

```
[Action(Kind=ActionAttributeKind.Scenario)]
Main()
  require init = false
  step
     h := new Highway()
  step
     a1 := new Automobile("auto1", 0, 95, 100, h)
     a2 := new Automobile("auto2", 0, 110, 80, h)
     // …
step
     // partial update
     h.addCar(a1)
```

```
    h.addCar(a2)
    // …
  step
    init := true
```

The object *Highway* represents the highway infrastructure, namely the localization system of the vehicles. After objects instantiations, the AsmL model is simulated in SpecExplorer through the execution of the *Run()* action, within all vehicles included in the scenario are driving in parallel, in an intelligent fashion.

```
[Action]
procedure Run()
  require init = true
  step forall a in h.cars
      a.Driving()
```

Using SpecExplorer, we can detect error states (having the red color in the FSM generated by the exploration algorithm), and then, using the information provided by the SpecExplorer related to the error discovered, we can correct our model or design.

By example, an error can occur from a postcondition violation (in method *setLane()*).

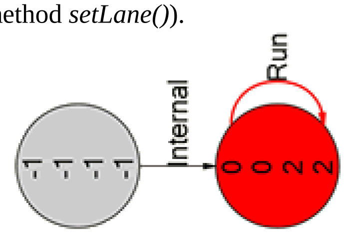

Fig. 3 An error state in SpecExplorer

```
procedure setLane(lane as Integer)
  step
    me.lane := lane
    me.y := lane * Highway.laneWidth +
        Highway.laneWidth/2
  step
    require not (exists a in h.cars
    where lane = a.getLane()
    and (a.getX()-me.getX()) < h.front_dist and
    (a.getX()-me.getX()) > -h.front_dist)
```

XI. Conclusion

Reinforcement learning has attracted rapidly increasing interest in the machine learning and artificial intelligence communities. Its promise is beguiling - a way of programming agents by reward and punishment without needing to specify how the task (i.e., behavior) is to be achieved. Reinforcement learning allows, at least in principle, to bypass the problems of building an explicit model of the behavior to be synthesized and its counterpart, a meaningful learning base (supervised learning).

The reinforcement scheme presented in this paper satisfies all necessary and sufficient conditions for absolute expediency in a stationary environment. Used within a simulator of an Intelligent Vehicle Control System, this new reinforcement scheme has proved its efficiency.

An AsmL model program typically does much less than the implementation; it does just enough to capture the relevant states of the system and show the constraints that a correct implementation must follow. The goal is to specify from a chosen viewpoint what the system must do, what it may do and what it must not do.

Exploration of the the possible runs of the specification-program using SpecExplorer was used to validate the model, in other words, to see that no incorrect scenarios arise as a consequence of the design and that required scenarios are possible.

References

[1] A. Barto, S. Mahadevan, Recent advances in hierarchical reinforcement learning, *Discrete-Event Systems journal, Special issue on Reinforcement Learning*, 2003.

[2] R. Sutton, A. Barto, *Reinforcement learning: An introduction*, MIT-press, Cambridge, MA, 1998.

[3] O. Buffet, A. Dutech, and F. Charpillet. Incremental reinforcement learning for designing multi-agent systems, In J. P. Müller, E. Andre, S. Sen, and C. Frasson, editors, *Proceedings of the Fifth International Conference onAutonomous Agents*, pp. 31–32,Montreal, Canada, 2001. ACM Press.

[4] J. Moody, Y. Liu, M. Saffell, and K. Youn. Stochastic direct reinforcement: Application to simple games with recurrence, In *Proceedings of Artificial Multiagent Learning. Papers from the 2004 AAAI Fall Symposium,Technical Report FS-04-02*, 2004.

[5] S. Lakshmivarahan, M.A.L. Thathachar, Absolutely Expedient Learning Algorithms for Stochastic Automata, *IEEE Transactions on Systems, Man and Cybernetics*, vol. SMC-6, pp. 281-286, 1973

[6] Gurevich Y., Evolving Algebras 1993: Lipari Guide, *Specification and Validation Methods*, ed. E. Börger, Oxford University Press, 1995, pg. 9-36.

[7] Cem Ünsal, Pushkin Kachroo, John S. Bay, Multiple Stochastic Learning Automata for Vehicle Path Control in an Automated Highway System, *IEEE Transactions on Systems, Man, and Cybernetics -part A: systems and humans*, vol. 29, no. 1, january 1999

[8] Florin Stoica, Emil M. Popa, An Absolutely Expedient Learning Algorithm for Stochastic Automata, *WSEAS Transactions on Computers*, Issue 2, Volume 6, February 2007, ISSN 1109-2750, pp. 229-235

[9] N. Baba, New Topics in Learning Automata: Theory and Applications, *Lecture Notes in Control and Information Sciences* Berlin, Germany: Springer-Verlag, 1984.

[10] K. S. Narendra, M. A. L. Thathachar, *Learning Automata: an introduction*, Prentice-Hall, 1989.

[11] C. Rivero, Characterization of the absolutely expedient learning algorithms for stochastic automata in a non-discrete space of actions, *ESANN'2003 proceedings - European Symposium on Artificial Neural Networks Bruges (Belgium)*, 23-25 April 2003, d-side publi., ISBN 2-930307-03-X, pp. 307-312

[12] K.P. Topon, I. Hitoshi, Reinforcement Learning Estimation of Distribution Algorithm, *Proceedings of the Genetic and Evolutionary Computation Conference 2003 (GECCO2003)*

[13] Foundations of Software Engineering - Microsoft Research, *AsmL: The Abstract State Machine Language*, Reference manual, revised 2006, Available: http://research.microsoft.com/fse/asml/

[14] Foundations of Software Engineering - Microsoft Research, *Spec Explorer Reference*, Reference manual, Available: http://research.microsoft.com/specexplorer/.