Original Scientific Paper

doi:10.5937/jaes14-10931 Paper number: 14(2016)2, 376, 239 - 247

DEVELOPING SELF-MODIFYING CODE MODEL

Goran Đurić*
University of Belgrade, Faculty of Mechanical Engineering, Belgrade, Serbia
Časlav Mitrović
University of Belgrade, Faculty of Mechanical Engineering, Belgrade, Serbia
Goran Vorotović
University of Belgrade, Faculty of Mechanical Engineering, Belgrade, Serbia
Ivan Blagojević
University of Belgrade, Faculty of Mechanical Engineering, Belgrade, Serbia
Miloš Vasić
University of Belgrade, Faculty of Mechanical Engineering, Belgrade, Serbia

This paper presents the technology of constructing and linearization of binary program utilized for program generation, analysis and transformation into a self-modifying code. An example model of the self-modifying software system and its experimental application in vehicle control have been presented in this paper. The module responsible for vehicle control comprising two subsystems has been created within the simulation software. The first subsystem has emerged through the classical software process developed by a human-programmer. The second subsystem has been created as a result of a separate piece of software substituting the part of a programmer in a software process part. The result of this approach is software creation in conjunction with natural and Artificial Intelligence in addition to experimental integration into the vehicle control system.

Key words: Genetic programming, Self-modification, Software, Software process, Artificial intelligence

INTRODUCTION

A Self-modifying code, program or software are different names used in order to describe software systems having the property to change independently in some form. The self-modifying code has a long history. In one of its first applications, it was used as a way of hiding instructions for copy protection. Key commands were not visible in the code. Neither did they “appear” in working memory at the moment of carrying out a program. In his famous text, published in Scientific American in 1984, Dewdney presents “core wars” simulation [1] in which two opposing programs sharing the same memory space are trying to disable each other by utilizing peek and poke commands, i.e. by alternating reading and writing on arbitrary memory locations, followed by attempts at preserving integrity of each individual code by means of allocating and repairing code parts.

The self-modifying code makes an analysis difficult thereby preventing the unwanted reversible engineering [02]. Computer viruses, for instance, try to hide their internal structure and the recognizable signature by utilizing the self-modifying code. The self-modifying code should not be confused with “at the moment of execution” generated code used by the Java Virtual Machine, for example. The reason for applying the self-modifying code can be also the limited working memory space or the required resistivity to failure that was the case in designing the operating system for the Space Shuttle program [03].

The software self-modification can be a solution even in cases when the unpredictable environment change is expected and required during the execution. Such software system that has the possibility to adapt to changes represents a type of Artificial Intelligence. Artificial neural networks represent one of the most popular AI technologies. They are most frequently used for systems having complex and conditioned characteristics, and where the solution cannot be explicitly defined. One of the important properties of neural networks is the capability to learn. Genetic Programming (GP) as an aspect of au tomatic programming is particularly interesting to us. In AI, GP is a methodology of finding a computer program that solves certain task, and

* *Faculty of Mechanical Engineering, Kraljice Marije 16, 11000 Belgrade, Serbia; gdjuric@mas.bg.ac.rs*

The greatest contribution to the development and application of GP in the field of diverse problems was provided by John R. Koza [4]. The basic difference between a genetic algorithm and GP lies in the way of representing solution, in which case the genetic algorithm creates a series of signs as a solution, whereas GP creates a computer program as its solution. Essentially, this is a situation in which a computer program programs a computer program.

This paper presents a model of self-modifying software system and the experimental application of the model in the domain of vehicle control in simulation.

Softver process model

A software process, or a process of creating software, is a set of activities the aim of which is software development. The international standard ISO/IEC 12207 [5] establishes the general working framework and describes software life cycle processes. This standard can be implemented also in cases when a piece of software is regarded as an independent entity or when it is an integral part of a larger system. A large number of software process models have emerged through software engineering development. A software process model is an idealized layout of software process, that is, an abstract representation of such a process. The most famous models of software processes are classical phase – waterfall model, iterative model, V model, prototypes, spiral model etc. Three phases are usually noticed, these being: analysis phase wherein the domain is being analyzed, design phase wherein the solution is designed, and the implementation phase in which a concrete software solution is created and applied. Sommerville [6] considers that each software process has the following phases: specification – defining what a system should do, design and implementation – defining the system organization and system implementation, validation – checking out whether a solution is what has been required it to be, and evolution – changing the solution in accordance with new requirements.

In Figure 1, we propose a spiral model version of the software process with the aim of introducing AI in the software creation process. This software process model implies minimizing the human role in a process part that keeps repeating in a spiral with the idea of constant automated redesign and implementation. Such a model entails the need for intelligent software that should participate in design phases, implementation and evaluation of the target software solution. The complex software system comprising thus defined intelligent development software and tightly connected target executable software can be regarded also as the self-modifying software system.

