APPLICATION OF DEECO FRAMEWORK TO MDVRPTW PROBLEM

Artioma Merabiani
International Black Sea University, Tbilisi, Georgia
bartender318@gmail.com

Summary

In the paper the concept of Autonomic Components Ensembles (ACE), applied to the real-world Multi-Depots Vehicle Routing Planning with Time Windows (MDVRPTW), is proposed. Each vehicle is associated with the corresponding autonomic component AC (a virtual machine in datacenter) and exchange on-line information with other vehicles. Besides, ACs can reschedule routes in order to find the acceptable alternative routes that enable vehicles to meet time windows requirements and, at the same time, avoid the congested roads. Implementation of DEECo (Distributed Emergent Ensembles of Components) model to create dynamic ensembles of vehicles and non-congested route segments is also proposed in the paper. Detailed description of components, components' knowledge, processes and interfaces is given.

Keywords: Multi Depots. Vehicles. Routes planning. Time Windows. Autonomic component. Datacenter. Virtual machines.

1. Introduction

In [1] we described the adaptive algorithm to solve Multi Depots Vehicle Routing Planning with Time Windows (MDVRPTW) problem. The algorithm is aimed to account for realistic real-world situation, such as presence of various congestion types. The congestions are the most important critical factors for the successful and practically acceptable solution of the MDVRPTW problem.]. Since traffic congestion cause heavy delays, it is very costly for intensive road users such as logistic service providers and distribution firms. In particular, such delays cause large costs for hiring the truck drivers and the use of extra vehicles, and if they are not accounted for in the vehicle route plans they may cause late arrivals at customers or even violations of driving hour's regulations. Therefore, accounting for traffic congestion has a large potential for cost savings. We have developed a modification of the ALNS algorithm [2] (written in the *Jsprit* framework). Namely, our algorithm takes into account a probability of links' congestion, estimation of probability of their release of busy route sections. Our modification of the algorithm can plan routes for any starting and finishing nodes.

To provide the real-time adaptability the proposed approach uses the concept of *autonomic components (AC) and autonomic component ensembles (ACE)*[1]. Each vehicle is associated with the corresponding autonomic component AC (implemented as a virtual machine in datacenter) and exchange on-line information with other vehicles. This allows a vehicle to notify other vehicles about expected and actual congestion. Besides, ACs can reschedule routes in order to find the acceptable alternative routes that enable vehicles to meet time windows requirements and, at the same time, avoid the congested roads. It is necessary to point out that the algorithm of adaptation is able to reschedule and find alternative routed for several vehicles in parallel. The latter significantly increases the performance of proposed approach.

ACs are entities with dedicated knowledge units and resources that can cooperate while playing different roles. ACs are dynamically organized into ACEs. AC members of an ACE are connected by the interdependency relations defined through predicates (used to specify the targets of communication actions. The functional description of an AC and ACE is shown on Fig.1.

