Wojciech KOZIŃSKI

Warsaw University of Technology, Institute of Control and Industrial Electronics

Application of Clojure Language to Build Multitasking Simulation Tool

***Abstract**. Simulation tool for various kinds of planar vehicles is presented. Each of vehicle is treated as separate object and executed in a thread. Such approach leads to multithreaded system, where concurrency rules have to be obeyed. The proper software tool has to be selected. One of candidates is Java language, with TimerTask objects or Executors, other one is newly developed language Clojure. The Clojure language is interpreted and executed by Java Virtual Machine and possesses many features, which are helpful to build multithreaded, concurrent systems.*

***Streszczenie.** Prezentowane jest narzędzie do symulowania ruchu pojazdów na płaszczyźnie. Każdy pojazd traktowany jest jako oddzielny obiekt i wykonywany w wątku. Takie podejście prowadzi do systemu wielowątkowego, w którym muszą być zachowane zasady budowy bezpiecznych programów współbieżnych. Istotnym jest wybór właściwego narzędzia. Jedną z możliwości jest wybór języka Java. Inną możliwością jest wybór języka Clojure wykonywanego przez wirtualną maszynę języka Java. (**Narzędzie do symulowania ruchu pojazdów na płaszczyźnie z wykorzystaniem języka Clojure**)*

Keywords: concurrency, planar vehicles movement, Java, Clojure.
Słowa kluczowe: współbieżność, ruch pojazdów na płaszczyźnie, język Java, język Clojure.

Introduction

Unmanned autonomous vehicles (UAVs), working together, are becoming more and more common in our world. One of such example is Dockland Light Railway in London, which trains operate without driver. Other, more sophisticated are used by military (drones, bomb disarming units), work in outer space (Mars rovers Spirit and Opportunity with Mars orbiting satellites) or work underwater. These vehicles are unmanned ones, operating either under remote control or in autonomous way. Each of such vehicles is equipped in multilayer software, which calculates the necessary commands to reach the planned target, optimizes the own resources, ensures safety of the vehicle and its environment in many different and conflicting situations. In these examples the main problem is, how to optimal control single UAV with many conflicting objectives.

The technological advances make communication between various kind vehicles easier and more reliable. There is a choice, either build complicated, multi-purpose, large, single vehicle or build large number of simpler, smaller vehicles which can work together exchanging information between themselves. The latter approach poses new problems of control such vehicles especially when the information from the environment and/or cooperating vehicles is limited by communication bandwidth or disturbances. The recent overview of control problems in formation of vehicles can be found in [1]. In many situations the group of cooperating vehicles can perform the same task as a single, complicated and more powerful vehicle in more efficient and reliable way.

The software framework to simulate behavior of a formation of unmanned autonomous vehicles is presented. It is built using Clojure language. The main aim of the product is to visualize and evaluate the performance of each vehicle equipped with certain control algorithm, moving with other similar or different vehicles in desired manner. The flow of information between vehicles can be structured and limited. The main idea of the tool is to model each vehicle as an independent task. Other task displays actual positions of vehicles with traces of their movement. The tasks read positions of other vehicles and write their own positions to global repository, which is organized as a special object. The repository should be thread safe [2], all data read or written should not be corrupted by possible, simultaneous actions of other tasks.

Basic assumptions and requirements of the framework are presented In the next section. Then some necessary facts from vehicle modeling, known control laws and collision avoidance rules are stated. Next basic information on Clojure language are given with examples of crucial structures. In the final section two examples using different vehicle models and different control laws are presented.

The functional requirements for simulation tool

As usual, for any software, it is necessary to specify basic functional requirements. The requirements are specified verbally, typical UML notation is not used. The tool under consideration is shaped like framework, where user can supply her/his own code to define various objects. Some typical objects can be included in the framework, so with minimal coding effort they can be used. With such very mild assumptions, the main functional requirements can be stated as follows:

- Possibility to use any of vehicle models, with any control algorithm and any obstacle avoidance rules,
- Vehicle model, control algorithm, rules for obstacle avoidance can be defined as algebraic formulas, differential equations or difference equations. Simple differential equation solver is provided.
- Message passing between vehicle models can be defined,
- Possibility to run many objects simultaneously (swarm, herd, flock) in multiprocessor (multi-core) environment. This requirement forces the user to define behavior of each vehicle model as separate task, which can be executed by single thread.

