BULLETIN OF THE POLISH ACADEMY OF SCIENCES
TECHNICAL SCIENCES, Vol. 63, No. 3, 2015
DOI: 10.1515/bpasts-2015-0078

Control system architecture for the investigation of motion control algorithms on an example of the mobile platform Rex

M. JANIAK[1*] and C. ZIELIŃSKI[2]

[1] Chair of Cybernetics and Robotic, Faculty of Electronics, Wrocław University of Technology, 11/17 Z. Janiszewskiego St., 50-320 Wrocław, Poland
[2] Institute of Control and Computation Engineering, Faculty of Electronics and Information Technology, Warsaw University of Technology, 15/19 Nowowiejska St., 00-665 Warsaw, Poland

Abstract. This paper presets the specification and implementation of the control system of the mobile platform *Rex*. The presented system structure and the description of its functioning result from the application of a formal method of designing such systems. This formalism is based on the concept of an embodied agent. The behaviours of its subsystems are specified in terms of transition functions that compute, out of the variables contained in the internal memory and the input buffers, the values that are inserted into the output buffers and the internal memory. The transition functions are the parameters of elementary actions, which in turn are used in behaviour patterns which are the building blocks of the subsystems of the designed control system. *Rex* is a skid steering platform, with four independently actuated wheels. It is represented by a single agent that implements the locomotion functionality. The agent consists of a control subsystem, a virtual effector and a virtual receptor. Each of those subsystems is discussed in details. Both the data structures and the transition functions defining their behaviours are described. The locomotion agent is a part of the control system of the autonomous exploration and rescue robot developed within the *RobREx* project.

Key words: mobile robot control, robot control architecture, robot control system specification.

1. Introduction

Design of a complex systems requires both the design of controllers for its subsystems and design of the overall system controller. Both parts of the design are interrelated. The design can only be treated as adequate if the subsystems can be integrated into the overall system in such a way that it executes the allotted tasks satisfactorily. In the case of research into control algorithms for a particular subsystem often the future integration of such a controller into a larger system is neglected. To enable the designer to concentrate on a subsystem controller, yet to avoid problems in the integration phase, an adequate system specification method should be chosen. As robot control systems are inherently complex, this choice is of paramount importance, thus when subsystems of a rescue and exploration robot had to be designed, a systematic specification method, well embedded in the robotics domain, was selected. It was assumed that the designed robot is to consist of a mobile platform carrying a manipulator and diverse sensors. This paper focuses on the specification of a skid steering mobile platform. By assuming that the mobile platform is an embodied agent [1], it can readily be incorporated into a larger multi-agent system, where each of the subsystems is represented as an agent.

Systematic design methods distinguish between the system specification phase and its implementation. It is desirable that the specification should be detailed enough to guide the implementation in a straightforward manner. In the domain of software engineering there is a plethora of different approaches to the design of software [2, 3]. Those methods can be used for the design at hand as general guidelines, yet domain specific methods are generally easier to follow by the designers having background specific to that domain. Here the domain is robotics. A design method fulfilling the above mentioned requirements has been presented in [4–10]. It is based on the assumption that the overall control system is composed of multiple embodied agents, where each of the agents, representing a specific system component, is decomposed into: a control subsystem, real effectors, virtual effectors, real receptors and virtual effectors. Moreover, as the agents can communicate with each other through their control subsystems a multi-agent system controlling the complex system results.

The mentioned robot is being designed for research purposes, i.e. to study diverse control algorithms implemented for specific subsystems, yet tested within the whole system. Here the specification of the embodied agent representing just the mentioned mobile platform is described. The *Rex* platform has four independently driven wheels with constant orientation with respect to the body – Fig. 1.

Generally, the *Rex* platform is a test platform for algorithms concerning motion planning and real-time trajectory tracking. The motion planning methods that are currently tested involve the Endogenous Configuration Space Approach (ECSA) [11, 12] in constrained [13] and unconstrained [14] versions. Competitively to ECSA the Optimal Control Method (OPCM) is considered. A comparative study of those two

*e-mail: mariusz.janiak@pwr.edu.pl

planning methods has been presented in [15, 16]. In the field of real-time trajectory tracking, two major approaches are considered: Nonlinear Model Predictive Control (NMPC) [17–19] and an Artificial Force Method (AFM) [20,21]. Both methods rely on the knowledge of the kinematics and dynamics models of the considered mobile platform and its interaction with the ground. An effective numerical algorithms implementing the OPCM, NMPC are provided by the ACADO toolkit [17, 22]. Elaborated motion planning and trajectory tracking algorithms require easily modifiable software running on the real model of the platform. The injection of the required modification into the control software is simple only when, on the one hand, there exists a formal specification of this software pointing to the exact places that those changes have to be done. On the other hand, there are available tools accompanying the formal specification, that enable fast prototyping and testing. Such a specification of the controller structure and integration tools selection is the subject of this paper. This paper presents the outcome of utilizing the proposed design approach. Focus is on disclosing those parts of the control structure which need to be modified while testing diverse control algorithms.

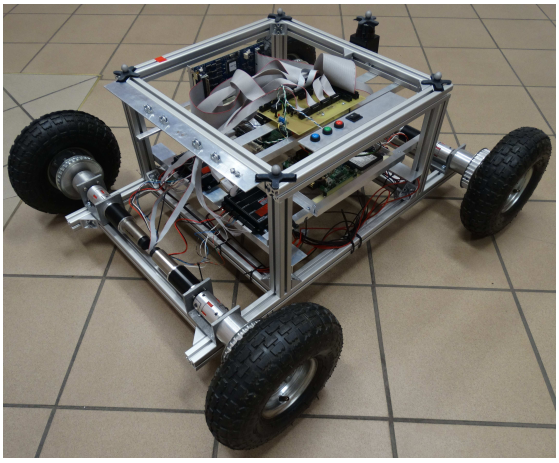

Fig. 1. The mobile platform *Rex*

The selected hardware-software toolchain is a mixture of the well integrated hardware components, supervised by a real-time operating system, and well known, open-source middleware frameworks supporting component-oriented programming paradigm, providing ready to use application building blocks and standard interfaces. With these hardware-software tools, the user is able to build a centralized real-time control system based on powerful computational devices, equipped with multifunctional I/O cards, as well as a distributed one utilizing many computers and embedded custom devices that communicate with each other in real-time.

