The WCCI 2008 Simulated Car Racing Competition

Daniele Loiacono, Julian Togelius, Pier Luca Lanzi, Leonard Kinnaird-Heether,
Simon M. Lucas, Matt Simmerson, Diego Perez, Robert G. Reynolds, Yago Saez

***Abstract*— This paper describes the simulated car racing competition held in association with the IEEE WCCI 2008 conference. The organization of the competition is described, along with the rules, the software used, and the submitted car racing controllers. The results of the competition are presented, followed by a discussion about what can be learned from this competition, both about learning controllers with evolutionary methods and about competition organization. The paper is co-authored by the organizers and participants of the competition.**

I. Introduction

In conjunction with the IEEE World Congress on Computational Intelligence (WCCI) 2008 we organized a simulated car racing competition, where the goal was to learn, or otherwise develop, a controller for a car in the TORCS open-source racing game. This paper describes the motivations for holding the competition, the software developed, the rules applied, the entries submitted and the competition results.

There are several reasons for holding competitions as part of the regular events organized by the computational intelligence community. A main motivation is to improve benchmarking of learning algorithms. Benchmarking is frequently done using very simple testbed problems, that might or might not capture the complexity of real-world problems. When researchers report results on more complex problems, the technical complexities of accessing, running and interfacing to the benchmarking software might prevent independent validation of and comparison with the published results. Here, competitions have the role of providing software, interfaces and scoring procedures to fairly and independently evaluate competing algorithms and development methodologies.

Another strong incentive for running these competitions is that it motivates researchers. Existing algorithms get applied to new areas, and the effort needed to participate in a competition is (or at least, should be) less than it takes to come up with the results for and write a completely new paper. Competitions might even bring new researchers into the computational intelligence fields, both academics and non-academics. One of the reasons for this, especially for games-related competitions, is that it simply looks cool.

Daniele Loiacono and Pier Luca Lanzi are with the Department of Computer Science, Politecnico Di Milano, Italy. Julian Togelius is with IDSIA, Galleria 2, 6298 Manno-Lugano, Switzerland. Leonard Kinnaird-Heether and Robert G. Reynolds are with Wayne State University, USA. Simon M. Lucas is with the University of Essex, Colchester CO4 3SQ, United Kingdom. Matt Simmerson is an independent researcher residing in New Zealand. Diego Perez and Yago Saez are with the Universidad Carlos III de Madrid, Madrid, Spain. emails: loiacono@elet.polimi.it, julian@idsia.ch, lanzi@elet.polimi.it, lkinnaird@wayne.edu, sml@essex.ac.uk, mattsimmerson@gmail.com, akilascartas@gmail.com, reynolds@cs.wayne.edu, yago.saez@uc3m.es

In 2007, simulated car racing competitions were organized as part of the IEEE Congress on Evolutionary Computation (CEC) and Computational Intelligence and Games Symposium (CIG). These competitions used a graphically and mechanically simpler game. Partly because of the simplicity of the software, these competitions enjoyed a good degree of participation. The organization, submitted entries and results of these competitions were recently published in [1].

This competition was similar to the 2007 competitions in its overall idea and execution, but there are several important differences. The main difference is that we decided to build the event around a much more complex car racing game, the open-source racing game TORCS. While the main reason for using this game was that the more complex car simulations (especially the possibility for having many cars at the track at the same time with believable collision handling) poses new challenges for the controllers to overcome, other reasons included the possibility of convincing e.g. the game industry that computational intelligence algorithms can handle "real" games and not only academically conceived benchmarks, and the increased attention that a more sophisticated graphical depiction of the competition generates (see figure 1).

Fig. 1. The TORCS game.

II. Competition setup

In what follows, we will describe the TORCS game, the modifications we did to allow communication with the game clients, the clients and their interfaces, example controllers and training software, and the rules of the competition.

A. The TORCS game

The Open Racing Car Simulator (TORCS) [2] is a state-of-the-art open source car racing simulator. It falls somewhere between being an advanced simulator, like recent

978-1-4244-2974-5/08/$25.00 ©2008 IEEE

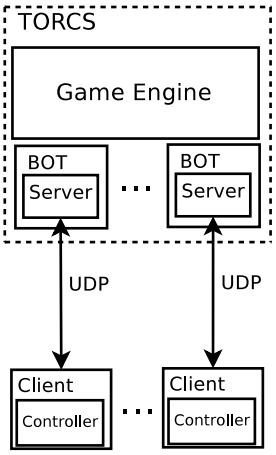

Fig. 2. The architecture of the API developed for the competition.

commercial car racing games, and a fully customizable environment, like the ones typically used by computational intelligence researchers for benchmark purposes. From one hand, TORCS features a sophisticated physics engine, that takes into account many aspects of the racing car (e.g. collisions, traction, aerodynamics, fuel consumption, etc.) as well as a 3D graphics engine for the visualization of the races. On the other hand, TORCS was not been only conceived as a free alternative to commercial racing games, but it was specifically devised to make it as easy as possible to develop your own car controller. In fact, the controllers are implemented as separated software modules so that it is easy to develop a new controller and to plug it into the game. In addition, TORCS does not only provide a complete and complex game environment, but it also provides a lot of game content (i.e., several tracks, car models and controllers, etc.), resulting in a countless number of possible game situations.