Structure of software should reflect limitations of real vehicles and coded control laws can be directly downloaded into laboratory equipment.

Common kinematic models and their control laws

In the framework, two basic kinematic models of the vehicle in two dimensional space are used. First one is described by three differential equations with trigonometric function.

Such model can be used to describe movements of ground vehicles, without specifying the vehicle dimensions. To make it more realistic the user can constrain the rate of turning. The vehicle kinematic model is the set three nonlinear equations with input saturation.

(1) $$\dot{x} = v\cos(\theta); \dot{y} = v\sin(\theta); \dot{\theta} = u; |u| \le u_{\max}$$

The states are: x and y, the Cartesian coordinates on the plane and heading angle θ, defined with the same coordinates. The control variables are: the speed v and derivative of turning angle u, which, here (1) is subject of saturation. The set of equations is nonlinear, with number of states less than number of controls, which makes control design difficult. The control action can be further limited. In case of plane flying with fixed altitude, the forward speed v can be set as constant. Some models of ground vehicles can move with constant speed forwards or backwards or stop. The unmanned underwater vehicle can move only forwards.

The second basic kinematic model consists of four differential equations, which are linear.

(2) $$\dot{x} = v_x; \dot{y} = v_y; \dot{v}_x = a_x; \dot{v}_y = a_y$$

where the states x and y define vehicle's location in Cartesian coordinates. Two control variables a_x and a_y are vehicle's accelerations along axis. As in the model (1) user can limit admissible values of control signals.

Typical control design is based on Lapunov function approach [3, 4], which guaranties stability, but not performance. To achieve satisfactory performance controller parameters have to be selected from admissible range. The equations below form the control law which can be applied to model (1).

(3) $$e_x = x_d - x;\ e_y = y_d - y; \theta_d = \arctan(e_y, e_x);$$

(4) $$e_\theta = \theta_d - \theta; D = \sqrt{e_x^2 + e_y^2}$$

(5) $$v = K\ D \cos(e_\theta);\ u = K_\theta e_\theta - \hat{\dot{\theta}}_d; \quad K, K_\theta > 0$$

The control objective is to steer the plant (1) to the desired position given by coordinates x_d and y_d. The equations (3) and (4) define the errors with respect to current position. These errors with control parameters K and K_Θ are used to calculate control signals v and u. Parameters K and K_Θ are to be selected by user from the range which guaranties stability

Recently the novel Cucker-Smale control algorithm is presented [5]. The algorithm is applied to the situation where large number of elements (flock, swarm) move simultaneously in common direction:

(6) $$\dot{x}_i(t) = v_i; \dot{v}_i(t) = \sum_{j=1}^{k} a_{ij}(x)\,(v_j - v_i);$$

(7) $$a_{ij}(x) = \frac{H}{\left(1 + \left\| x_i - x_j \right\|^2\right)^\beta}; \quad H > 0; \quad \beta \geq 0$$

where the position of each element of flock is described by (6). The velocity of each direction is calculated as special average defined by equations (6) and (7). The averaging coefficients cause the stronger influence of close neighbors to the distant ones. The desired model is obtained by selecting parameters H and β. The proof of stability of movement of the flock is given [5].

In movement of vehicles it is necessary to avoid collisions with stationary obstacles and the other vehicles. In most cases the potential function approach is used. When vehicle locates other object a repelling action is added to the control law. Such additional action should not cause instability and formal proof of stability is required.

For avoiding obstacles, potential function for model (1) with control (3-5) is proposed as in [4].