The paper is organised in the following way. Section 2 introduces the general concept of an embodied agent, describing its structure and behaviour. Section 3 describes the REX robot control system specification. Section 4 presents the locomotion agent implementation details. Finally Sec. 5 provides conclusions.

2. Embodied agent

An embodied agent a_j, where j is the agent's designator, physically interacts with its environment through its effectors E_j and gathers information through its receptors R_j. As rarely the physical devices E_j and R_j enable interaction by using adequate level of abstraction, data used by those devices has to be transformed into appropriate concepts, e.g. motor positions have to be transformed into the pose of the end-effector represented in terms of a homogeneous transform, or a bitmap obtained from the camera has to be transformed into the pose of object detected in the image. Those transformations are performed by the virtual effectors $e_{j,n}$ and virtual receptors $r_{j,k}$ respectively, where n and k are the designators of a particular virtual effector or receptor. The definition of the appropriate abstraction level is one of the design tasks. This level is defined by the set of concepts that are used in the design of the control subsystem c_j of the agent a_j. The thus introduced subsystems communicate with each other using buffers (Fig. 2). The agent itself can communicate with other agents through its transmission buffers located in its control subsystem.

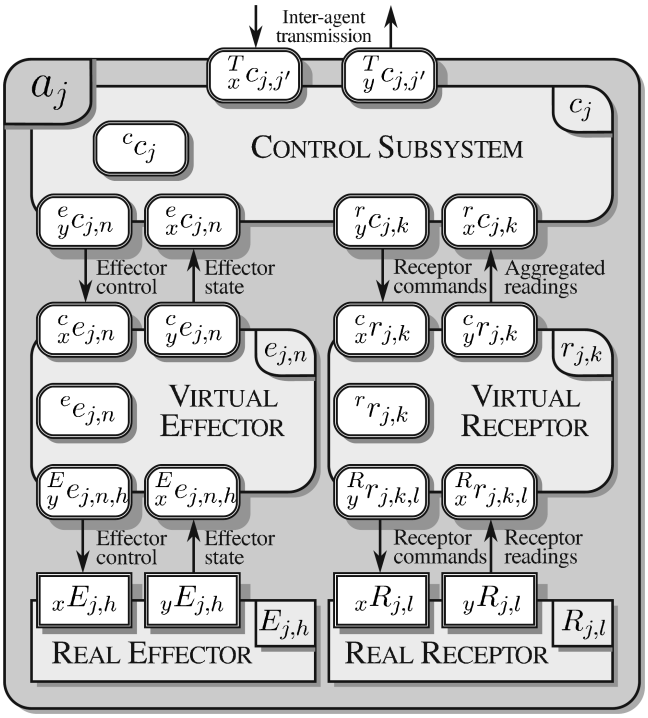

Fig. 2. General structure of an embodied agent

The systematic buffer naming convention is the following. The central symbol refers to a subsystem: e, r, c, E or R. The leading superscript designates the component that the buffer is connected to. If it is a repetition of the central symbol it implies internal memory of the subsystem. Leading subscript x labels an input buffer, while y an output buffer. Lack of this symbol implies the internal memory. The following superscript designates a discrete time instant, while the following subscript names: the agent and the instance of the considered subsystem, so it usually is a compound index. Each of those buffers has its internal structure, which has to be designed

taking into account both the type of the controlled robot and the task that it has to execute.

The mentioned buffers provide just the infrastructure. The transition functions define how the subsystem containing those buffers will behave. The input buffers and the internal memory are the input arguments of those function, while the output buffers and the memory (once again) constitute the store for the results of the computation of those transition functions. Each of the subsystems is governed by its own transition functions. The control subsystem c_j uses ${}^{c}f_j$, a virtual receptor r_j employs ${}^{r}f_j$, while a virtual effector e_j exploits ${}^{e}f_j$. Each of those transition functions must be further decomposed, as one function would be usually too complex to exhibit all the necessary behaviours of the considered subsystem. The canonic form of decomposition is the division according to the buffer in which the results of the function evaluation are placed. This decomposition requires an extra index – in this case an additional leading superscript (c, e, r, E, R or T), located after a coma, designates the target subsystem. Thus the transition functions for the three types of subsystems are decomposed in the following way.

$$\begin{cases} {}^{c}_{y}e_j^{\iota+1} = {}^{e,c}f_j({}^{e}e_j^{\iota}, {}^{E}_{x}e_j^{\iota}, {}^{c}_{x}e_j^{\iota}), \\ {}^{e}e_j^{\iota+1} = {}^{e,e}f_j({}^{e}e_j^{\iota}, {}^{E}_{x}e_j^{\iota}, {}^{c}_{x}e_j^{\iota}), \\ {}^{E}_{y}e_j^{\iota+1} = {}^{e,E}f_j({}^{e}e_j^{\iota}, {}^{E}_{x}e_j^{\iota}, {}^{c}_{x}e_j^{\iota}), \end{cases} \tag{1}$$

$$\begin{cases} {}^{c}_{y}r_j^{\iota+1} = {}^{r,c}f_j({}^{r}r_j^{\iota}, {}^{R}_{x}r_j^{\iota}, {}^{c}_{x}r_j^{\iota}), \\ {}^{r}r_j^{\iota+1} = {}^{r,r}f_j({}^{r}r_j^{\iota}, {}^{R}_{x}r_j^{\iota}, {}^{c}_{x}r_j^{\iota}), \\ {}^{R}_{y}r_j^{\iota+1} = {}^{r,R}f_j({}^{r}r_j^{\iota}, {}^{R}_{x}r_j^{\iota}, {}^{c}_{x}r_j^{\iota}), \end{cases} \tag{2}$$

$$\begin{cases} {}^{c}c_j^{i+1} = {}^{c,c}f_j({}^{c}c_j^{i}, {}^{e}_{x}c_j^{i}, {}^{r}_{x}c_j^{i}, {}^{T}_{x}c_j^{i}), \\ {}^{e}_{y}c_j^{i+1} = {}^{c,e}f_j({}^{c}c_j^{i}, {}^{e}_{x}c_j^{i}, {}^{r}_{x}c_j^{i}, {}^{T}_{x}c_j^{i}), \\ {}^{r}_{y}c_j^{i+1} = {}^{c,r}f_j({}^{c}c_j^{i}, {}^{e}_{x}c_j^{i}, {}^{r}_{x}c_j^{i}, {}^{T}_{x}c_j^{i}), \\ {}^{T}_{y}c_j^{i+1} = {}^{c,T}f_j({}^{c}c_j^{i}, {}^{e}_{x}c_j^{i}, {}^{r}_{x}c_j^{i}, {}^{T}_{x}c_j^{i}), \end{cases} \tag{3}$$

where ι is a superscript signifying the discrete time instant (the iteration number of subsystem activity). The ι superscripts are usually different for different virtual effectors and receptors. This distinction is not made evident here, because of its contextual obviousness. A superscript i in (3) signifies the discrete time instance for the control subsystem.