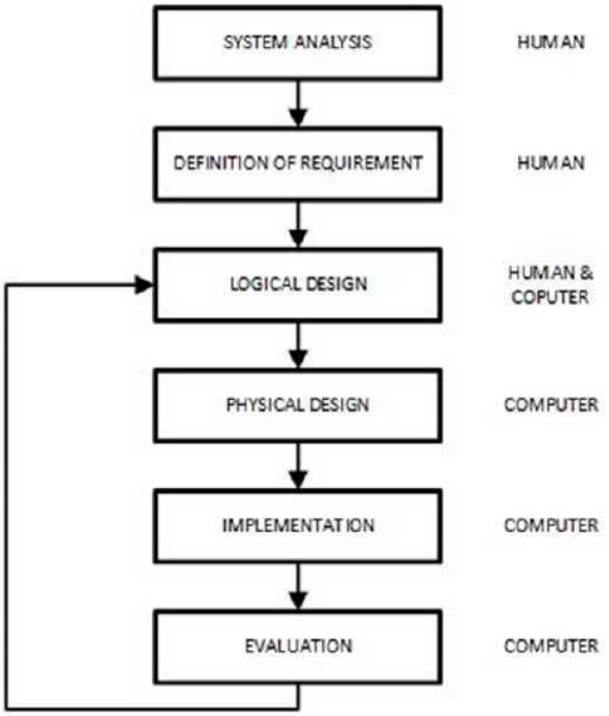

Figure 1: The observed model of a software process

The described intelligent development software has been labeled as Movens (In Latin: a starting device, the one that initiates moving). Movens creates an autonomous software solution based on the input information, initial logical model solution and the accumulated experience. It partially takes part of a software designer and programmer. The ultimate aim of the Movens operation is to create the autonomous software solution that is nearer to the stated goal by repeating its activity cycle through multiple iterations. Thus created solution will have the equal operation speed in its operation, as would be the case for the one created by classical means.

The Movens model provides for a significant role of intelligent development software and precise work distribution with the human being in the software development process. As a consequence, there appears to be a module for performing diverse tasks in development phases, as is the case with, for example, a module for coding into

a pseudo-code, a module for converting a pseudo-code into a concrete programming language and the like...

At a higher level, at the level of logical design solution, Movens could make use of UML diagrams [7] for writing the domain model and developing solution model. The basis of this model contains an idea of a modified software solution aimed at achieving the goal and the best solution. The proposed general modification algorithm is shown in Figure 2. In the most general case, three or more nested modification levels are provided for, or more precisely: the logical modification, implementation modification and parameter modification.

```
1.  start
2.  input: initial LSM (logical solution model)UML-XML
3.  processing: logical modification LSM
4.  processing: pseudo encoder
5.  output: initial SPC (solution pseudo code)
6.  processing: implementation level modification of SPC
7.  processing: language specific encoder
8.  output: initial SC (solution code)
9.  processing: parametric level modification
10. external operation: running&testing of solution
11. question: if par.mod. is not ok then go back to 8.
12. question: if imp.mod. is not ok then go back to 5.
13. question: if log.mod. is not ok then go back to 2.
14. end
```

Figure 2: Modification algorithm

The application to developing softvare for vihicle control wihin thesimulation softvere

In this paper, we applied the described approach to software creation to an example of realization of vehicle control software. For the purpose of this paper, we used the TORCS (The Open Racing Car Simulator) [8], a software simulator of car races. More precisely, it is the open-source multiplatform software written in C++ programming language. It is used in different scientific and research projects in addition to academic instruction at faculties. It provides quality visualization (Fig. 3), but is not supposed to be an alternative to commercial entertainment programs. It represents a framework for research and comparison of diverse solutions based on Artificial and Computer Intelligence.

The simulation vehicle can be automatically controlled by a selected software controller. The controller is implemented as a software module that is programmed separately and simply installed in the simulation. The basic application during the execution of each step of a work cycle enables the controller to access the data on the current simulation condition, the data on vehicle condition, path and other vehicles, and after the internal analysis and decision processes and the ability to carry out vehicle control.