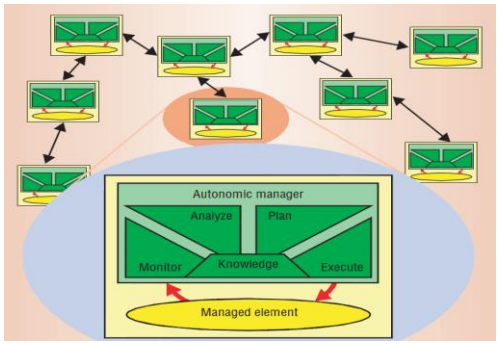

Fig.1 Functional description of a component

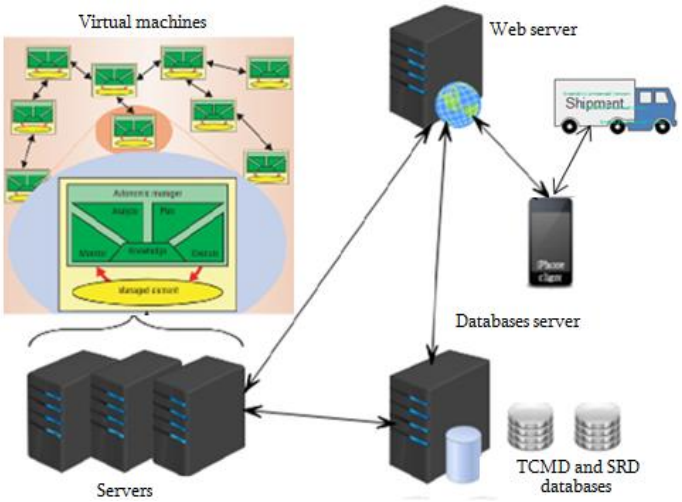

Fig.2 . General infrastructure

The ACs in an ACE may be implemented as *virtual machines (VMs)* in *datacenters (ACE)*. Each AC is associated with the concrete vehicle and comprehensive information of the current location of the vehicle on the route, relevant data on its current state and etc. In our approach the knowledge repository is used to store these data and exchange them with other ACs. Occasionally so called *spatial-temporal* event (that is, a vehicle arrives to a certain service point at a certain time) occurs. The equipment in the car (GPS receivers and GSM telephones (or some similar wireless communications technology)) determines location using the GPS receiver and sends the coordinates and other relevant data to the Web server. The general infrastructure of our approach id shown in the Fig.2.

The ***base*** virtual machine VM_0 hosts all main structural components of proposed system: JSpirit, MatSim, travel and congestion management database (TCMD), database of simulation results (SRD), web servers for connection with vehicles, GPS, etc. Although VM_0 is permanently used and maintained, it is convenient to represent it as a virtual machine because it will intensively interact and exchange data with other virtual machines. each of which represents autonomic component (AC). As it will be shown later, autonomic components are associated with concrete vehicles and constitute an Autonomic Component Ensemble (ACE). The base VM_0 executes the initial solution of MDVRPTW problem and generates the initial set of routes RI. The input parameters, such as time windows for each service points, are held at VM_0 as well. After generating the initial set of routes, new virtual machines, enumerated from 1 to ***nr*** (where nr is the amount of routes in the initial set RI), are created. The recourses of the datacenter's servers are dynamically allocated to virtual machines.

In this paper we describe the usage of DEECo (Distributed Emergent Ensembles of Components) component model [3] and its framework. This framework is applied to the MDVRPTW's case. The detailed description of the framework is given below.

2. DEECO general concepts applied to the MDVRPTW problem

DEECo is built on top of two first-class concepts: component and ensemble [3]. A component is an independent and self-sustained unit of development, deployment and computation. An ensemble acts

as a dynamic binding mechanism, which links a set of components together and manages their interaction. A grounding idea in DEECo is that the only way components bind and communicate with one another is through ensembles. The two first-class DEECo concepts are in

detail elaborated below. An integral part of the component model is also the runtime framework providing the necessary management services for both components and ensembles.

A *component* in DEECo comprises *knowledge*, exposed via a set of *interfaces*, and *processes*[3]. Knowledge reflects the state and available functionality of the component (lines 6-19). It is organized as a hierarchical data structure, which maps knowledge identifiers to values. Specifically, values may be either potentially structured data or executable functions. In this context, the term *belief* refers to the part of a component's knowledge that represents a copy of knowledge of another component, and is thus treated with a certain level of uncertainty as it might become obsolete or invalid.

A component's knowledge [3] is exposed to the other components and environment via a set of interfaces (lines 5, 60). An interface (e.g., lines 1-2) thus represents a partial view on the component's knowledge. Specifically, interfaces of a single component can overlap and multiple components can provide the same interface, thus allowing for polymorphism of components.

Component processes are essentially soft real-time tasks that manipulate the knowledge of the component. A process is characterized as a function (lines 23-27) associated with a list of input and output knowledge fields (line 21,22). Operation of the process is managed by the runtime framework and consists of atomically retrieving all input knowledge fields, computing the process function, and atomically writing all output knowledge fields [3].