(8) $$d_a = \sqrt{\left(\frac{x_a - x}{\alpha}\right)^2 + \left(\frac{y_a - y}{\beta}\right)^2}; \quad \alpha, \beta > 0$$

(9) $$V_a = \left(\min\left\{0, \frac{d_a^2 - R^2}{d_a^2 - r^2}\right\}\right)^2; r > 0; R > 0; R > r$$

(10) $$\frac{\partial V_a}{\partial x} = 4\frac{\left(R^2 - r^2\right)\left(d_a^2 - R^2\right)}{\left(d_a^2 - r^2\right)^3}\left(x_a - x\right)$$

(11) $$\frac{\partial V_a}{\partial y} = 4\frac{\left(R^2 - r^2\right)\left(d_a^2 - R^2\right)}{\left(d_a^2 - r^2\right)^3}\left(y_a - y\right)$$

(12) $$E_x = e_x + \frac{\partial V_a}{\partial x}; E_y = e_y + \frac{\partial V_a}{\partial y}; \theta_d = \arctan\left(E_y, E_x\right)$$

where the set of equations (8-12) extends the control law (3-5) for vehicle described by (1). The centre of obstacle is located at coordinates x_a and y_a. The d_a value (8) calculates the distance of the vehicle to the obstacle with some weighting (parameters α and β). The parameter R denotes the distance when the obstacle is visible (located) and repelling force starts to act. The parameter r denotes the minimal, safe distance (radius) from the obstacle. The potential V_a function is defined in (9). The derivatives of the potential function over coordinates (10, 11) influence the errors (12) used later to calculate the control signals. The proof of stability with avoidance property is given in [4].

Similar, potential function approach to avoid collisions between elements of flock is presented [6] for model described by (6, 7):

(13) $$\dot{v}_i(t) = \sum_{j=1}^{k} a_{ij}(x)\,(v_j - v_i) + \Lambda(v)\sum_{j \neq i} f\left(\left\| x_i - x_j \right\|^2\right)\left(x_i - x_j\right)$$

(14) $$\Lambda(v) = \left(\frac{1}{k}\sum_{i>j}\left\| v_i - v_j \right\|^2\right)^{1/2}; f(r) = \frac{1}{(r - d_0)^\theta}; \theta > 1.$$

where the equation (13) is extended version of equation (6). Extension, with weighted term, serves as a potential, repelling action generated by other elements of the swarm. The formula (14) defines function $f(r)$, with d_0 as safe radius from another element. The weighting $\Lambda(v)$ depends on velocity difference of elements. If in the initial moment of movement, all elements are separated by at least safe distance, then by control law of equations (7, 13, 14) no collision will occur for the elements described by (6). Therefore the flock movement as a whole is safe, without collisions, with proof of stability [6].

Other simple, but effective way to avoid collision is to introduce common rule for all vehicles. The rule requires hard turn (maximal for given velocity) left (right), if there is other vehicle on the path to the goal. After the other vehicle is out of the line of sight, the vehicles continues its journey toward the goal.

The full geometrical analysis of collision conditions for two moving objects of non zero dimensions is given in [7].

Typical problems of building concurrent software

Computer hardware with operating system is ready to execute many tasks simultaneously. In case of one processing unit, time sharing principle can be used, in case of many processing units (cores), true parallel execution can take place. Appropriate, development software tools have to be used to build tasks and their interactions in correct way. In case of Java programming language,

authors of [2] present many potential problems of concurrent programming and suggest solutions. Here only some of them are mentioned, relevant to the considered simulation problem.

By design, in simulation problem, there are many tasks, each is responsible for movements (state evolution) of vehicle. The tasks should exchange information between themselves in order to be able to calculate control law with property of avoiding collisions. Each task is executed in separated thread. The simulations should run as fast as possible, so it is necessary to control execution order of threads by available processing units (cores).

In many simulations problems, there exist global variables. They usually serve as repositories of important information for all tasks having access to them. The stored information can be read, written or changed in asynchronous way. The special care is required to perform these operations in thread-safe manner.