Transition functions are just computational elements. There must be a mechanism for reading in their arguments and distributing the results of their computations, i.e. animating the subsystems using the communication middleware. Elementary actions ${}^{c}\mathcal{A}_j$, ${}^{e}\mathcal{A}_j$ or ${}^{r}\mathcal{A}_j$ are responsible for that. Elementary actions are parameterised by transition functions. One should note that the canonically decomposed transition functions (1), (2) and (3), that define the computations that need to be done by particular subsystems of the agent, tend to be complex, thus further decomposition is necessary. To distinguish between those functions a following subscript after a coma following the agent's designator j is used (here m). Algorithm 1 presents the general pattern of an elementary action in the case of the control subsystem. In the case of virtual effectors and receptors the pattern is similar, however the leading superscript c has to be substituted either by e or r. The statement $i \rightarrow i+1$ models the elapse of cycle time, while the symbol $\rightarrowtail$ represents the inter-subsystem data transmissions.

input : ${}_{x}c_j^{i}$
output: ${}_{y}c_j^{i+1}$

${}_{y}c_j^{i+1} \leftarrow {}^{c}f_{j,m}\,({}_{x}c_j^{i})$; // Compute the transition function;
${}^{e}_{y}c_j^{i+1} \rightarrowtail {}_{x}e_j$; ${}^{r}_{y}c_j^{i+1} \rightarrowtail {}_{x}r_j$; ${}^{T}_{y}c_j^{i+1} \rightarrowtail {}^{T}_{x}c_{j'}$; // Execute the action;
$i \leftarrow i+1$; // Wait;
${}_{y}e_j \rightarrowtail {}^{e}_{x}c_j^{i}$; ${}_{y}r_j \rightarrowtail {}^{r}_{x}c_j^{i}$; ${}^{T}_{y}c_{j'} \rightarrowtail {}^{T}_{x}c_j^{i}$; // Read in the arguments;

Algorithm 1. Elementary action of the control subsystem ${}^{c}\mathcal{A}_{j,m}$

Elementary actions, and thus transition functions associated with them are executed cyclically. Cyclic execution of a particular elementary action is termed a behaviour. Further decomposition of transition functions multiplies the number of elementary actions and thus behaviours. This multiplicity implies the necessity of selection of the behaviour that should be active in each instant of time and a method of choosing the discrete instant when the particular behaviour should terminate its repetitive actions. Both choices are made by predicates, i.e. Boolean valued functions, one of which is termed the initial condition and the other the terminal condition. In the case of the control subsystem those will be: ${}^{c}f^{\sigma}_{j,m}$ and ${}^{c}f^{\tau}_{j,m}$ respectively. If the terminal condition associated with a certain behaviour is fulfilled its cyclic execution is terminated and a new behaviour has to be picked for execution. This is done by taking into account the initial conditions associated with the behaviours. Thus a finite state machine (FSM) structure results. The nodes of its graph are labeled by the behaviours, e.g. in the case of the control subsystem those will be: $c\mathcal{B}_{j,m}({}^{c}f_{j,m}, {}^{c}f^{\tau}_{j,m})$, and the initial conditions are used to form the labels of the arcs.

Now two forms of behaviour can be defined. One is the while type behaviour and the other is the repeat type behaviour. In the case of the control subsystem those are: ${}^{c}_{*}\mathcal{B}_{j,m}({}^{c}f_{j,m}, {}^{c}f^{\tau}_{j,m})$ (Algorithm 2), which tests the terminal condition before executing an iteration of the loop, and ${}^{c}_{+}\mathcal{B}_{j,m}({}^{c}f_{j,m}, {}^{c}f^{\tau}_{j,m})$ (Algorithm 3), which performs the test after the execution of the loop.

input : ${}^{c}f_{j,m}$, ${}^{c}f^{\tau}_{j,m}$
output:

while ${}^{c}f^{\tau}_{j,m}({}_{x}c_j^{i})$ = *false* **do**
  ${}^{c}\mathcal{A}_{j,m}\,({}^{c}f_{j,m})$;
end

Algorithm 2. Control subsystem behaviour ${}^{c}_{*}\mathcal{B}_{j,m}({}^{c}f_{j,m}, {}^{c}f^{\tau}_{j,m})$

input : ${}^{c}f_{j,m}$, ${}^{c}f^{\tau}_{j,m}$
output:

repeat
  ${}^{c}\mathcal{A}_{j,m}\,({}^{c}f_{j,m})$;
until ${}^{c}f^{\tau}_{j,m}({}_{x}c_j^{i})$ = *true*;

Algorithm 3. Control subsystem behaviour ${}^{C}_{+}\mathcal{B}_{j,m}({}^{c}f_{j,m}, {}^{c}f^{\tau}_{j,m})$

The behaviour patterns ${}^{c}_{*}\mathcal{B}_{j,m}({}^{c}f_{j,m}, {}^{c}f^{\tau}_{j,m})$ and ${}^{c}_{+}\mathcal{B}_{j,m}({}^{c}f_{j,m}, {}^{c}f^{\tau}_{j,m})$ label the nodes of the FSM governing the control subsystem. They are parameterised by transition functions ${}^{c}f_{j,m}$, and terminal conditions ${}^{c}f^{\tau}_{j,m}$ that must be provided by the designer. The designer is also responsible for providing the graph of the FSM, thus the initial conditions must be defined by him/her too. The behaviours of the virtual effectors and receptors are defined likewise, but the leading superscript c should be substituted by e or r respectively.