Figure 3: TORCS visualization

The controller perceives the current simulation condition by means of reading the simulated sensor values. The most important pieces of information are obtained from the distance sensor set. There are 36 defined distance sensors from other vehicles (i.e. opponents), which are uniformly distributed 10 degrees each in the circular area around a vehicle. Additionally, 19 distance sensors are provided from the track boundaries that are distributed at 90,75,60,45,30,20,15,10,5 and 0 degrees to the left and to the right in relation to the vehicle motion direction taking into account also the vehicle motion course. Each sensor has the maximum scope comprising 200 meters. There are sensors of vehicle lateral deviation from the mean path, the angle sensor forming a vehicle in relation to the path axes and the sensor indicating the distance from the path start. The information on the current vehicle gear movement, the vehicle motion speed, and engine revolution. In contrast to simulated sensors, in addition to enabling the reading of the current condition the provided simulated actuators make possible direct value change thereby enabling vehicle control. These are acceleration actuators (accel) that simulate the regulator of „gas“ engine revolutions per minute the value of which can range from 0 to 1, brake actuator that simulates brake pedal and can have the value ranging from 0 to 1, steer actuator simulating the control wheel i.e. “the steering wheel” of a vehicle can have values ranging from -1 to 1, where -1 indicates the leftmost, and 1 the rightmost positions and the gear change actuator simulating

the vehicle gearshift and can have the values: -1,0,1,2,3,4,5 and 6. The improvement developed by Loiaconno [9] enabled the client-server architecture for TORCS thereby separating physically the controller entirely from the basic application and enabling the controller development in an arbitrary programming language.

The Movens model of software development was applied in the part of vehicle software controller that refers to the vehicle gearshift control (Figure 4.). This was enabled by controller functional distribution into two software components. The controller part aimed at controlling other vehicle controls was used solely as a basic code providing for vehicle movement from the beginning to the end of the defined path [10].

In the paper by E. Onieva et al. [11], the modular structure of a software controller was also proposed. These are the modules responsible for transmission rate change, acceleration and brake processes, determining the desired speed, steering wheel control and a module for other vehicles. Gearshift module implements simple operation rules based on the steadily defined table the basic parameter of which is the engine revolutions per minute. In the paper by T.S.Kim et al., the methods of evolutionary strategy are used for optimization of the autonomous vehicle controller [12]. The propose parameter set optimization used by the previously developed controller. Shichel and Sipper utilize genetic programming in order to develop their vehicle controller [13]. They use GP in order to bring about the Lisp population evolution of an expression (Lisp being a programming language). The controller is governed by means of two Lisp expressions, one being a speed actuator, the other being turn actuator. Similarly, Ebner and Tiede develop their own vehicle controller [14] utilizing GP with a program represented in the form of a tree. In essence, their controller has two generated symbolic expression that enable the control of turning and accelerating/slowing down of a vehicle. They used the famous Evolutionary Computation in Java package (ECJ). The optimization of transmission rate change is in the focus of the paper put forward by A.P.Becher and C.Stoean [15]. The authors find optimum values for the engine rpm number as the basis for the table of transmission rate change by utilizing the familiar heuristic optimization algorithms, firstly the bottom-up algorithm and then the simulated development algorithm.

In our solution, we imposed the requirement for the gearshift control that based on the input parameters, such as current vehicle movement speed and the gearshift transmission rate, it should determine the new transmission rate as its output. The process of making this decision can be represented in the form of binary decision tree that can be transformed into a programming code. Given the fact that our newly proposed model of software development provides for automatic solution creating and modification, we applied GP for the procedure of solution finding (Figure 5). It is generic programming that enables us to search the plausible program space and find the programs that solve well the set task.

Within the tree that represents the only possible solution, the nodes can be internal and external.