One of special objects in Java programming are immutable objects. By definition these are objects, which state cannot be changed after proper creation, so they can be safely used by many threads. When another object of the same class, but with different state is required, the new object is created, instead of modifying the old one. Unused objects are removed from the memory by garbage collector of Java Virtual Machine.

In most cases the state variables of mutable object are logically interconnected. The typical example are 2D coordinates of vehicle location. Both must be updated in single operation, otherwise they can originate from different measurements. Such logical connection of states is called state invariants and must be kept through all time of life of mutable object.

Java language approach

The Java language [8], and its implementation with help of Java Virtual Machine (JVM) is one of strong candidates to develop concurrent software. Since first versions of Java there are commands and language constructions helpful in creating multitasking, concurrent software. Typical example are `Timer` and `TimerTask` objects which can help to define task, which can be executed repeatedly, at given time or after specified time. When user needs the execute many tasks, she/he is responsible to assign threads to execute created `TimerTask` type objects and control performance.

Since Java version 5, there is included powerful, concurrent library. The library has been extended in version 6 and very recently in version 7. Among concurrent utilities there are executors, which can help to organize execution of many tasks with limited number of threads. There are also thread-safe classes to construct global variables (repositories), which can be accessed by many threads. In many cases user can delegate responsibility of constructing thread-safe modules to ready-made library objects.

`TimerTask` objects and objects from Java concurrent library are used in [9-11] to build different simulation frameworks. There is presented, how the frameworks can be used to simulate movements of vehicles under different control laws in different environments.

Clojure language approach

The Java Virtual Machine (JVM) became well recognized standard in software industry. It is used as an interpreter for many interesting languages like Groovy, Ruby (JRuby), Python (Jython), Clojure and several others. The Clojure language [12-14] has been published in 2008 and gained some popularity. It poses several properties, which make it helpful in building concurrent software.

Clojure is functional programming language, as it avoids mutable state, uses recursion, enables to build higher-order functions. It is another dialect of Lisp programming language, Lisp notation is used. Clojure is dynamically typed language, user does not need to define types of objects. It poses full Java language interoperability, one can use all available Java libraries. Clojure language is concurrency oriented and has some distinct features related to concurrent programming. These features are exploited in discussed simulation framework.

By default all variables and data structures are immutable, so there is no need to protect them in multi-threaded environment. For mutable structures there are created special types of objects: `var`, `atom`, `ref` and `agent`. `Var` objects are designed as single valued, mutable objects used in thread-local context, `atom` objects are designed as single valued, mutable objects used in global context, `ref` objects are designed as multi-valued valued, mutable objects used in global context and `agent` objects are designed as single valued, mutable objects used in global context subject to asynchronous changes. Objects types `ref` and `agent` are updated by specially build-in Software Transactional Memory (STM).

To present flavor of Clojure few important examples are given. The first example of code is to calculate distance between two points, when their Cartesian coordinates *(x1 y1)* and *(x2 y2)* are given.

```
(defn distance2 [x1 y1 x2 y2]
  (+ (*(- x1 x2)(- x1 x2))(*(- y1 y2)(- y1 y2))))
(defn distance [x1 y1 x2 y2]
  (Math/sqrt (distance2 x1 y1 x2 y2)))
```

Function `distance2` calculates squared distance. Function `distance` calls function `distance2` and uses `sqrt` function from Java library package `java.lang.Math`.

The second example presents typical use of recursive Clojure function `reduce` to find minimal element in given vector `vec`. Function `min-vec` accepts vector of numbers `vec` and return minimal number found in the vector.

```
(defn min-vec [vec]
  (reduce (fn [x y] (if (< x y) x y))
     (first vec)
     (rest vec)))
```

The Clojure function `reduce` accepts three arguments: function which operates, initial value of result and list of next arguments for first argument function. The `if` clause returns the smaller value of value of `x` and first value taken from rest of the vector, denoted as `y`.