Being active entities of computation implementing feedback loops, component processes are subject to cyclic scheduling, which is again managed by the runtime framework [3]. A process can be scheduled either periodically (line 74), i.e., repeatedly executed once within a given period, or as triggered (line 28), i.e., executed when a trigger condition is met.

```
1. interface RouteSegmentsCongestionAware:
2.      initialSP, routeSegment, congestionStatus,  expectedCongestionInducedDelay
3. interface RouteSegmentAvailabilityAggregator:
4.           position, timetable, routeSegmentsAvailability
5. component  Vehicle  features RouteSegmentAvailabilityAggregator:
6.       knowledge:
7.            position = GPS(…),
8.            currentSP=(position, …),
9.            routeSegmentsAvailability=List<segmentsStatus>
10.           timetable = List<TimeWindowsForSPs>,
11.               route = {
12.                           List<SPs>,
13.                           onSchedule=TR
14.                            isFeasible=TRUE
15.                       },
16.             expectedCongestionInducedDelay=(…),
17.             vehicleParameters=List<Parameters>,
18.              costDriverWaitPayment=(….),
19.              costViolationTimeWindows=(….)
20.       process computeNewRoute:
21.                in routeSegmentsAvailability, in timetable,
22.                inout   route
23.               function:
24.                      if (!route.isFeasible ∧ (costDriverWaitPayment
```

```
                    >costViolationTimeWindows))
            route ← ME.ALNS.computeRoute (position, timetable,
                                          routeSegmentsAvailability)
        scheduling: periodic(2000)
process checkRouteFeasibility:
        in route, in position, in timetable, in routeSegmentsAvailabilities,
        out route.isFeasible
      function:
        route.isFeasible ← ME.checkRouteFeasibility (route, position, timetable,
                                                     routeSegmentsAvailabilities)
      scheduling: triggered(changed(routeSegmentsAvailabilities) ∨
                                    changed(onSchedule))
  process computeCostDriverWaitPayment:
        in routeSegment,
        in CongestionInducedDelay,
        in vehicleParameters,
        out CostDriverWaitPayment
      function:
        CostDriverWaitPayment← ME.computeCostDriverWaitPayment(routeSegment,
                                vehicleParemeters, CongestionInducedDelay)
      scheduling: triggered(changed(changed(routeGenerated.isFeasible) ∨
                        changed(onSchedule) ∨
                            changed(routeSegmentsAvailabilities) )
   process computeCostViolationTimeWindows:
        in routeSegment,
        in CongestionInducedDelay,
        in vehicleParameters,
        out costViolationTimeWindows
      function:
        costViolationTimeWindows ←
            ME.computeCostViolationTimeWindows(routeSegment,
                CongestionInducedDelay, vehicleParameters)
      scheduling: triggered(changed(changed(routeGenerated.isFeasible) ∨
                        changed(onSchedule) ∨
                            changed(routeSegmentsAvailabilities) )
component RouteSegmentsCongestion features RouteSegmentsCongestionAware:
  knowledge:
      initialSP=(…),
      endSPs =List<adjacentSPs>,
      routeSegment =(initialSP, endSP ∈ endSPs),
      segmentAvailability=(…),
      congestionStatus=[congestionStatus, type, startingTime,
                expectedCongestionClearanceTime,
                congestionClearanceProbability],
       expectedCongestionInducedDelay=(….)
```

```
70.     process observeSegmentAvailability:
71.         out  segmentAvailability
72.       function:
73.          segmentAvailability ← MessageFromVehicle.getSegmentCurrentAvailability
74.       scheduling: periodic(1000)
75.     process computeCongestionInducedDelay:
76.         in routeSegment,
77.         in congestionDuration, in segmentNonCongestedCapacity,
78.         in segmentCongestedCapacity, in arrivalRate,
79.         out CongestionInducedDelay
80.       function:
81.          CongestionInducedDelay ←
82.                ME.computeCongestionInducedDelay(routeSegment,
83.                congestionDuration, segmentNonCongestedCapacity,
84.                segmentCongestedCapacity)
85.         scheduling: triggered(changed(congestionStatus) )
```