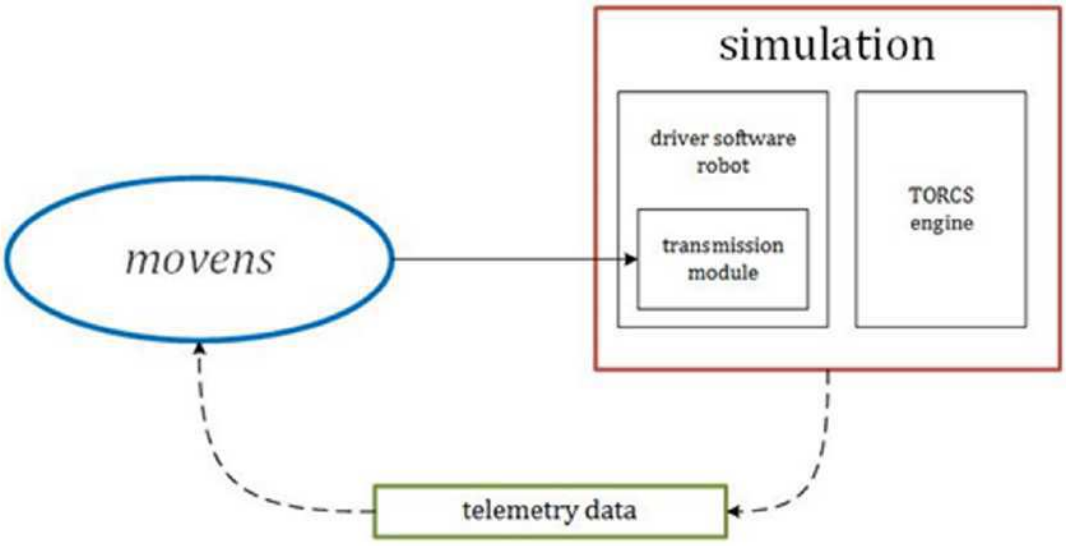

Figure 4: Controller and Movens

In this case, the internal nodes represent the logical function – a question, and the external nodes that are terminal nodes as well, represent the operation of assigning a value to the output parameter. The procedure of generating the initial population uses the growing method of generating tree. The method starts with the random node generation without limiting the tree depth, and with limited maximum node number.

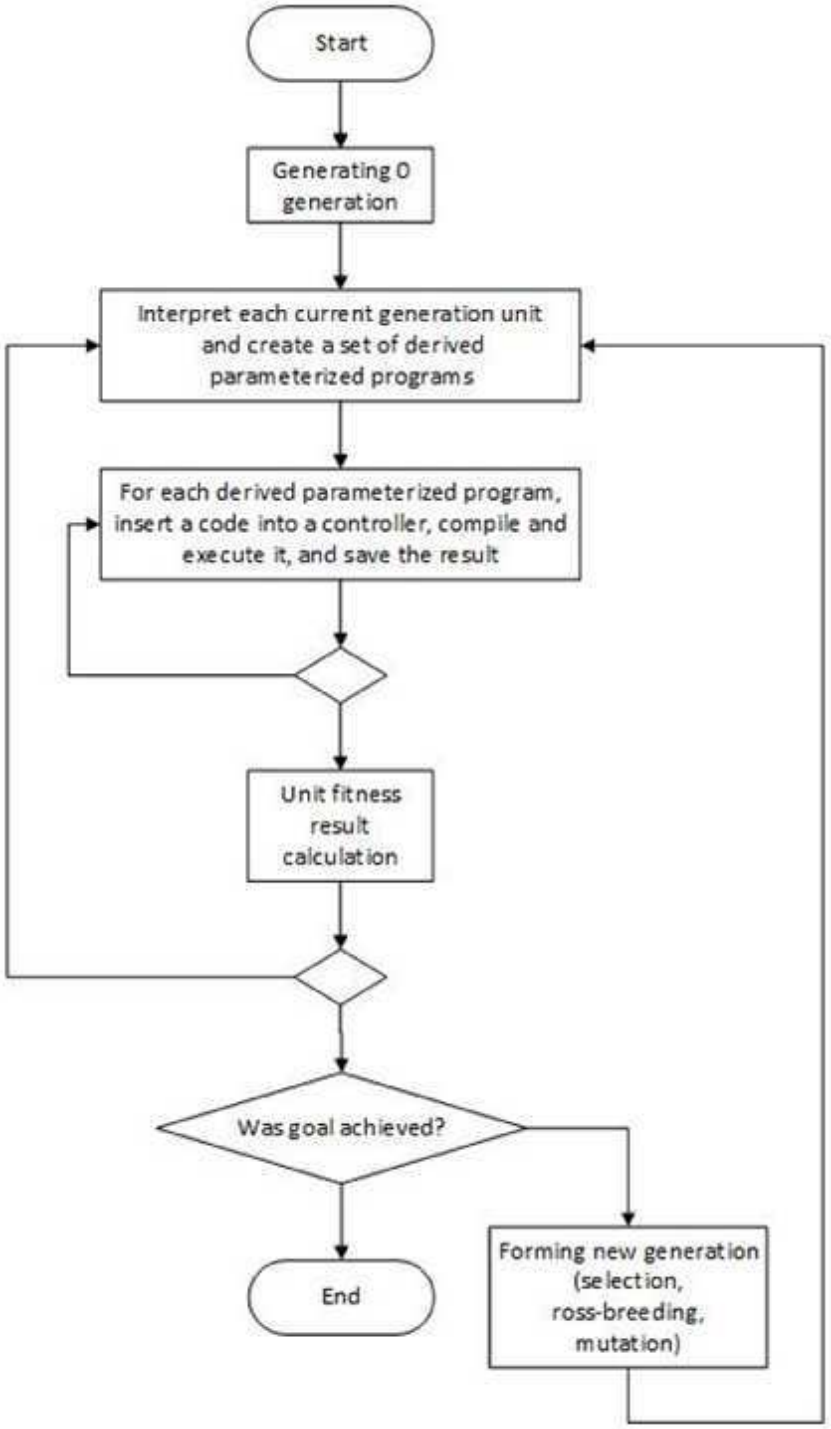

Figure 5: The applied GP algorithm

The initial node – the root, must be the function so as not to reduce the tree to a single node. Tree generation ends also in the case when there are no free internal nodes.