The next example is essential for presented simulation framework. It defines object `vehicle`, which is declared as an `agent`. Each `vehicle` has declared: its state variables, non-linear functions describing evolution of state for given time (set of ordinary differential equations), control law and sampling time.

```
(defn veh-behave [who];;behavior of vehicle agent
  (let [samp (get-in @vehicles [who :sampling])]
  (when @running (send-off *agent* #'veh-behave))
  (veh-one-step who (/ samp 1000) 0)
  (. Thread (sleep samp))
  who
  ))
```

The `veh-behave` function accepts one argument named who, which is `agent`. If framework is in operating state (running), the recent `agent` is activated by send-off command. Then `veh-one-step` function is called which

calculates and changes the state of vehicle. Next for `samp` period of time agent is inactive (by Java `sleep` function).

The agent objects describing vehicles are stored in an array `vehs`. They are activated by Clojure command `dorun` which activates each vehicle by mapping it with `veh-behave` function and executing `send-off` function.

```
(dorun (map #(send-off % veh-behave) vehs))
```

The examples illustrate how powerful is Lisp syntax used by Clojure. Each command can invoke complicated actions.

The simulation framework is coded in Clojure. Many functions from Java library packages are used, especially to plot lines or perform calculations. The code is compact with clear, easy to understand structure. Main function calls sub-functions, these call sub-sub-functions. It is possible to build local binding where the results of two or more functions are gathered are locally stored to be used as parameters for next function.

Examples

The simulation framework is used to present some examples of vehicles moving under various control laws and avoiding collisions between themselves or other stationary obstacles.

In the first example four vehicles (denoted as planes) move on 2D plane. Each of them is described by equation (1) with control law (3)-(5). One vehicle has its own target, three other should tend to the same target. Shortest route, straight line, for each vehicle is depicted by the dashed line. In the middle of simulation area there is a circular no entry zone. Vehicles are equipped with simple avoidance control of banking left, when they observe obstacle on their path (the maximal sensing rage is specified). When there is no obstacle, they move on shortest path to target. The target for three vehicles is the same., so it can be reached by only one, the other are pushed away, to keep safety distance.

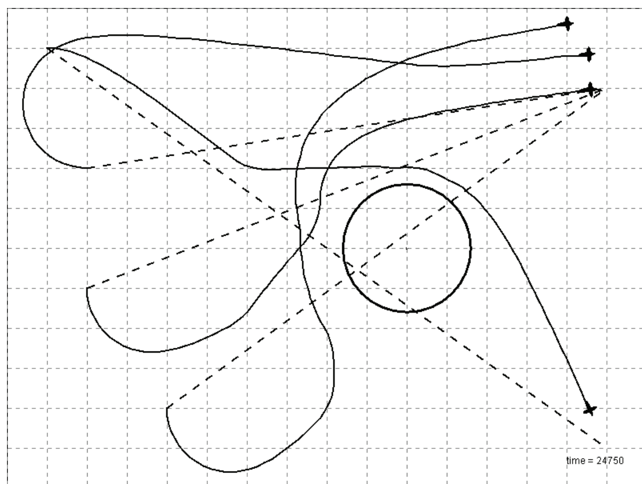

Fig.1. Movement of four vehicles with obstacle avoiding

In the Figure 1 traces of movement of four vehicles are presented. All vehicles encircled fixed obstacle by making left turns (roundabout principle). The vehicles move forward with steady velocity. Their rate of turn is limited. Initial direction of vehicles are different from the direction to the target, so at first they have to make a turn. There is limited range, when each vehicle can notice the obstacle. If the distance to obstacle is less then sensing range, avoiding collision function is activated. Similar result is obtained when all vehicles are equipped with obstacle avoidance control law of (8)-(12) equations.

The second experiment involves four vehicles, which are simulated as a flock using Cucker-Smale model (6)-(7) with collision avoidance (13-14) feature. Their starting position are well separated, with kept minimal distance between each other. Their directions of movement lead towards collision as indicated by the dashed lines. Each vehicle has different initial velocity.