Referring to the MDVRPTW running example, the components (each occurring in multiple instances) are the *Vehicle* and the *RouteSegmentsCongestion*. A Vehicle maintains a belief over the availability of the relevant *RouteSegmentsCongestion* (routeSegmentsAvailability, line 9). It uses a Adaptive Large Neighborhood Search (ALNS) library to (re-) compute its route according to the availability belief

and its timetable (lines 20-28) every time the availability belief or route feasibility changes (line 28). The Vehicle also checks if its route remains feasible, with respect to the corresponding *routeSegmentsAvailabilities* and its route's *onSchedule* propertycurrent position (lines 29-36). A *RouteSegmentsCongestion* just keeps track of its available route's segment availability and computes

the expected Congestion Induced Delay time (lines 60-85).

An *ensemble* (see the description below) implements a dynamic binding among a set of components and thus determines their composition and interaction [3]. In DEECo, composition is flat, expressed implicitly via a dynamic involvement in an ensemble. Among the components involved in an ensemble, one always plays the role of the ensemble's coordinator while others play the role of the members. This is determined dynamically (the task of the runtime framework) according to the membership condition of the ensemble.

```
1.  ensemble UpdateRouteSegmentAvailabilityInformation
2.     coordinator: RouteSegmentAvailabilityAggregator
3.     member: RouteSegmentsCongestionAware
4.     membership:
5.          ∃ vehicle  ∈ coordinator. routeSegmentsAvailability:
6.                    isAvailable(member.routeSegmentsAvailability)==TRUE
7.     knowledge exchange:
8.          coordinator: routeSegmentsAvailability ← member. routeSegmentsAvailability
9.          coordinator: expectedCongestionInducedDelay ← member.
```

```
10.                    expectedCongestionInducedDelay
11.    scheduling: periodic(2000)
```

As to interaction, the individual components in an ensemble are not capable of explicit communication with the others [3]. Instead, the interaction among the components forming the ensemble takes the form of *knowledge exchange.* Specifically, definition of an ensemble consists of:

- *Membership* condition. Definition of a membership condition includes the definition of the interface specific for the *coordinator* role – coordinator interface (line 2), as well as the interface specific for the member role (and thus featured by each member component) – member interface (line 3), and the definition of a *membership predicate* (lines 4-6). A membership predicate declaratively expresses the condition under which two components represent a coordinator-member pair of the associated ensemble. The predicate is defined upon the knowledge exposed via the coordinator/member interfaces and is evaluated by the runtime framework when necessary.
- *Knowledge exchange*. Knowledge exchange embodies the interaction between the coordinator and all the members of the ensemble (lines 7-8); i.e., it is a one-to-many interaction (in contrast to the one-to-one form of the membership predicate). Being limited to coordinator-member interaction, knowledge exchange allows the coordinator to apply various interaction policies. In principle, knowledge exchange is carried out by the runtime framework; thus, it is up to the runtime framework when/how often it is performed. Similarly, to component processes, knowledge exchange can be carried out either periodically or when triggered (line 11). Based on the ensemble definition, a new ensemble is dynamically formed for each group of components that together satisfy the membership condition.

The only ensemble of the running example is the *UpdateRouteSegmentAvailability-Information* ensemble. Its purpose is to aggregate the route segments availability information of the members, i.e. RouteSegmentsCongestions, on the side of the coordinator, i.e., Vehicle (lines 9-10). The ensemble is formed only when a route segment is available and the expected congestion induced delay time is acceptable.

3. jDEECo run-time realization of MDVRPTW problem [3,4]

By building on Java annotations, the mapping of DEECo concepts relies on standard Java language primitives and does not require any language extensions or external tools [3].