These rules lead to generating diverse tree forms and different tree sizes (Figure 6). The initial population was formed by 100 randomly generated units.

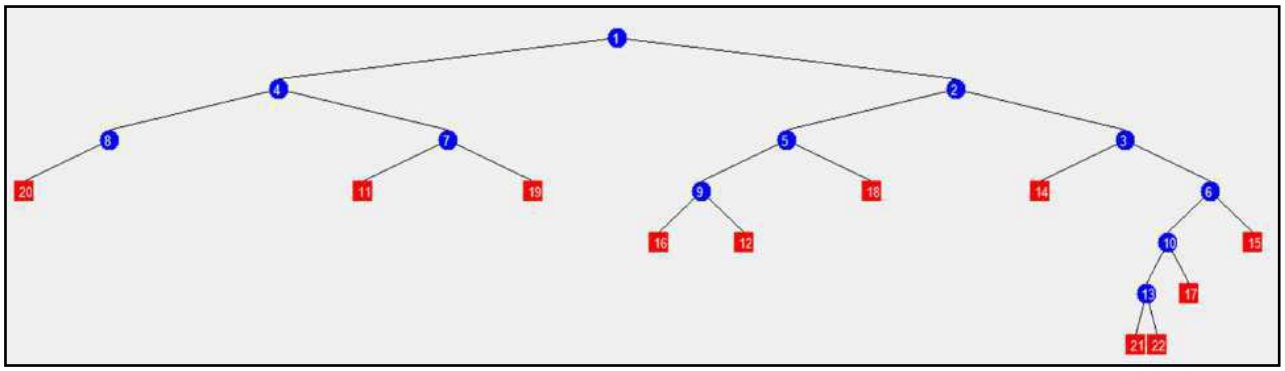

Figure 6: The display of randomly generated tree

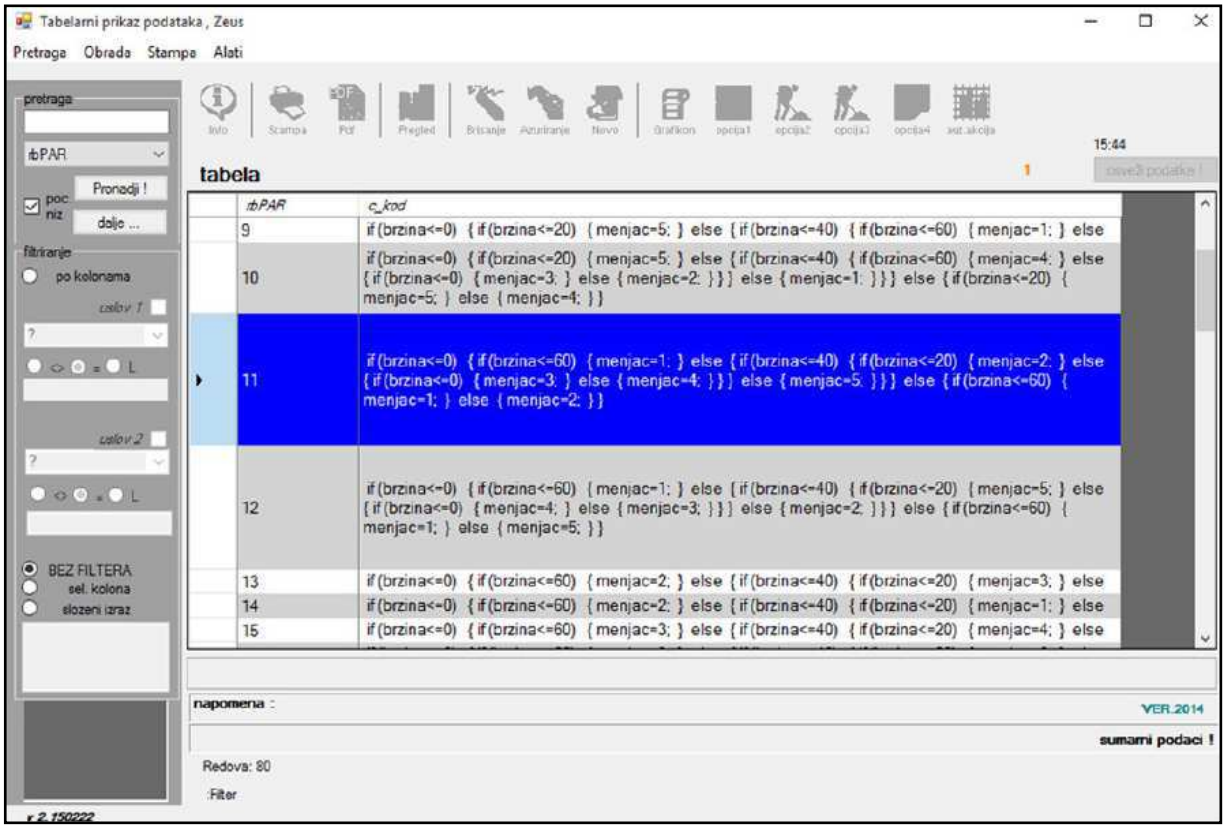

Figure 7: The unit in the form of C code

Measuring fitness – unit grading is realized by means of the real test of the formed program operation in the simulator session. First of all, the unit should be filled in by randomly generated constants (i.e. ephemeral random constant). This issue was resolved by forming the set of 64 unit versions with different constants. Each formed unit with these parameters, represented in the form of C code (Figure 7) is being inserted into the vehicle controller design for TORCS. The newly formed controller is the tested in the batch defined simulator session accompanied by memorizing the defined telemetric data. The data are memorized within the XML file [16] for subsequent processing. The basic parameter for unit grading is the time necessary to carry out the assigned drive. The units with shorter time of simulation execution are given better grades.

As a criterion of process finalization, we established the creation of 100 population generations. In order to form a new generation, we used the method of generation selection wherein the best units are selected as the basis for a new generation. Evolution parameters determining the rules according to which the GP procedure is being carried out are defined as fixed ones in this case. The reproduction is applied to 30% of population by repeating the tournament selection of three units until the necessary unit numbers trans-

ferred into a new generation without changes have not been provided. The crossbreeding is applied during the creation of 50% of new units of a new generation. The crossbreeding of two units generates two descendants in such a way that they mutually change the sub-trees out of the given internal node. The mutation is applied for creating 10% of the population. The randomly selected unit and randomly selected node become the place where the sub-tree is being erased, and instead of it, a new sub-tree is formed by the same procedure. The last 10% of the new generation is formed by creating utterly new units. The new generation thus formed re-enters the testing and grading processes.

The complete described procedure is carried out automatically, controlled by a separate C# control application (Figure 9) specially written for the purpose of this paper. There are two major functionality groups within this application. The one provides the basic operation framework for genetic programming, and the other one enables controlling the external creation control and controlling the test executive programs that emerge as a result of genetic programming.