B. Competition-specific software modifications

In order to develop their car controllers, the competitors have been provided with a specific software interface developed on a client/server basis. Figure 2 shows the architecture of such interface. The controllers run as external programs and communicate with a customized version of TORCS through UDP connections. Each controller perceives the racing environment through a number of sensor readings which would reflect both the surrounding environment (the tracks and the opponents) and the current game state and they could invoke basic driving commands to control the car. The complete list of sensors is reported in Table I and includes rangefinders to perceive the nearby track limits as well as the distance of nearby opponents, the current speed, the engines RPM, the current gear, the fuel level, etc (we refer the interested reader to the software manual of the competition [3] for additional details). Table II reports all the driving commands: besides the rather typical driving commands (i.e., the steering wheel, the gas pedal, the brake pedal, and the gear change) a meta-command is available to reset the state of the race from the client-side. Controllers had to act quickly on the basis of the most recent sensory information to properly control the car; a slow controller would be inherently penalized since it would be working on lagged information. To make it easy to enter the competition, a client with simple APIs as well as a sample programmed controller were provided for C++ and Java languages and for Windows, Mac, and Linux operative systems.

TABLE I

DESCRIPTION OF AVAILABLE SENSORS.

Name	Description
angle	Angle between the car direction and the direction of the track axis.
curLapTime	Time elapsed during current lap.
damage	Current damage of the car (the higher is the value the higher is the damage).
distFromStartLine	Distance of the car from the start line along the track line.
distRaced	Distance covered by the car from the beginning of the race
fuel	Current fuel level.
gear	Current gear: -1 is reverse, 0 is neutral and the gear from 1 to 6.
lastLapTime	Time to complete the last lap
opponents	Vector of 18 sensors that detects the opponent distance in meters (range is [0,100]) within a specific 10 degrees sector: each sensor covers 10 degrees, from $-\pi/2$ to $+\pi/2$ in front of the car.
racePos	Position in the race with to respect to other cars.
rpm	Rumber of rotation per minute of the car engine.
speedX	Speed of the car along the longitudinal axis of the car.
speedY	Speed of the car along the transverse axis of the car.
track	Vector of 19 range finder sensors: each sensors represents the distance between the track edge and the car. Sensors are oriented every 10 degrees from $-\pi/2$ and $+\pi/2$ in front of the car. Distance are in meters within a range of 100 meters. When the car is outside of the track (i.e., pos is less than -1 or greater than 1), these values are not reliable!
trackPos	Distance between the car and the track axis. The value is normalized w.r.t to the track width: it is 0 when car is on the axis, -1 when the car is on the left edge of the track and +1 when it is on the right edge of the car. Values greater than 1 or smaller than -1 means that the car is outside of the track.
wheelSpinVel	Vector of 4 sensors representing the rotation speed of the wheels.

C. Rules and organization

In brief, the goal of the competition was to learn, or otherwise design, a controller which could be able to race for a certain number of laps on a set of three unknown tracks, alone or against other controllers. Altough the aim of the competition was to encourage the application of computational intelligence techniques to the design of car controllers in TORCS, also human programmed controllers were accepted as entries. Thus, the competitors were free either to use the APIs and the sample controller provided to develop their controllers or they could develop their own solution from scratch. The only constraint was to submit a final controller that follows the same communication protocol defined in the client provided. To evaluate the performance of the submitted controllers we measured the distance raced

TABLE II

DESCRIPTION OF AVAILABLE EFFECTORS.

Name	Description
accel	Virtual gas pedal (0 means no gas, 1 full gas).
brake	Virtual brake pedal (0 means no brake, 1 full brake).
gear	Gear value.
steering	Steering value: -1 and +1 means respectively full left and right, that corresponds to an angle of 0.785398 rad.
meta	This is meta-control command: 0 do nothing, 1 ask competition server to restart the race.

by each controller within 10000 game tics, corresponding to 200s of simulated time. Then, a tournament of competitive races among the best performing controllers was set to elect the winner of the competition and to validate that controllers perform well also in the presence of other cars. The evaluation of the submitted controllers was run by the organizers just before the 2008 IEEE World Congress on Computational Intelligence (more details on the evaluation process are provided in a later section). The set of TORCS tracks used for the final evaluation was not known in advance by the competitors to promote the submission of controllers with a reliable performance on a large range of tracks in spite of controllers tuned on a small set of tracks.