In the following text the internal structure of the subsystem buffers and their transition functions as well as the terminal and initial conditions will be defined for the REX robot. The thus defined functions are used as arguments of elementary actions and behaviours as defined by Algorithms 1, 2 and 3. Moreover the FSMs will be created, thus producing the specification of the whole control system.

3. Locomotion agent

The general description of the designed system structure and its functioning are as follows. The mobile platform *Rex* is represented by the agent that performs the locomotion task. This is an embodied agent that directly interacts with the environment, and is able to read its state. Figure 3 illustrates the location of this agent in the control system of an autonomous service robot. The ontology agent a_2 generates the tasks for the locomotion agent a_1 in the form of a plan. The plan consists of a series of points corresponding to the positions specified on the map. Those points have to be traversed in a fixed order. The agent a_1 reports the completion of the task by sending the acknowledge signal *ack* to the agent a_2. The locomotion agent exchanges information also with the map agent a_3, which possesses the information about the position of the robot on the map. The map agent obtains this information from a motion capture system during indoor experiment phase or a navigation system in the case of outdoor operation. The locomotion agent a_1 receives the current robot position estimates from the map agent. Optionally it can send back the state estimates worked out by itself.

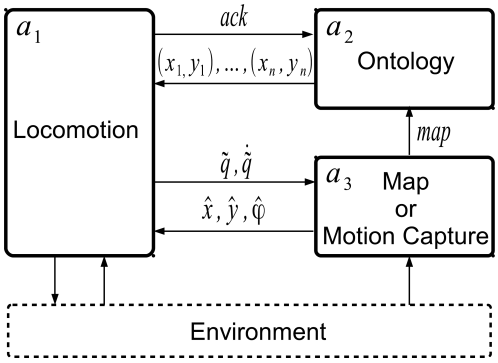

Fig. 3. A fragment of the autonomous service robot system

The structure of the locomotion agent a_1 is presented in Fig. 4. The locomotion agent a_1 consists of a control subsystem c_1, a virtual effector $e_{1,1}$ and a virtual receptor $r_{1,1}$. The control subsystem c_1 coordinates the agent's actions consisting in the realization of the high level control goals. This involves supervision of the plan execution, trajectory planning and communication with the other agents. The virtual effector $e_{1,1}$ directly controls the mobile platform movements – currently it implements the model based control algorithm and the real-time state estimator. The mobile platform E_1 consist of four torque driven electric motors ${}_{x}E_1$ and proprioceptors: inertial measurement unit ${}_{y}E_{1,1}$, a set of force sensors ${}_{y}E_{1,2}$ mounted between body frame and each wheel actuation unit that measure wheels load, and four rotary encoders associated with platform wheels ${}_{y}E_{1,3} \ldots {}_{y}E_{1,6}$. The visual odometry utilizing information from the stereo camera R_1 is represented by a virtual receptor $r_{1,1}$. Further, each subsystem of the locomotion agent will be discussed in more detail. The presentation follows a top-down fashion, which facilitates understanding of the functioning of the system, but does not necessarily imply that the system as a whole was designed in this fashion. Usually top-down and bottom-up approaches are mixed and applied iteratively. This was the case here, but in the following description this fact is not evident, for obvious reasons.

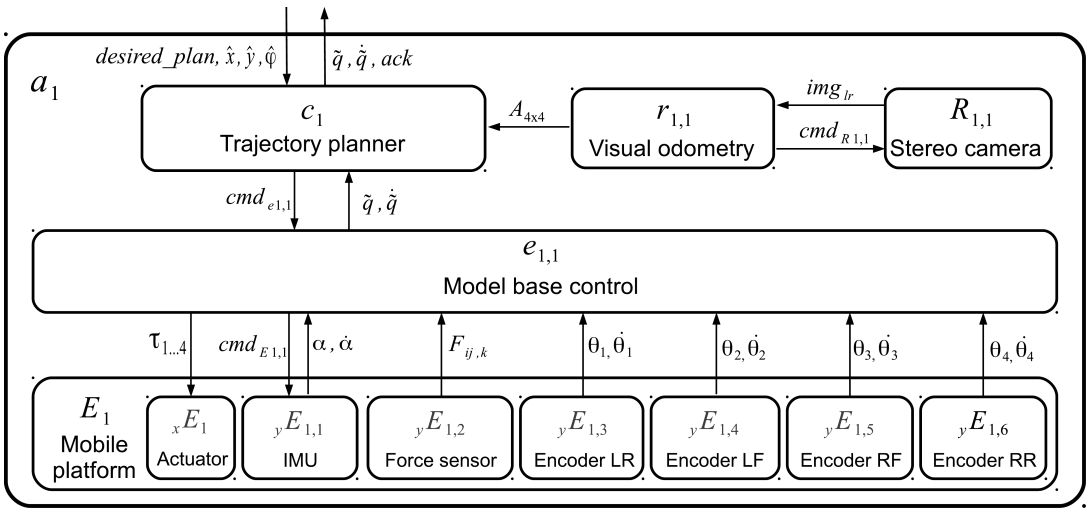

Fig. 4. The locomotion agent

3.1. Control subsystem c_1. In general the control subsystem c_1 of the embodied agent a_1 acts in the following way. It is responsible for the execution of the agent's task as well as the coordination of functioning of the subsystems that this agent is built of. Here the goal of the locomotion agent is to execute the plan commissioned by the ontology agent. The plan is defined as a series of points on the map, through which the robot has to pass in a specified order. It is assumed that the ontology agent will provide a sufficiently dense list of points to avoid collisions with obstacles. The control subsystem adds to each point the information about the desired platform orientation, then transforms the thus modified series of points into the mobile platform trajectory and finally passes it to the virtual effector which will try to execute it. The trajectory is generated by the *Trajectory Planner* module that operates within the control subsystem. The planner implements the endogenous configuration space approach [11]. The performance of this method in a similar application has been presented in [15]. Due to the high computational complexity of this method and a relatively long time needed for trajectory planning, a planning task and a trajectory execution task have been split into two concurrent processes running within the *Trajectory Planner* module. The planner generates, as fast as possible, the consecutive fragments of the trajectory, that connect respective points forming the plan. These fragments are assembled into a dedicated list. Independently, the trajectory execution process supervises the progress of the trajectory realization. It takes a trajectory fragment from the list and passes it to the virtual effector adding information about the number of the steps (*step*) needed for trajectory execution and number of the step (*notify*) after which the virtual effector should report its ready state. This method of interaction of the virtual effector and the control subsystem was used by all systems based on the MRROC++ robot programming framework [10, 23]. Keeping *notify* smaller than *step* prevents the platform from unintended stops. The control subsystem assumes that the plan has been completed when the entire trajectory has been generated and sent to the virtual effector for execution. This fact is reported to ontological agent a_2 by an acknowledge signal *ack*. In fact, transfer by the control subsystem of the last fragment of the trajectory to the virtual effector is not equivalent to reaching the end of the trajectory by the mobile platform. Usually in such a case, the virtual effector will be somewhere in the middle of the trajectory fragment execution. The advantage of early reporting of the plan accomplishment is to avoid the situation when the robot holds up the task execution due the lack of a new plan, assuming that ontological agent needs some time for the preparation of a new plan.