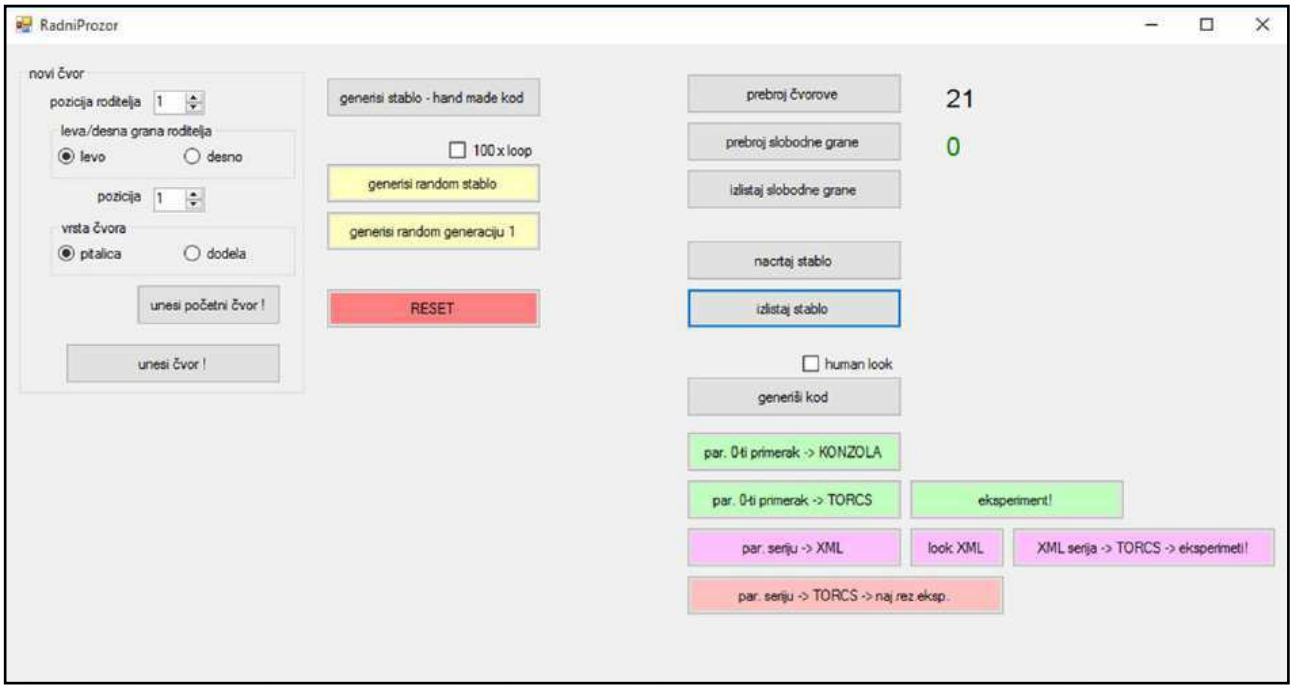

Figure 9: the control application Movens-GP

TESTING IMPLEMENTATION

In the simulation software TORCS, a great selection of over 30 pre-defined paths were offered in order to test controllers. It is possible to select around 42 offered vehicle models. As a task for fitness grading of each controller specimen, we set up vehicle control from the beginning to the end of the path labeled as “e-track-4” the length of which totals 7042m. We did not place any other vehicles or obstacles of other sort preventing vehicle movement in the simulated environment, except for the form of the above stated path.

Although simulation software has the ability to operate in the client-server manner mode, we implemented the basic movement form and control switching. This means that each controller specimen was independently compiled and loaded as a separate module during the simulation execution. We did not use the possibility to access data on path that the basic application has in the course of execution. The controller independently provided memorizing telemetric data (Figure 8 and Figure 9), out of which we actively used primarily the spent time for path covering in this experiment.

It was initially envisaged that formation of 100 program population generations should be carried out. However, since the best result, reached in the 24th generation, had stopped further enhancement, the further process was interrupted by the definitive 30th generation that can be seen in the Table 1. We are of the opinion that the most appropriate time per generation was constantly improved due to direct reproduction of 30% of each generation.