III. SUBMITTED CONTROLLERS

A. Leonard Kinnaird-Heether and Robert Reynolds

The controller was designed from scratch using Java. The initial phase of the development of the controller focused on creating a simple, untrained, system that could navigate the track without getting stuck or crashing too often. This initial stage provided a basic framework for the controller that governed three basic behaviors; acceleration, steering, shifting, and error correction.

Acceleration is based on setting target speeds. These target speeds are governed by a set of speed limit variables that will be discussed in depth later in this section. Simply put, the controller will tell the car to accelerate if its current speed is less than the target speed, not accelerate if the current is less than or equal to 10 mph over the target speed, and brake if the current speed is greater than 10 mph over the target speed. These instructions are in the form of rules. The Cultural Algorithm is used in the second phase to learn these rules for given tracks.

Steering is performed by finding the furthest distance to the edge of the track by using the distance readings gained from communication with the server. The angle that corresponds to the furthest distance is then used as the current steering angle (normalized to fit with a range of [-1, 1].) This calculation is made in every game turn, thus reducing the chance of oversteering. The controller will modify the turning angle as the car passes through a turn,. This is due to the fact that the angle corresponding to the furthest distance to the edge of the track will tend to move toward 0 degrees(straight) as the turn progresses. In this way, the controller finds an adequate, if not the best, method of navigating a turn. In addition to this, there is also a function that attempts to return the car to the center of the track when it is driving in a straight line. This function helped smooth out some problems that occurred during turning, but caused a small problem later in the training of the controller.

Shifting is done based on readings retrieved from the server for the current Revolutions Per Minute(RPM) reading for the car. When the RPM reading reaches a certain point, the controller will send a signal to either shift up or down. In the implementation the controller was required to wait a short period of time after shifting before deciding whether or not to shift again in order to avoid oscillation.

Error correction in the controller falls into three categories; whether or not the car is stuck, whether or not the car is off the track, and whether or not the car is going in the wrong direction. If the car does not move farther than a specified distance over one game turn, the controller increments a counter to indicate that the car may be stuck. When this counter reaches a certain level, the controller directs the car to drive in reverse for a number of game turns. By doing this, the car should have backed up far enough to be able to turn out of the situation that led it to get stuck. If the car is detected to be beyond the boundary of the track, the controller will direct it to slow down, and turn in the direction that corresponds to the opposite side of the track. If the controller detects that the car is traveling at an angle that is greater than +/-90 degrees from the track normal (0 degrees) the controller indicates that the car is traveling in the wrong direction. The controller then directs the car to reduce speed and turn in the direction that will provide it the fastest way of returning to the correct direction. The reduction in speed is necessary to ensure that the car has enough room to successfully turn around. If multiple error states are detected at the same time, the controller handles them hierarchically. The stuck state commands the highest priority, followed by the wrong direction state, and then the off the track state.

1) Learning phase: In our view race car driving is a social activity, and the drivers behavior, in terms of using the kinematic functions discussed above, needed to reflect that activity. In particular, we viewed the race cars as forming a pack or swarm. Thus, the rules for acceleration that we used need to learn within a social context. This was an ideal context for the use of Cultural Algorithms [4]. The Java-based Cultural Algorithm Toolkit (CAT) that uses an agent-based simulation environment (REPAST) was the framework in which the learning task was expressed [5]. The goal of the learning was to learn driving rules that are consistent with the two basic kinematic activities of agents moving in a swarm: orientation relative to the swarm's direction, and spacing between agents so that collisions are less likely. These were implicit factors in the fitness function used to train the driver.

Once the initial controller was completed, the system had to be tailored so that it could interface with the cultural algorithms toolkit. Though several methods were considered,

the decision was made to train the controller to optimize the speed during turns. To accomplish this, the concept of a speed limit, as mentioned previously in this paper, was introduced. Essentially, this concept provides that the controller will direct the car not to travel faster than a certain speed, when navigating a turn with a specific angle, by setting its target speed equal to the current speed limit. To accomplish this, ten variables were introduced to the controller to hold the speed limit values. These variables correspond to straight ahead, turning 10 degrees, 20 degrees, and so on, up to 90 degrees.

CAT requires that a testing program specify four things, the dimensions of the problem, the range of values that are acceptable for each dimension, what type of numeric value each dimension uses, and a fitness function. In this case, the testing program defines the dimension number as 10, corresponding to the number of speed limit variables. The range of each dimension was determined to be [0,300] because 300 mph is roughly the fastest that the specified car can safely travel on any track. Since the controller communicates speed to TORCS as a floating point number, the testing program was set to treat all values generated for each dimension as real numbers, rather than as integers. In a typical application of the CAT, the fitness function is an equation that can be quickly calculated. In this case, it was concluded that the best fitness function for training the controller would be to use the distance traveled over a set number of game turns. To get the fitness function, the TORCS system had to be used with the controller and the generated speed limit values. To improve efficiency, whenever a new set of dimension values was tested, the fitness was stored in a list. This way, if the fitness value for that specific set of dimension values was ever needed again, the testing program would not have to spend the time waiting for it to be calculated again. Thus, when all input values given to the controller were the same, it would always return the same result.