An examples of an *component* definition has the form of a Java class:

```
1.  @DEECoComponent
2.  public class Vehicle extends ComponentKnowledge {
3.      public Position position;
4.      public ServicePoint currentSP
5.      public List< TimeWindowsForSPs >  timetable;
6.      public Map<ID, segmentsStatus >  routeSegmentsAvailability
7.      public  Route  route;
8.      public  Delay expectedCongestionInducedDelay;
9.      public   List <vehicleParameters>  vehicleParameters
10.     public   Cost  costDriverWaitPayment,
11.     public   Cost  costViolationTimeWindows
12.  public Vehicle() {
13.      // initialize the initial knowledge structure reflected by the class fields
```

```
14.     }
15. @DEECoProcess
16. public static void  computeNewRoute(
17.         @DEECoIn("routeSegmentsAvailability ") @DEECoTriggered Map<…>
18.                                                 routeSegmentsAvailability
19.         @DEECoIn("timetable") List< TimeWindowsForSPs > timetable,
20.         @DEECoInOut("route") Route   route
21.      ) {
22.            // re---compute the vehicle's route  if it's infeasible
23.     }
24. @DEECoProcess
25. @DEECoPeriodScheduling(2000))
26. public static void checkRouteFeasibility (
27.             @DEECoIn("route") Route route,
28.             @DEECoIn("timetablel") List< TimeWindowsForSPs >   timetable,
29.             @DEECoIn("position") Position position,
30.             @DEECoOut("route.isFeasible") OutWrapper<Boolean> isRouteFeasible
31              ){
32.                     // determine feasibility of the route
33.              }
34                 . ...
35.         }
```

A component definition has the form of a Java class (see the above code). Such a class is marked by the @DEECoComponent annotation and extends the ComponentKnowledge class. The initial knowledge structure of the component is captured by means of the public, non-static fields of the class (lines 3-11). At runtime, this initial knowledge structure is initialized either via static initializers or via the constructor of the class (lines 12-14). The component processes are defined as public static methods of the class, annotated with @DEECoProcess (e.g., lines 15-23).

The input and output knowledge of the process is represented by the methods' parameters.

The parameters are marked with one of the annotations @DEECoIn, @DEECoOut or @DEECoInOut, in order to distinguish between input and output knowledge fields of the process (e.g., lines 17-20). Each annotation also includes an identifier of the knowledge field that the associated method parameter represents. When a non-structured knowledge field constitutes an inout/out knowledge of a process, the associated method parameter is for technical reasons (related to Java immutable types) passed inside an OutWrapper object (e.g., line 30). Periodic scheduling of a process is defined via the @DEECoPeriodicScheduling annotation of the process's method, which takes the period expressed in milliseconds in its parameter (line 25). Triggered scheduling is defined via @DEECoTriggered annotation of the method's parameter, change of which should trigger the execution of the process (lines 17-19).

Below the example of an ensemble definition Java (jDEECO) is given:

```
1. @DEECoEnsemble
2. @DEECoPeriodicScheduling(2000)
3. public  class UpdateRouteSegmentAvailabilityInformation extends Ensemble {
4.
5.         @DEECoEnsembleMembership
6.         public static Boolean membership (
```

```
7.              @DEECoIn("coordinator.routeSegmentsAvailability ")
                            List< segmentsStatus>,
8.              @DEECoIn("member.routeSegmentsAvailability ")  SegmetStatus,
9.               @DEECoIn("member. expectedCongestionInducedDelay ") Delay
10.         ) {
11.              for (RouteSegment rs : segmentRoute)) {
12.                  if (isAvailable(rs.routeSegmentsAvailability)==TRUE
13.               return true;
14.              }
15.           return false;
16.    }
17.
18.
19.  @DEECoEnsembleKnowledgeExchange
 20. @DEECoPeriodScheduling(2000))
21. public static void knowledgeExchange (
22.     @DEECoOut("coordinator. routeSegmentsAvailability ") Map <…> SegmentStatus,
23.     @DEECoOut("coordinator. expectedCongestionInducedDelay ")  Delay,
24.     @DEECoIn("member. routeSegmentsAvailability]") Map <…> SegmentStatus,
25.     @DEECoIn("member. expectedCongestionInducedDelay "") Delay,,
26.    )
27.  }
```

The ensemble definition takes also the form of a Java class [3]. In particular, the class is marked with the @DEECoEnsemble annotation and extends the Ensemble class (see the above example). Both the membership predicate and the knowledge exchange are defined as specifically-annotated static methods of this class. While the method representing the membership predicate is annotated by @DEECoEnsembleMembership (line 5), the method representing knowledge exchange is annotated by @DEECoEnsembleKnowledgeExchange (line 19).