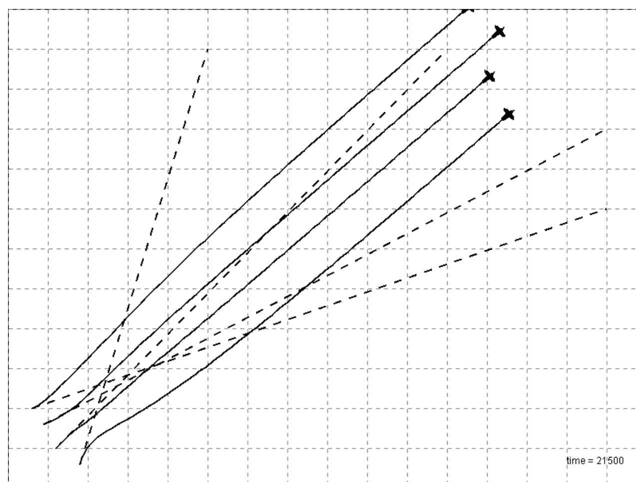

Fig. 2. Movement of four vehicles as a flock with Cucker-Smale law

In the Figure 2 traces of movement of four vehicles as a flock are presented. The actions of control law cause that common direction and velocity is established with keeping safe distances between members of the flock.

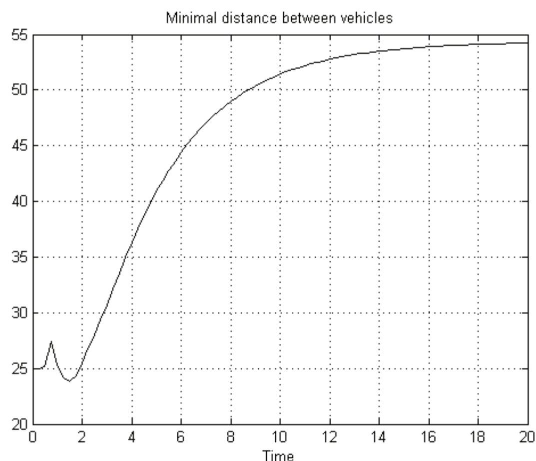

Fig.3. Minimal distances between vehicles from flock simulation

In the Figure 3 the minimal distance between flock member is presented. The minimal safe distance is set to 20 units (d_0 in equation (14)). At starting time the minimal distance is satisfied, so according to [6], it should be at least kept.

In the Figure 4 and Figure 5, the velocities of vehicles in axis x and y are presented. The velocities at starting time are different. As time flows and flock moves, the velocities of all vehicles are unified, so the flock moves in the same direction, with the same velocity.

The plots presented in the Figures 1 and 2 are generated from Clojure code, after finishing the simulation run. The calculations for minimal distance and velocities in x, y axis, plotted in the Figures 3, 4 and 5 are done with help of Matlab® from collected data from simulation run.