2) Race results: In examining the videos of the race final, the social aspect of the learned behavior is clear. Our driver tended to find gaps in the pack and slip into them in order to maintain speed and direction. Thus, it used pack information implicitly in its driving rules. The main weakness with the approach is that when it took the lead and extricated itself from the pack it no longer had the pack information to rely on to regulate its speed. So, it tended to over accelerate and spin out. It was clear, that when the driver had extricated itself from the pack that we needed to augment the driving rules and variables to compensate for this.

B. Simon M. Lucas: Hacked controller

The entry *HackedController* was based on a modification of the supplied SimpleSoloController. SimpleSoloController (referred to as the original controller for the rest of this section) is able to drive around most tracks without crashing. It does this by keeping close to the center of the track, by not driving too quickly, and by limiting the maximum steering angle to avoid excessive skidding. On one of the Oval tracks, it was possible to greatly improve its performance simply by increasing its target speed. However, if this was done without taking other compensating measures, then there was too great a likelihood of crashing on more complicated tracks. I did most of the testing on E-Track 3, as this has a good mix of shallow curves, tight curves, and straight sections. On three runs, the original controller scored an average fitness of 4959 metres. Two of the runs had a fitness of over 5500, but on one run it crashed on a tight bend, and scored only 3733. When SimpleSoloController crashes, it often fails to recover, and may waste the rest of a fitness evaluation stuck against a barrier, when simple reversing manoeuvre would free it.

The main approach I took to optimising a controller based on this was to increase the default top speed, while adding in some measures to try to reduce the risk of crashing. SimpleSoloController had a target speed of 100 km/h, but increasing this without some compensating measures could be counter-productive.

The competition API provides an array of range-finder style sensors. A simple approach was taken to summarise the information from these. If any of the range-finders fell below their maximum value, then the car went into a type of safety mode, and set the target speed to 50. Furthermore, if the current speed was greater than the breakSpeedLimit (a newly created variable, optimised to be 105) then the cars brakes were applied (and acceleration set to false). However, if the range finders were all at maximum, then this indicated a straight section immediately ahead, and also that the car was heading close to straight along it. Under these conditions, the target speed was set to 250.

Other changes were as follows. If the car position deviates from the center of the track by more than trackPositionLimit, then steering mode is engaged. Under steering mode, the target speed is reduced by a factor of two in the original controller (from 100 to 50), and by a factor of 1.5 in the hacked controller (from 250 to 167). The variable trackPositionLimit was set to 0.3 in the original, and 0.15 in the hacked version, with the effect that the car makes more effort to stay close to the centreline of the track. In the original controller, the target angle to the track is made equal to the track position. In the hacked version, this value is multiplied by a variable, trackFac. After optimisation, this had value of 0.38. This means that smaller adjustments are made to the steering, in order to reduce the risk of skidding. If the selected steering angle is small (0.01) then it is set to zero in the original controller. In the hacked version, this was changed to 0.005. Its not clear whether this logic makes much difference for either controller, but the thinking behind it is to prevent the car from constantly making tiny adjustments to the steering.

The final significant change was to make the steering adjustment speed dependent. In the original controller a steering adjustment was made that depended only on the distance from the center line of the track, and the angle to that center line. In the hacked controller the maximum steering angle reduced as the speed increased, since setting a large steering angle at a high speed will most likely induce a skid, and the hacked controller is not sophisticated enough to maintain control in a skid. The variable steeringFac, was

introduced to adjust the slope of the line controlling the dependence of the steering on the speed.

The original intention with hacked controller was to extract the main variables into a vector, and then use an evolutionary strategy to evolve the values of those. However, in order to meet the competition deadline there was insufficient time for this, and instead the values of the variables were optimised by hand. The first approach was to observe the behaviour of the car as it was being driven by the hacked controller, and then to make adjustments to the variables to correct problems that were observed. Due to the non-linear interactions of the variables, and the excessive time spent watching the car in visual mode, this approach proved to be ineffective.

Best results were obtained by running the simulation in results only mode, which allows evaluations to be done much faster than real-time, and therefore many more settings of the variables can be tried. The approach taken was a kind of direct policy optimisation, where variables were initially set based on intuition, or taken from the SimpleSoloController. Trials were made with each one adjusted by around +/- 10% or 20% of its current value. The fitness of each setting was noted, and I looked for patterns in values of variables, or combinations of variables that worked well. In total, around 50 trials were made.