As mentioned earlier, besides the task execution, the control subsystem c_1 also coordinates the data flow inside the agent. It transfers the information from the external world through the virtual receptor $r_{1,1}$ to the virtual effector $e_{1,1}$. Moreover, it retrieves the platform state estimates computed by the virtual effector $e_{1,1}$.

The control system activities have been decomposed into three behaviors: waiting for the ontological agent a_2 to send the plan ${}^{c}_{+}\mathcal{B}_{1,1}({}^{c}f_{1,1}, {}^{c}f^{\tau}_{1,1})$, plan execution ${}^{c}_{*}\mathcal{B}_{1,2}({}^{c}f_{1,2}, {}^{c}f^{\tau}_{1,2})$ and notification by an acknowledge signal *ack* that the plan has been executed ${}^{c}_{+}\mathcal{B}_{1,3}({}^{c}f_{1,3}, {}^{c}f^{\tau}_{1,3})$. These behaviors are selected by the finite state machine presented in Fig. 5. The data flow diagrams of the transition functions implementing each behavior have been shown in Fig. 6. The function ${}^{c,e}f_1$ is used by behaviors 1 and 3, while functions ${}^{c,c}f_{1,3}$ and ${}^{c,r}f_1$ are not necessary. As mentioned earlier, the transition functions take as arguments the contents of the input buffers and internal memory and they produce the contents of the output buffers and the same memory. The contents of the control subsystem memory and buffers is the following.

- The internal memory contains:
  - ${}^{c}c_1[plan]$ – the plan containing the list of positions and orientations of the mobile platform with respect to the map,
  - ${}^{c}c_1[traj]$ – the list of trajectory fragments generated by the planner,
  - ${}^{c}c_1[trajgen]$ – the structure containing information relevant for the trajectory planner,
  - ${}^{c}c_1[new]$ – new plan flag.
- The input buffer consist of:
  - ${}^{c}_{x}c_{1,1}[\widetilde{q},\ \dot{\widetilde{q}}]$ – the mobile platform state estimate worked out by the virtual effector,
  - ${}^{e}_{x}c_{1,1}[ready]$ – the virtual effector ready flag,
  - ${}^{r}_{x}c_{1,1}[A_{4\times4}]$ – a transform matrix provided by the virtual receptor,
  - ${}^{T}_{c}c_{1,2}[desired\_plan]$ – the plan generated by the ontology agent,
  - ${}^{T}_{x}c_{1,3}[\widehat{x},\ \widehat{y},\ \widehat{\varphi}]$ – the position and orientation of the robot with respect to the map provided by the map agent.
- The output buffer includes:
  - ${}^{e}_{y}c_{1,1}[\widehat{x},\ \widehat{y},\ \widehat{\varphi}]$ – the position and the orientation of the robot with respect to the map,
  - ${}^{e}_{y}c_1[A_{4\times4}]$ – a transform matrix,
  - ${}^{e}_{y}c_1[traj]$ – the trajectory fragment transferred for execution,
  - ${}^{e}_{y}c_1[steps]$ – number of steps that are needed to execute the trajectory fragment,
  - ${}^{e}_{y}c_1[notify]$ – step number in which the virtual effector should report the ready state ($notify < steps$),
  - ${}^{T}_{y}c_{1,2}[ack]$ – the acknowledge flag reporting that the agent is ready for a new plan,
  - ${}^{T}_{y}c_{1,3}[\widetilde{q},\ \dot{\widetilde{q}}]$ – state estimate.