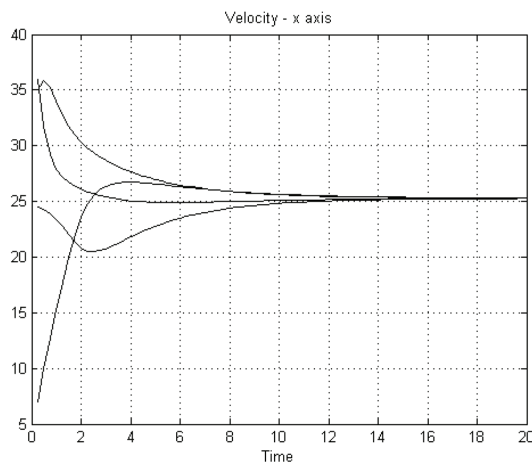

Fig.4. Velocities of vehicles in x-axis in flock simulation

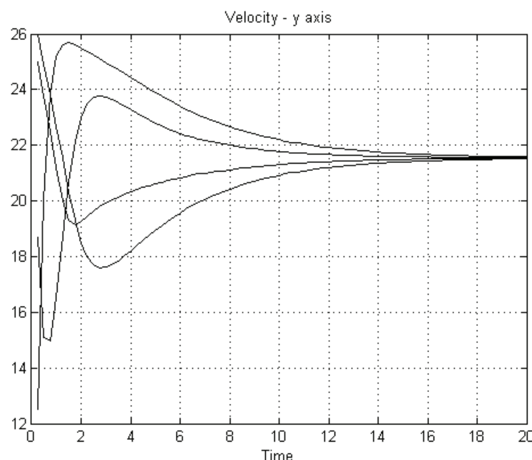

Fig.5. Velocities of vehicles in y-axis in flock simulation

Conclusions

Some considerations how to build framework for simulation of vehicles with different control laws are presented. In previous attempts [9-11] various Java approaches were used, which required considerably attention to build concurrent software as thread-safe one. Clojure language was created with hope to simplify concurrency coding, which is confirmed by presented approach. Clojure is functional language, using Lisp prefix notation. Software build with Clojure poses very clear structure, is relatively easy to test. It uses Java Virtual Machine (JVM), which is recognized as industry standard, to interpret and execute code. All Java libraries, accessible by JVM, can be used in Clojure programs. It is hoped, that in some future, software written in functional languages can be automatically distributed between processing units (cores) making parallel processing easier. Therefore in some situations Clojure can successfully replace Java.

REFERENCES

[1] Murray R.M. Recent research in cooperative control of multi-vehicle systems. ASME Journal of Dynamics Systems, Measurement and Control. Vol.129, September 2007, issue 5.

[2] Goetz B. et al. Java Concurrency in Practice. Addison Wesley 2006.

[3] Aicardi M., Casalino G., Bicchi A., Balestrino A. Closed Loop Steering of Unicycle-like Vehicles via Lapunov Techniques. IEEE Robotics and Automation Magazine. March 1995.

[4] Mastellone S., Stipanovic D.M., Spong M.W. Remote Formation Control and Collision Avoidance for Multi-Agent Nonholonomic Systems. 2007 IEEE International Conference on Robotics and Automation, Roma, Italy 10-14 April 2007, pp.1062-1067.

[5] Cucker F.,Smale S., Emergent Behavior In Flocks. IEEE Transactions on Automatic Control, vol. 52, no. 5, pp.852-862. May 2007.

[6] Cucker F., Dong J-G. Avoiding Collisions in Flocks. IEEE Transactions on Automatic Control, vol. 55, no. 5, pp.1238-1243. May 2010.

[7] Chakravarthy A., Ghose D. Obstacle Avoidance in Dynamic Environment: A Collision Cone Approach. IEEE Transactions on Systems, Man and Cybernetics – Part A: Systems and Humans, Vol. 28, no. 5, September 1998, pp.562-574.

[8] Java documentation and tutorial J2SE6, J2SE7. www.oracle.com/technetwork/java/index.html

[9] Kozinski W., Narzędzie do symulacji zespołów (formacji) pojazdów autonomicznych. XI Krajowa Konferencja Inżynierii Oprogramowania 2009. Pułtusk wrzesień 2009 (In Polish)

[10] Koziński W., The Framework for Simulation of Formation of Unmanned Autonomous Vehicles (UAVs). Przegląd Elektrotechniczny nr 9, 2010.

[11] Koziński W., Porównanie dwóch metod symulacji zadań współbieżnych. XII Krajowa Konferencja Inżynierii Oprogramowania 2010. Gdańsk wrzesień 2010 (In Polish)

[12] Clojure page and documentation. www.clojure.org, www.clojuredocs.org

[13] Vanderhart L.,Sierra S., Practical Clojure.Apress 2010

[14] Volkmann R.M. Clojure – Functional Programming for JVM. java.ociweb.com/mark/clojure/article.html

Author: *dr inż. Wojciech Koziński,* Warsaw University of Technology, Institute of Control and Industrial Electronics*, pl. Politechniki 1 00-662 Warszawa, E-mail: wojciech.kozinski@ee.pw.edu.pl*.