The variables adjusted in this way (together with their final values in parentheses) were: steeringFac(0.35), trackFac(0.38), breakSpeedLimit(105), trackPositionLimit (0.15). On three trials (on E-Track 3) this hacked controller scored 7815, 7776, and 7811, giving an average of 7801: a significant improvement over the original controller.

In summary, hacked controller represents a somewhat hastily designed effort to use the available sensor data to improve on the supplied SimpleSoloController. The supplied controller was extended in the ways described above, and a kind of manually operated evolutionary process was used to tune the parameters. This was found to me more effective than tuning the parameters by trying to directly observe their effects on specific aspects of the driving behaviour. It would be interesting to try evolving these values using an evolution strategy, to see whether the manually chosen parameters can be significantly improved on: they probably can.

C. Matt Simmerson: NEAT controller

The idea behind this controller was to evolve a neural network that controlled a racing car around many tracks, equally well, based on a set of input data provided by the racing environment. The added complication for my controller was to evolve the network topology given no domain knowledge a priori.

1) Defining the controllers: The controller was trained using the NEAT [6] algorithm that evolved populations of neural networks, and was created using the NEAT4j [7] software, an implementation of the NEAT algorithm. The NEAT4j implementation allows for the initial selection of a sub set of 3 inputs of the available 29 sources of input data.

The 3 outputs controlled the power in the range [0,1], the gear change in the range [0,1] and the steering in the range [-1,1]. The throttle and brake actions were Boolean so the power output node was used to apply throttle for ¿= 0.6 or apply the brake for ¡= 0.4. For the middle values then the neither the throttle or brake would be applied and hence the controller would coast. The gear change would attempt to change up if the output node was ¿= 0.5 and change down otherwise. The gears selected were limited to [R, N, 1, 2, 3, 4, 5, 6]. As all the output nodes used a sigmoid activation function, the actual values created as the controller actions were scaled from [0,1] to the appropriate range.

The subset of inputs, from the table defined above, available for this controller were:

1) Current speed
2) Angle to track axis
3) The 19 track sensors
4) Track position (with respect to left and right edges
5) Current gear selection
6) The 4 wheel spin sensors
7) Current RPM

All the inputs were scaled to be in the range [-1,1] or [0,1] depending on the sign of the input This prevents large input values completely swamping other, smaller, input signals.

The initial controllers were very simple with 3 randomly selected input nodes connected to one of the output nodes, such that all output nodes were connected to exactly one input node, but an input node could be connected to more than one output node.

2) Controller evolution: The NEAT algorithm, in essence, uses a genetic algorithm to create a neural network topology from a given genome. Each genome consists of a set of node genes that describe an individual neuron and connection genes, which describe a nodes connections.

The population size was just 100 as this was a reasonable compromise between evolution and the time it took for each epoch. The mutation and crossover operators were those defined by the NEAT algorithm. The parent selector function was a tournament round where the allowed parents were pitted against each other with the winner taking the spoils (i.e. the fittest). Recurrency was allowed in this experiment.

3) Training the controllers: The car was trained on just one track, G3, which was not one selected for the initial valuation. This track was selected as it had some varying turns i.e. left, right, curve and also straights of varying lengths into the various corners. Ideally, for the sake of generalisation, I would have liked to train the cars over several tracks; however, the current version of the TORCS environment prevented this.

4) Controller evaluation: The car was tested on the track for a maximum of 4000 steps, equivalent to around 80 seconds real time. If the car sustained more than 100 points of damage (out of a maximum of 10000), the evaluation for that car was aborted.

The overall fitness of the car was calculated thus:

$$Fc = (2 * Dr)do + speedmax + C \quad (1)$$

1) Dr was the distance raced value reported by TORCS engine and could be both positive and negative.

2) d was the value reported by the TORCS engine and has a maximum value of 10000.
3) o is a measure of how much the car stayed on the track. This was necessary to prevent the car using the barriers as a guide with no damage penalty. Until this variable was added, it prevented any really successful controllers. The edge of the track was represented by -1 (left) and +1 (right). The outside value was calculated as 0 if the car was in these limits, and (Abs(track position) - 1) for values outside these edges.
4) Max speed was calculated throughout the cars trial based on the speeds reported by the TORCS engine. This was to try and reward fast cars early on that crashed, as the name of the game is speed.
5) C was used to ensure the fitness value was always positive and was set to 10000. Negative values were created when cars went the wrong way round the track or had high-speed crashes near the start, resulting in large damage.

Whilst the 4000 time steps, used for evaluation, represented 80 seconds real time, the TORCS engine allowed a non-GUI version, which was evaluated in around 3-4 seconds. With the population was set to 100, each evolutionary epoch lasted 400 seconds. The entry, for Neat4J, was selected from the winning phenotype from the 170th generation i.e. nearly 19 hours on my dual core laptop. This had a good level of performance over a number of different track types e.g. oval, twisty etc.