The jDEECo runtime framework is primarily responsible for scheduling component processes, forming ensembles, and performing knowledge exchange. It also allows for distribution of Components [3].

As illustrated in Figure 3, it is internally composed of the management part and the knowledge repository. The management part is further composed of two modules. One is responsible for scheduling and execution of component processes and knowledge exchange of ensembles. The other is responsible for managing access to the knowledge repository. Exploiting the fact that all modules of the runtime framework implementation are loosely coupled, we are able to introduce implementation variants for each of them. As a result, different variants can be selected in order to reflect specific requirements imposed to the platform [3].

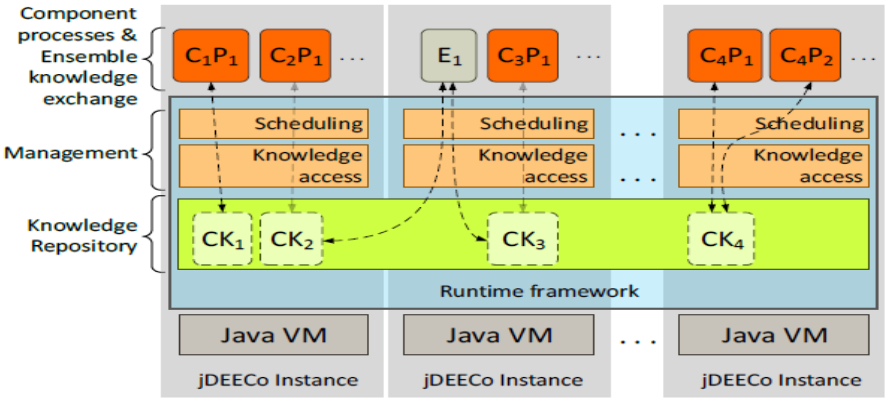

Fig. 3: jDEECo runtime framework architecture

The role of the knowledge repository is to store the component's knowledge (e.g., CK1 – knowledge of component C1 – in Figure 3). Its responsibility is also to provide component processes and knowledge exchange of ensembles with access to this knowledge. In fact, we provide a local and a distributed implementation of the knowledge repository; the former is employed for simulation and verification of the code) while the latter is used in case the runtime framework needs to run in a distributed setting (i.e., the distribution is achieved at the level of knowledge repository). Specifically, the distributed implementation of the knowledge repository allows each component to run in a different Java virtual machine (in Figure 3).

The approach described above was implemented by using cloud computing service provider *Google Cloud Platform*. Namely, *IaaS* (Infrastructure-as-a-Service) was used for creation and deployment Virtual Machines (VM), associated with the vehicles (totally 17 VMs) and the Virtual Machine, associated with the **base** Virtual machine VM_0 [1]. The VM_0 hosts all main structural components of proposed system: JSpirit, MatSim, ALNS, travel and congestion management database (TCMD), database of simulation results (SRD), web servers for connection with vehicles, GPS, etc. VMs, associated with vehicles, run local reduced copies of ALNS algorithm, and local copy of TCMD and SRD databases [1]. Payments Pay-as-you-go for consumed resources of the Google Cloud Platform datacenter are on average 60% less for many compute workloads than other clouds. Implementation of Autonomic Components Ensembles (ACE) on Google Cloud Platform (and, in general, on other cloudproviders platforms) shifts most of the costs from *capital expenditures* (or buying and installing servers, storage, networking, and related infrastructure) to an *operating expenses* model, where customers pay only for usage of these types of resources.