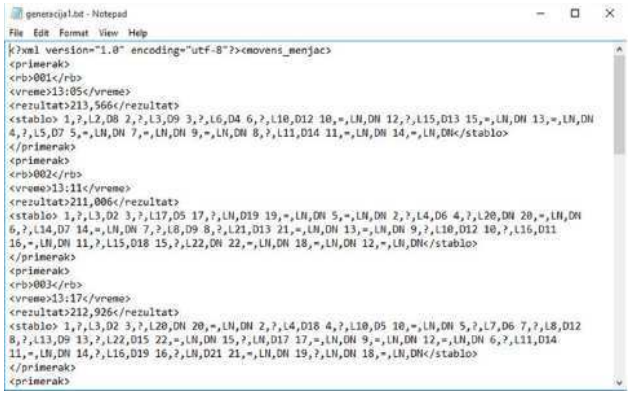

Figure 8: XML file with the simulation data

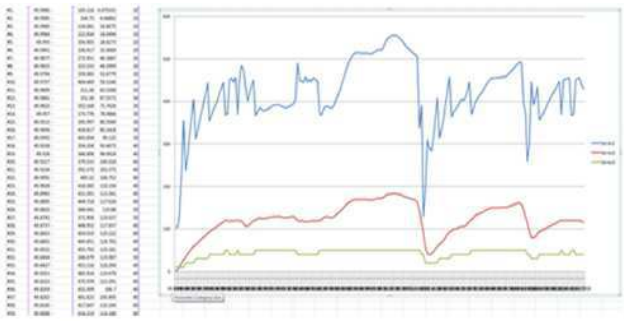

Figure 9: Telemetric data memorized by a controller

Table 1: Fitness statistics

Gen.	the best time	average time	Gen.	the best time	average time	Gen.	the best time	average time
1	220,55	248,5	11	149,78	175,66	21	147,5	162,77
2	212,25	220,76	12	149,78	177,89	22	147,5	160,87
3	205,25	215,5	13	147,8	178,52	23	147,27	164,85
4	205,25	220,97	14	147,8	180,89	24	147,25	160,12
5	187,5	200,99	15	147,8	175,22	25	147,25	160,15
6	160,45	200,58	16	147,75	170,86	26	147,25	159,99
7	149,88	170,87	17	147,7	170,85	27	147,25	159,9
8	149,88	169,57	18	147,66	169,58	28	147,25	160,98
9	149,88	168,5	19	147,6	160,57	29	147,25	159,86
10	149,88	175,89	20	147,5	161,55	30	147,25	159,81

The best-realized time, achieved by a controller specimen equals 147,25s. This is the result we approximately succeeded in achieving by direct vehicle control instead of automatic control. The average time of the first (random) controller generation totaled 248,50s, so that the ratio of the best-achieved time and the average zero generation equaled 1,69. Obviously, there was significant improvement of generated controllers through the evolution process.