D. Diego Perez and Yago Saez: Rule-based controller

The idea of this controller is to evolve a set or rules that drives a vehicle, using sensors as input data. The usage of sensors to obtain autonomous driving has been addressed by numerous researchers ([8], [9], [10], [11]), as well as the use of evolutionary algorithms in this field ([12], [13], [14]).

1) Input data, effectors and rules: The input data is discretized from the values of four *sensors*: **angle**, discretized to [0,4], where 0 means the smaller angles; **trackPos**, with a discretization performed in a range [0,1], where 0 means centered on the track and 1 the car near the edges; **speedX**, in a range [0,3] where 0 means lower speed than higher values; and **track**, where only three of these sensors have been used (front and inmediate sensors on right and left) and discretized to a unique range [0,2], where 0 means that a track edge has been detected beyond 20 meters, 1 when the track edge is up to 20 meters and 2 in case no track edge is detected.

A key part of the design is the usage of symmetry for the first two sensors. This concept works using the absolute value of the sensor to match to the proper discretized value. The objective of this approach is to avoid duplication of efforts by reducing the search space.

The *effectors* of the controller have been designed as follows: **throttle and brake**, where both pedals have been codified in a common output to avoid non-sense values (as full gas in both pedals simultaneously). Hence, a unique value is applied and both gas pressures are extracted from it; **steer**, codified as a real number, from -1 to 1, and discretized with a precision of 0,1; **gear**, which changing process consists on increasing the current gear when the rpm value is higher than 6000 and decreasing it if it is below 3000. The discretization and codification applied over the input data and effectors allows us to create a set of 120 rules, where conditional part is composed by the sensors, and the actions are formed by acceleration, braking and steering. These rules compose the base individual.

2) Application of an evolutionary rule system: Traditional random initialization of individuals used in evolutionary techniques do not work properly in this field, because it is almost impossible to obtain a configuration that drives the vehicle correctly by chance. This is the reason of getting a base individual before evolve it to obtain better results. The algorithm used to get this base individual is a generation of a subset of rules that allows the vehicle to end a lap, minimizing the angle of the car with the track axis. Each one of this rules is created by testing how each allowed combination of acceleration and steering behaves when the condition of the rule is triggered.

Once we get the base rule set, the evolving individual is extracted from it, taking all its rules. Therefore, the individual is composed by a set of rules, each one of them formed by condition and effectors, that need to be evolved to obtain the controller. To evolve this individual, the algorithm executes evolutionary steps until a stopping criteria is reached.

The evaluation of the individual is performed recording lap time and damage suffered, setting the fitness using a linear combination of both values, with weights of 0.4 and 0.6 respectively, in order to avoid overfitting to the training circuit. In this system, we can not decide when a rule is better than another because the behaviour of the individual depends on the whole set of rules used. Because of this, selection operator has been designed as a random pick-up from the rules pool, taking two of them to apply uniform crossover. Finally, mutation operator is performed over the new rule, applying an addition of ± 1 unit to the left part and ± 0.3 to the effectors of the rule (obeying limits and codification precision).

The next step in this algorithm consists of searching for a rule from the individual where its conditional part is most similar to the new rule. This rule is extracted from the pool and the new one substitutes it. The new set of rules is then evaluated and its fitness is compared with the one calculated before inserting that new rule. Only if the new rule set is worse, the substituted one is retrieved and the new rule is eliminated.

Results have proved this algorithm to be effective, reducing lap times of the base individual in few generations, keeping the car damage almost nonexistent. The usage of symmetry, however, brought a side effect that was not expected: the car drives in a smoothly zig-zag trajectory centered on the circuit. This is because a small steering value can center the vehicle on the track, but not necessarily drive it parallel to the track axis. Nevertheless, the controller

TABLE III

RESULTS OF THE FIRST STAGE OF THE EVALUATION PROCESS. THE REPORTED ARE THE MEDIAN OVER 10 RUNS.

Entry	Ruudskogen	Street-1	Speedway
Kinnaird-Heether et al.	6716.7	3692.9	14406.9
Lucas	4134.2	5502.8	12664.5
Simmerson	5934.0	6477.8	12523.3
Perez et al.	3786.9	2984.8	-317.3
Tan et al.	3443.5	2998-5	10648.2
C++ Sample Controller	4465.1	4928.8	7464.5
Java Sample Controller	5593.8	2963.2	5689.9

TABLE IV

RESULTS OF THE SECOND STAGE OF THE EVALUATION PROCESS. THE SCORES REPORTED ARE THE MEDIAN OVER 10 RUNS.