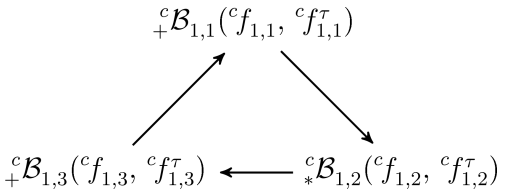

Fig. 5. The finite state machine of the control subsystem c_1

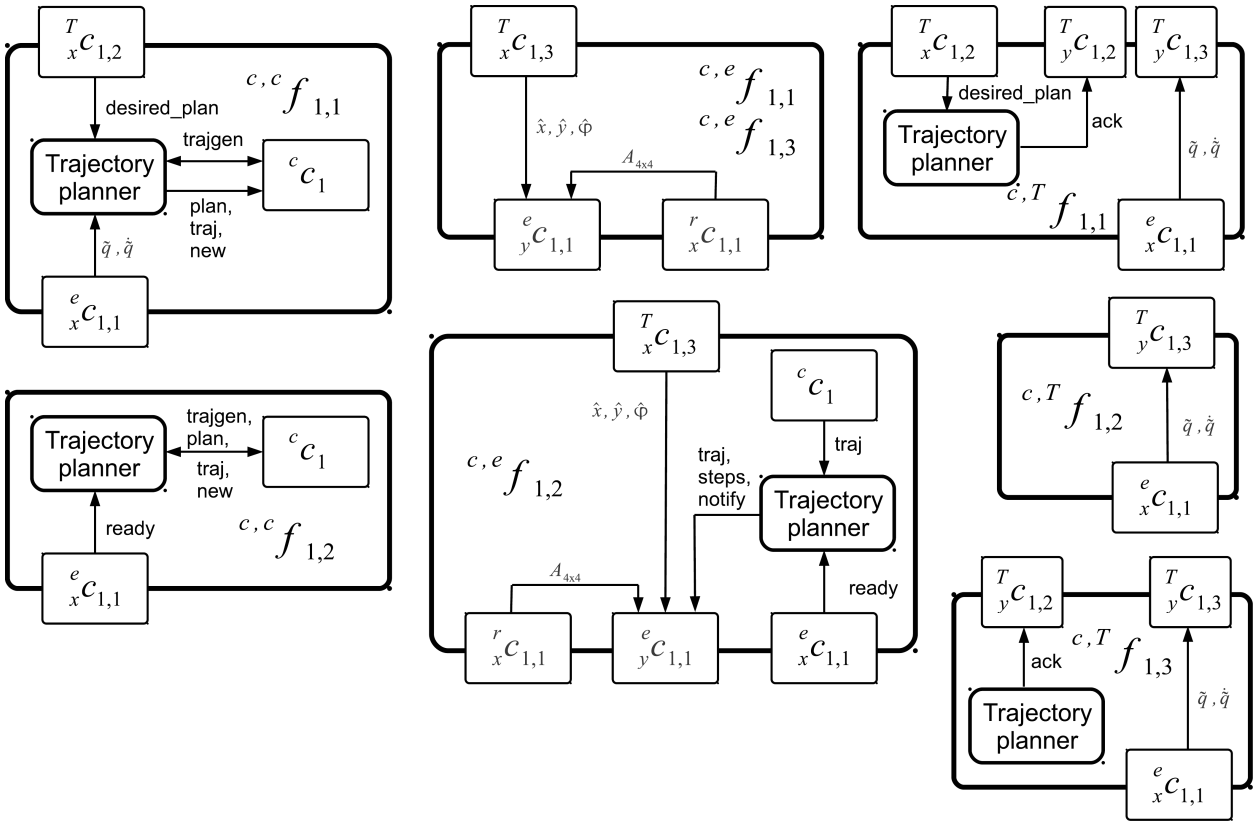

Fig. 6. The data flow diagrams for the transition functions of the control subsystem c_1

The *trajectory planner* has been decomposed into a set of transition subfunctions described by the Algorithms 4–8. The following auxiliary functions have been defined:

- `isnew`(buf) checks whether buffer *buf* contains new data,
- `proc`$(desired\_plan)$ appends to the *desired plan* the information about consecutive platform orientations,
- `trajgenInit`$(trajgen, state)$ initializes the trajectory planner structure *trajgen* using the current platform *state*,
- `trajgenStart`$(trajgen, point)$ starts the planning process of the trajectory that links the last processed position with the given *point*,
- `trajgenStop`$(trajgen)$ finalizes the trajectory planning process,
- `trajgenIsReady`$(trajgen)$ checks whether the trajectory planning process has been completed,
- `trajgenStat`$(trajgen)$ gets the current state of the trajectory planner,
- `listIsEmpty`$(list)$ checks whether the *list* is empty,
- `listAddLast`$(list, data)$ adds a new element containing *data* at the end of the *list*,
- `listDelFirst`$(list)$ removes the first element from the *list*,
- `listAddLastDelFirst`$(list, data)$ adds a new element containing *data* at the end of the *list*, and simultaneously removes the first element from the *list*,
- `listGetData`$(list)$ gets the data from the first element of the *list*,
- `trajToSteps`$(traj)$ calculates the number of steps required for the trajectory *traj* execution,
- `trajToSteps`$(traj)$ calculates the number of the step after which the virtual effector should send the ready signal.

input : $desired\_plan^i$, $trajgen^i$, $(\tilde{q}, \dot{\tilde{q}})^i$
output: $plan^{i+1}$, $traj^{i+1}$, $trajgen^{i+1}$, new^{i+1}

if $isnew(desired\_plan^i)$ **then**
  $plan^{i+1} \leftarrow$ `proc`$(desired\_plan^i)$;
  $traj^{i+1} \leftarrow \{\}$;
  $trajgen^{i+1} \leftarrow$ `trajgenInit`$(trajgen^i, (\tilde{q}, \dot{\tilde{q}})^i)$;
  $new^{i+1} \leftarrow true$;
end

Algorithm 4. The control subsystem transition subfunction ${}^{c,c}f_{1,1,tp}({}^{c}c_1^i, {}_{x}c_1^i)$ associated with the trajectory planer

input : $desired\_plan^i$
output: ack^{i+1}

if $isnew(desired\_plan^i)$ **then**
  $ack^{i+1} \leftarrow false$;
end

Algorithm 5. The control subsystem transition subfunction ${}^{c,T}f_{1,1,tp}({}^{c}c_1^i, {}_{x}c_1^i)$ associated with the trajectory planer

input : $traj^i$, $trajgen^i$, $plan^i$, new^i, $ready^i$
output: $traj^{i+1}$, $trajgen^{i+1}$, $plan^{i+1}$, new^{i+1}

if $ready^i$ & $\sim$`listIsEmpty`($traj^i$) **then**
  if `trajgenIsReady`($trajgen^i$) **then**
    $traj^{i+1} \leftarrow$ `listAddLastDelFirst`($traj^i$, `trajgenGetTraj`($trajgen^i$))
  else
    $traj^{i+1} \leftarrow$ `listDelFirst`($traj^i$)
  end

else if `trajgenIsReady`($trajgen^i$) **then**
  $traj^{i+1} \leftarrow$ `listAddLast`($traj^i$, `trajgenGetTraj`($trajgen^i$));
end
if new^i **then**
  $new^{i+1} \leftarrow false$;
  $trajgen^{i+1} \leftarrow$ `trajgenStart`($trajgen^i$, `listGetData`($plan^i$));
  $plan^{i+1} \leftarrow$ `listDelFirst`($plan^i$);

else if `trajgenIsReady`($trajgen^i$) **then**
  if `listIsEmpty`($plan^i$) **then**
    $trajgen^{i+1} \leftarrow$ `trajgenStop`($trajgen^i$)
  else
    $trajgen^{i+1} \leftarrow$ `trajgenStart`($trajgen^i$, `listGetData`($plan^i$));
    $plan^{i+1} \leftarrow$ `listDelFirst`($plan^i$);
  end
else
  $trajgen^{i+1} \leftarrow$ `trajgenGetStat`($trajgen^i$)
end