References:

1. Prangishvili A., Rodonaia I., Shonia O., Merabian A. (2017). Adaptive real-world algorithm of solving MDVRPTW (Multi Depots Vehicle Routing Planning with Time Windows) problem, *International Journal of Transportation Systems*, is. 2, 1-6

2. Rodonaia I., Merabian A.. (2016). Real-world applications of the vehicle routing problem in Georgia. *Journal Technical Science & Technologies (JTST),* is. (2) 41 -44 November, Tbilisi.

3. Bures T., Gerostathopoulos I., Hnetynka P., Keznikl J., Kit M., Plasil F. (2014). DEECo– an Ensemble-Based Component System. Charles University in Prague, Faculty of Mathematics and Physics, Prague, Czech Republic.

4. D3S. Charles University in Prague. jDEECo website. Accessed April 17, 2013. https://github.com/d3scomp/JDEECo, 2013.

DEECO გარემოს გამოყენება მრავალდეპოიანი დროის ფანჯრების მქონე შეზღუდვების სატრანსპორტო მარშრუტიზაციის დაგეგმარების (MDVRPTW) ამოცანისათვის

არტიომა მერაბიანი
შავის ზღვის საერთაშორისო უნივერსიტეტი, თბილისი

რეზიუმე

შემოთავაზებულია ავტონომიური კომპონენტებისგან შემდგარი ანსამბლის (ACE) კონცეპციის გამოყენება რეალურ პირობებში მრავალ დეპოიანი დროის ფანჯრების შეზღუდვების მქონე სატრანსპორტო მარშრუტიზაციის დაგეგმარების (MDVRPTW) ამოცანისთვის. თითოეული მანქანა ასოცირებულია შესაბამის ავტონომიურ კომპონენტთან AC (განლაგებული მონაცემთა დამუშავების ცენტრის ვირტუალურ მანქანაზე) და ოპერატიულად (on-line) ცვლის ინფორმაციას სხვა მანქანებთან. გარდა ამისა, AC ხელახლა გეგმავს მარშრუტებს, რათა მიიღოს ალტერნატიული შედეგი, რომელიც დააკმაყოფილებს დროის ფანჯრების მოთხოვნებს, და, ამავე დროს, მოახდინს გაუვალი გზების მონაკვეთების შემოვლას. შემოთავაზებულია DEECo (განაწილებული საგანგებო კომპონენტების ანსამბლები) მოდელის განხორციელება გაუვალი ან გადატვირთული მარშრუტის მონაკვეთების და მანქანების დინამიკური ანსამბლების შესაქმნელად. ნაშრომში მოცემულია კომპონენტების, ცოდნის ბაზების, პროცესების და ინტერფეისების დეტალური აღწერა.

ПРИМЕНЕНИЕ СРЕДЫ DEECO ДЛЯ ЗАДАЧИ ПЛАНИРОВАНИЯ МУЛЬТИ-ГАРАЖНЫХ АВТОМОБИЛЬНЫХ МАРШРУТОВ СО ВРЕМЕННЫМИ ОГРАНИЧЕНИЯМИ (MDVRPTW)

Мерабян А.
Международный Университет Черного моря, Тбилиси, Грузия

Резюме

Предложен метод применения концепции Ансамблей Автономных Компонентов (ACE) для проблемы планирования мульти-гаражных автомобильных маршрутов со временными ограничениями (MDVRPTW) в реалистических условиях. Каждый автомобиль ассоциирован с соответствующим автономным компонентом АС (представленным виртуальной машиной в центре обработки данных) и оперативно обменивается информацией с другими автомобилями. Кроме того, автономные компоненты могут повторно производить планирование маршрутов для нахождения приемлемых альтернативных путей, которые позволят удовлетворять временные ограничения и, в то же время, обходить непроходимые участки маршрутов. Предложена реализация платформы моделей DEECo (Распределенные Чрезвычайные Ансамбли Компонентов) для создания динамических ансамблей автомобилей и незаторенных участков маршрутов. Дано детальное описание компонентов, базы знаний, процессов и интерфейсов.