Entry	Ruudskogen	Street-1	Speedway	Total
Simmerson	10	10	6	26
Kinnaird-Heether et al.	4	8	10	22
Lucas	6	6	8	20
Tan et al.	5	5	5	15
Perez et al.	5.5	4.5	5	14

behaved in a reasonable way, only with the exception of some specific circuits: the oval ones. These circuits, similar to Nascar tracks, have banked curves which make the zig-zag movement completely uncontrollable.

E. Chin Hiong Tan and Kay Chen Tan

The entry submitted by Chin Hiong Tan and Kay Chen Tan was developed in a three-step process. First, the sensory information was aggregated and preprocessed; second, a parametrized controller based on simple rules was designed; finally, the parameters of controller were optimized using evolution strategies. The resulting controller drives in the direction where the rangefinder sensors indicate the largest free distance, with a speed dependent on that distance.

IV. RESULTS

The entries were scored through a two stages process which involved three tracks available in TORCS: the Ruudskogen, the Street-1 and the D-Speedway. The first (warm up) stage was aimed at eliminating particularly bad performing controllers. Each controller raced alone in each of the three tracks and its performance was measured as the distance covered in 10000 game tics (approximately, 200 seconds of actual game time). For each of the three selected tracks, we run each controller ten times. The performance has been computed as the median (the 50th percentile) over the ten runs to avoid any issue about skewness. Table III compares the performance of the five controllers submitted to the one of the two sample programmed controllers provided by the organizers. The results show that the controller submitted by Leonard Kinnaird-Heether and Robert Reynolds outperforms the other controllers in all the tracks but the Street-1 track. As can be noted, the performances of the controllers are highly different among the three tracks but they generally compare well to the performances of the sample controllers provided by the organizers. In particular, the entries submitted respectively by Leonard Kinnaird-Heether et al., by Simon Lucas and by Matt Simmerson (the first three controllers reported in Table III) performs consistently better than the sample controllers almost in all the three tracks. As all the five submitted controllers performed well on the first stage, none of them was eliminated from the second stage, in which the controllers competed together in each of the three tracks. In this stage, the task consisted of completing three laps and each controller was scored based on its arrival order using the same point system used in F1: 10 points to the first controller that completed the three laps, 8 points to the second one, 6 to the third one, 5 to the fourth, and 4 to the fifth one. Ten runs for each track were performed using as start grid a random permutation of the competitors, in order to test the reliability of the controllers' performance. Then, the score of a controller on one track was computed as the median of the scores obtained during the ten runs. The final score for each controller was finally computed as the sum of the points collected on each track. Table IV shows the final scoreboard: Matt Simmerson won the competition with 26 points, followed by Leonard Kinnaird-Heether et al. with 22 points, by Simon Lucas with 20 points, by Tan Chin Hiong with 15 points, and finally by Diego Pérez 14 points. This results suggest that although the controllers submitted by Kinnaird-Heether and Reynolds is fast, the one submitted by Simmerson is more reliable, especially in the presence of other controllers. Finally, it is worthwhile to underline that the second stage of the evaluation process suggested that all the submitted controllers have poor overtaking and obstacle-avoidance capabilities, whereas these features are very important to succeed in a racing competition. Additional results and a video with the highlights of the competition are available on the webpage of the competition [15].

The reason Simmerson's controller won over Kinnaird-Heether and Reynold's was probably that the latter had been optimized for racing on tracks with smooth curves, in the presence of other cars. Simmerson's controller had been trained on the "G3" track that included sharp turns, like Ruudskogen, but on its own. Both of these controllers were optimized with stochastic algorithms, and it stands to reason that such approaches outperform the hand-tuning used by Lucas.

V. THE FUTURE OF THE CAR RACING COMPETITION

While this competition differed greatly from the competitions organized during 2007, in that a more sophisticated racing game was used, there was also a great deal of continuity. Not only in that some of the participants of the 2007 competitions also participated in the current competition, but also in the similarity of rules and arrangements. The organizers believe that this continuity is very important for the competition to be successful. We need a high participation level to ensure that a broad spectrum of approaches are represented, and regular repetitions of the competition to ensure that the participants have time

to perfect their approaches. Our aim is to ensure further continuity through holding a series of future competitions using gradual refinements of the rules and software used in the current competition.

The following improvement will be made to the software in time for the CIG competition:

- The installation process will be streamlined
- Reliability will be improved
- Support for multi-car and multi-track training will be added (making it easier to apply co-evolution and incremental evolution)
- More sample controllers and trainers, e.g. temporal difference learning trainers, will be supplied

An amusing illustration of the need to improve reliability is that in an early version of the software it was possible to achieve the fitness value of driving a whole lap, simply by slowly driving up to and passing the start line (the car starts 100 meters before that line), then turning and passing the line again. This flaw is inherent in TORCS, presumably because its developers never thought of anyone doing something so bizarre. Evolutionary algorithms, however, are good at coming up with bizarre solutions, and Matt Simmerson's algorithm quickly evolved a controller that exploited this bug. A patch for this bug is now part of the software package.