Algorithm 6. The control subsystem transition subfunction ${}^{c,c}f_{1,2,tp}({}^{c}c_1^i, {}_{x}c_1^i)$ associated with the trajectory planer

input : $ready^i$, $traj^i$
output: $traj^{i+1}$, $steps^{i+1}$, $notify^{i+1}$

if $ready^i$ & $\sim$`listIsEmpty`($traj^i$) **then**
  $traj^{i+1} \leftarrow$ `listGetData`($traj^i$);
  $steps^{i+1} \leftarrow$ `trajToSteps`(`listGetData`($traj^i$));
  $notify^{i+1} \leftarrow$ `trajToNotify`(`listGetData`($traj^i$));
end

Algorithm 7. The control subsystem transition subfunction ${}^{c,e}f_{1,2,tp}({}^{c}c_1^i, {}_{x}c_1^i)$ associated with the trajectory planer

output: ack^{i+1}

$ack^{i+1} \leftarrow true$;

Algorithm 8. The control subsystem transition subfunction ${}^{c,T}f_{1,3,tp}({}^{c}c_1^i, {}_{x}c_1^i)$ associated with the trajectory planer

3.2. Virtual effector $e_{1,1}$. The virtual effector implements only one behavior defined by the transition functions ${}^{e,e}f_{1,1}$, ${}^{e,E}f_{1,1}$ and ${}^{e,c}f_{1,1}$. The data flow diagrams of those functions are presented in Fig. 7. The virtual effector translates the high level motion command provided by the control subsystem c_1 in the form of trajectories into torque signals driving wheels of the mobile platform E_1. The translation process is handled by the *model-based controller* that implements predictive trajectory tracking algorithm [24] adopted to skid-steering platform. When a new trajectory fragment is received from the control subsystem, the virtual effector executes a behaviour in *step* number of iterations. The *ready* signal is sent back to the control subsystem after the execution of the *notify* iterations of the behaviour. When the platform reaches the end of the trajectory, and a new trajectory fragment is not delivered, the virtual effector will stabilize the last point of the trajectory being executed. The values of *step* and *notify* parameters are set for each trajectory fragment by the control subsystem separately.

In order to solve a trajectory tracking problem the following optimal control problem has been formulated: find an admissible control $u(\cdot) \in \mathcal{U}$ that minimizes the objective function

$$\mathcal{J}(u(\cdot)) = \int_0^T \big((q(t) - q_d(t))^T P(q(t) - q_d(t)) + u(t)^T R u(t)\big)\, dt,$$

conforming to the system equation $\dot{q} = f(q, u, p)$, as well as control and state constraints in the form

$$s(q(t), u(t)) \leq 0 \quad \text{(permanent constraints)}$$
$$s(q(T), u(T)) \leq 0 \quad \text{(boundary constraints)},$$

where P and R are positive defined matrices, $q = (x, y, \phi, \theta_1, \theta_2, \theta_3, \theta_4)$ are the natural generalized coordinates of the mobile platform [15] ($q(t)$ is the computed trajectory and $q_d(t)$ is the desired trajectory), $u = (\tau_1, \tau_2, \tau_3, \tau_4)$ represent the actuation torques and p represents the model parameters which correspond to the slip coefficients that characterize the friction of the wheels against the ground [15]. After discretization, the optimal control problem can be re-formulated as a constrained optimization problem, and solved using sequential quadratic programming and direct multiple shooting approach [22]. A predictive version of such constrained optimization algorithm provided by the *ACADO Toolkit* [17] is utilized by the model based controller.

The trajectory tracking algorithm during each iteration requires the information about the current platform state $\widetilde{q}$ and the values of the wheels slip coefficients $\widetilde{p}$. This information is provided by the *state estimator* that implements a MHE (*Moving Horizon Estimation*) algorithm [25,26]. This algorithm makes use of the platform motion model and sensor data provided by: the wheel rotary incremental encoders $(\theta_{1\ldots4}, \dot{\theta}_{1\ldots4})$, the inertial measurement unit $(\alpha, \dot{\alpha})$, force sensors $(F_{ij,k})$, the visual odometry $(A_{4\times4})$, and the map agent $(\widehat{x}, \widehat{y}, \widehat{\phi})$. The MHE problem is formulated as follows

$$\min_{q(\cdot),p} \int_0^T ||Y(t) - h\,(q(t), u(t), p)\,||^2 dt$$

subject to the system equation $\dot{q} = f(q, u, p)$, and constraints $s(q(t), p) \leq 0$, where q, u and p represent state, control and model parameters respectively, as in the case of the trajectory tracking problem. The $h\,(q(t), u(t), p)$ is the measurement function, while $Y(t)$ represents the current observations of $\theta_{1\ldots4}$, $\dot{\theta}_{1\ldots4}$, α, $\dot{\alpha}$, $F_{ij,k}$, $A_{4\times4}$, $\widehat{x}$, $\widehat{y}$ and $\widehat{\phi}$. Due to the fact, that the MHE problem is formulated as an optimization problem, it can be solved effectively with the methods implemented in the *ACADO Toolkit* as well.