Testing was carried out by means of the computer with Intel i7 4Ghz processor, in script mode without graphic display. The total of 153600 generated code cycles, compiling and simulation execution (24 generations x 100 units x 64 parameter sets) were realized, which lasted approximately 8 days, 20 hours and 10 minutes. The approximate time necessary for one test comprised 4,97s if other less demanding programming operations were not taken into consideration. This indicates high computing requirement of genetic programming procedure carried out in this manner.

CONCLUSION AND FURTHER PLANS

The new proposed model of self-modifying software system in this simple example have shown the potential for application and further development. Self-modification as a method envisaged by the Movens model was in this case provided by implementing the genetic programming procedure.

In experiment with using these or similar simulators, client-server operation mode should be utilized in order to enable solution coding in a programming language that is being interpreted. Thus, the operation cycle would be significantly accelerated. In certain future experiments concerning the same domain and task, the logic of a module that is developed by AI methods can be expanded into more complex parameter set.

Forming single solution specimens by including parameter sets has been shown as particularly time consuming. It would be worth to explore GP implementation to finding the best parameter set for each basic specimen that is being grades, which would represent using GP within GP.

For the sake fulfillment of the above described, initial research of application of evolution computing, meta-heuristics, artificial neural networks and above all genetic programming have all been carried out. The experience drawn from this paper will be of use in further research on the subject pertaining to the implementation of self-modifying software system. In addition to this, attention is paid to the issue of representing and installing the knowledge on the domain into the control application.

REFERENCES

1) Dewdney A.K.: Recreational Mathematics – Core Wars (May 1984). http://www.koth.org/.
2) Nikos Mavragiannopoulos, Nessim Kisserli, Bart Preneel: A taxonomy of self-modifying code for obfuscation, Computers & Security 30 (2011)
3) C. Enrique Ortiz: On Self-Modifying Code and the Space Shuttle OS http://weblog.cenrique-ortiz.com/computing/2007/08/18/on-self-modifying-code-and-the-space-shuttle-os/
4) Koza john R.: Genetic Programming: On the Programming of Computers by Means of Natural Selection. Cambridge, MA: The MIT Press
5) ISO/IEC 12207, http://www.iso.org/iso/iso_catalogue/catalogue_tc/catalogue_detail.htm?csnumber=43447
6) Sommerville I: Software Engineering, 7-th Edition. Addison-Wesley, Harlow, England, 2005. http://www.software-engin.com
7) Booch G., Jacobson I., Rumbaugh J., "UML User Guide", Addison-Wesley (1998)
8) B. Wymann, E. Espié, C. Guionneau, C. Dimitrakakis, R. Coulom, A. Sumner. TORCS: The Open Racing Car Simulator, v1.3.5, 2013
9) D.Loiacono, J.Togelius, P.L.Lanzi: Car Racing Competition WCCI2008, Software Manual, Apr.2008
10) B.Wymann, http://www.berniw.org/tutorials/robot/
11) E.Onieva, D.A.Pelta, J.Alonso, V.Milanes, J.Perez: A Modular Parametric Architecture for the TORCS Racing Engine, 2009 IEEE Symposium on Computational Intelligence and Games
12) Tae Seong Kim, Joong Chae Na, Kyung Joong Kim: Optimization of an Autonomous Car Controller using a Self-Adaptive Evolutionary Strategy, International Journal of Advanced Robotic Systems 2012, Vol.9.73
13) Yehonatan Shichel, Moshe Sipper:GP-RARS: evolving controllers for the Robot Auto Racing Simulator. Mimetic Computing 3(2): 89-99 (2011)
14) Marc Ebner and Thorsten Tiede: Evolving Driving Controllers using Genetic Programming. in CIG'09 : Proceedings of the 5th International Conference on Computational Intelligence and Games, pages 279-286, Piscataway, NJ, USA, 2009, IEEE Press
15) Alexandru Becheru, Catalin Stoean: Optimization of Gear Changing using Simulated Annealing, Annals of the University of Craiova, Vol 39, No 2 (2012)
16) Extensible Markup Language (XML), https://www.w3.org/XML/

Paper sent to revision: 13.05.2015.
Paper ready for publication: 30.05.2016.