A reviewer of the paper summarizing the previous car racing competitions [1] pointed out that in its current form, the competition is not only about learning algorithms. It is certainly possible to hand-code a non-learning controller that outperforms the best CI-based controllers. Indeed, the best controllers that come with the TORCS game (developed by the TORCS developers) are non-learning and by far outperform all the controllers submitted to this competition so far, though they often access information state information that is not directly available through the competition API.

Of course, we hope that future editions of this competition will see CI-based contributions that perform better than the best hand-coded ones, and there are no reasons why this should not happen. Still, it would be interesting to run a version of the competition that compared only the quality of the learning algorithm. One way could be to define a standard (e.g. neural network-based) controller architecture, and then provide an interface for a learning algorithm to set the parameters for this controller optimally given a certain numbers of laps around an unknown track. The participants would then submit an algorithm rather than a controller, to be run and evaluated by the organizers of the competition.

Another interesting version of the competition would be one where the controllers where presented with a richer but more primitive state description, in particular visual data. This could come in the form of the full rendered 3D view through the controlled car's windscreen, or a part of it. Such a state description would ultimately give the controllers more information and thus allow for better driving, but would also require more complex controllers.

Given the various interesting variations on the car racing concept that are possible, our plan is to organize editions of the car racing competition in conjunction with several international conferences, and at each conference hold both the competition in its original form, and some variation on the concept like the ones suggested above.

VI. Conclusion

We have described the organization, rules and software of the car racing competition in the form it was organized in conjunction with IEEE WCCI 2008. Four out of five participating teams described the architecture and training of their controllers. We have also reported the scoring procedure and results of the competition, and plans for future competitions. We hope that this paper, in addition to serving as a record of the competition, will provide organizers of similar competitions with inspiration and insights, and that the descriptions of the controllers will be useful for researchers working on learning vehicle control in general, and for participants in future car racing competitions in particular.

References

[1] J. Togelius, S. M. Lucas, H. Duc Thang, J. M. Garibaldi, T. Nakashima, C. H. Tan, I. Elhanany, S. Berant, P. Hingston, R. M. MacCallum, T. Haferlach, A. Gowrisankar, and P. Burrow, "The 2007 ieee cec simulated car racing competition," *Genetic Programming and Evolvable Machines*, 2008. [Online]. Available: http://dx.doi.org/10.1007/s10710-008-9063-0

[2] "The open racing car simulator." [Online]. Available: http://torcs.sourceforge.net/

[3] "Software manual of the car racing competition." WCCI-2008. [Online]. Available: http://cig.dei.polimi.it/wp-content/uploads/2008/04/manual_v03.pdf

[4] R. G. Reynolds and M. Z. Ali, "Computing with the social fabric: The evolution of social intelligence within a cultural framework," *IEEE Computational Intelligence Magazine*, vol. 3, no. 1, pp. 18–30, 2008.

[5] R. G. Reynolds, M. Z. Ali, and T. Jayyouzi, "Mining the social fabric of archaic urban centers with cultural algorithms," *Computer*, vol. 41, no. 1, pp. 64–72, 2008.

[6] K. O. Stanley, "Efficient evolution of neural networks through complexification," Ph.D. dissertation, Department of Computer Sciences, University of Texas, Austin, TX, 2004.

[7] M. Simmerson, "Neat4j homepage," 2006. [Online]. Available: http://neat4j.sourceforge.net

[8] S. Baluja and R. Caruana, "Removing the genetics from the standard genetic algorithm," in *Proceedings of the international conference on machine learning (ICML)*, 1995.

[9] R. Sukthankar, S. Baluja, and J. Hancock, "Protoyping intelligent vehicle modules," in *Proceedings of the International Conference on Robotics and Automation (ICRA)*, 1997.

[10] J. Togelius and S. M. Lucas, "Evolving controllers for simulated car racing," in *Proceedings of the Congress on Evolutionary Computation*, 2005.

[11] ——, "Evolving robust and specialized car racing skills," in *Proceedings of the IEEE Congress on Evolutionary Computation*, 2006.

[12] J. Bernard, J. Gruening, and K. Hoffmeister, "Evaluation of vehicle/driver performance using genetic algorithms," *Society of Automotive Engineers*, 1998.

[13] D. Floreano, T. Kato, D. Marocco, and E. Sauser, "Coevolution of active vision and feature selection," *Biological Cybernetics*, vol. 90, pp. 218–228, 2004.

[14] J. Togelius and S. M. Lucas, "Arms races and car races," in *Proceedings of Parallel Problem Solving from Nature*. Springer, 2006.

[15] "The car racing competition homepage." WCCI-2008. [Online]. Available: http://cig.dei.polimi.it/?page_id=5