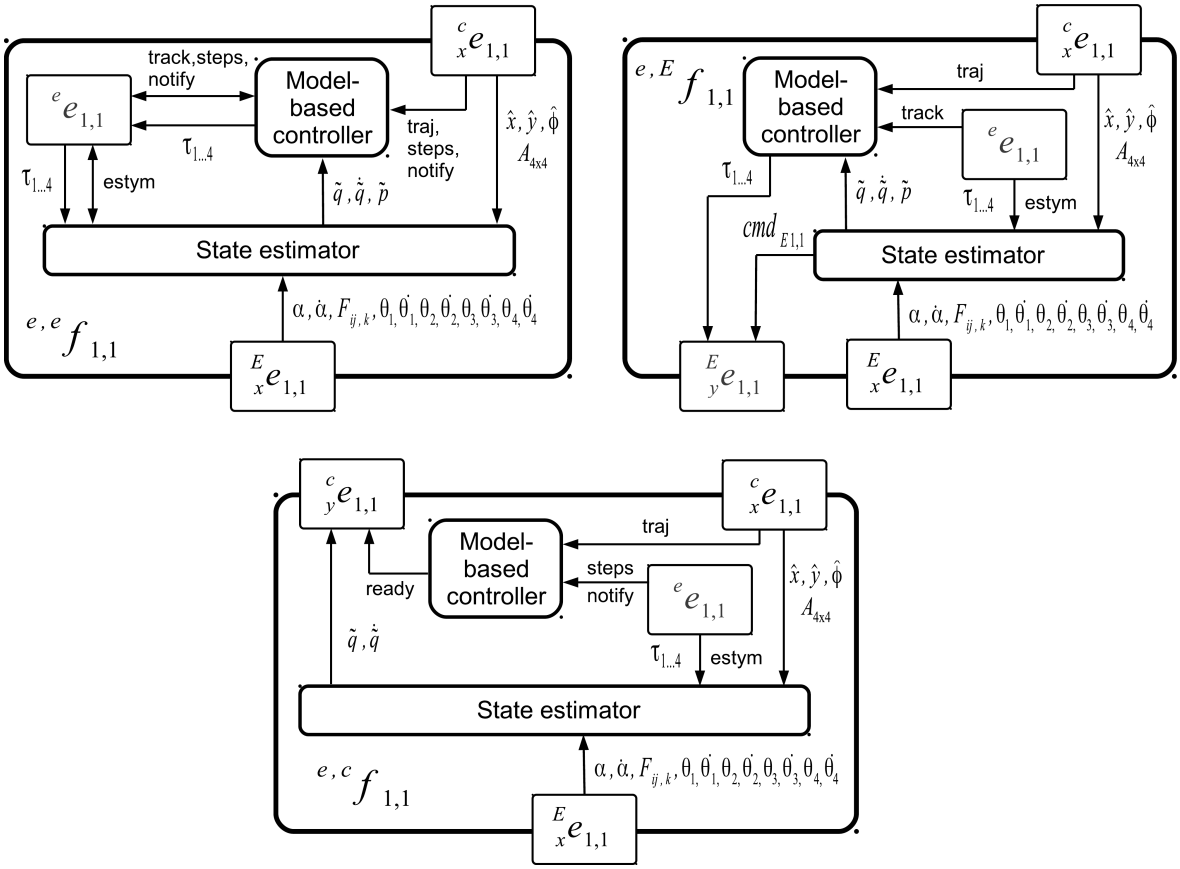

Fig. 7. The data flow diagrams of the transition functions of the virtual effector $e_{1,1}$

The contents of the virtual effector buffers are the following.

- The internal memory contains:
  - ${}^{e}e_{1,1}[track]$ – the structure containing information relevant to the trajectory tracking algorithm
  - ${}^{e}e_{1,1}[estym]$ – the structure containing information relevant to the state estimator algorithm,
  - ${}^{e}e_{1,1}[step]$ – the number of steps which remain to the end of the trajectory execution,
  - ${}^{e}e_{1,1}[notify]$ – the number of steps which remain to report the ready state,
  - ${}^{e}e_{1,1}[\tau]$ – control signals.
- The input buffer consists of:
  - ${}^{c}_{x}e_{1,1}[\widehat{x}, \widehat{y}, \widehat{\varphi}]$ – position and orientation of the robot on the map,
  - ${}^{c}_{x}e_{1,1}[A_{4\times4}]$ – the visual odometry transform matrix,
  - ${}^{c}_{x}e_{1,1}[traj]$ – trajectory fragment to be executed,
  - ${}^{c}_{x}e_{1,1}[steps]$ – number of steps needed for the trajectory fragment execution,
  - ${}^{c}_{x}e_{1,1}[notify]$ – number of steps after which the virtual effector should report the ready state,
  - ${}^{E}_{x}e_{1,1}[\alpha, \dot{\alpha}]$ – orientation and angular velocity of the platform measured by the inertial measurement unit,
  - ${}^{E}_{x}e_{1,1}[F_{ij,k}]$ – measured contact forces,
  - ${}^{E}_{x}e_{1,1}[\theta_{1\ldots4}, \dot{\theta}_{1\ldots4}]$ – position and angular velocity of each wheel measured by the associated rotary incremental encoder.
- The output buffer is composed of:
  - ${}^{c}_{y}e_{1,1}[\widetilde{q}, \dot{\widetilde{q}}]$ – state estimate,
  - ${}^{c}_{y}e_{1,1}[ready]$ – virtual effector ready flag,
  - ${}^{E}_{y}e_{1,1}[\tau]$ – control signals,
  - ${}^{E}_{y}e_{1,1}[cmd_{E1,1}]$ – the reset signal for the inertial measurement unit.

3.3. Virtual receptor $r_{1,1}$. The virtual receptor $r_{1,1}$ implements the real-time visual odometry algorithm presented in [27]. Although the algorithm is well optimized, it is computationally complex and thus runs with much lower frequency than the state estimator provided by the virtual effector. For this reason, during the decomposition process, it has been separated from state estimator and moved to a distinct subsystem. In order to increase the frequency of the visual odometry algorithm, it can be ported to GPU [28]. The virtual receptor defines only one behavior, its transition functions are presented in Fig. 8. Taking the stream of the stereo images captured by the stereo camera R_1 as input, the *visual odometry* estimates the motion of the cameras in 6DOF. The cameras are fixed to the mobile platform, therefore motion of the cameras

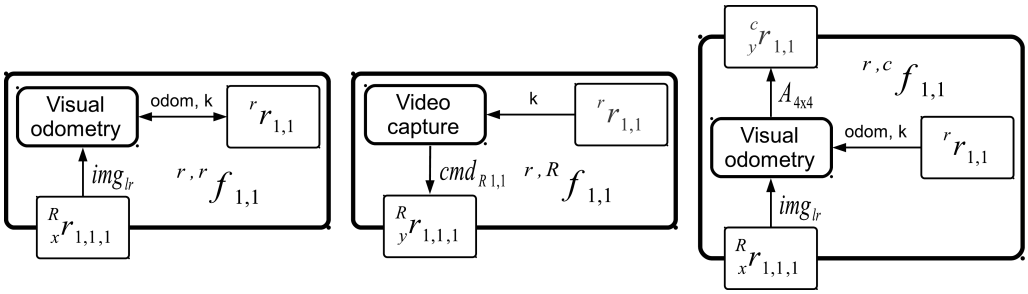

Fig. 8. The data flow diagrams of the transition functions of the virtual receptor $r_{1,1}$

unambiguously determines the motion of the platform. As the result of computations, *visual odometry* provides a transform matrix $A_{4\times4}$ that determines the motion of the platform that took place between the two last measurements. The frequency of image acquisition is controlled by the *video capture* block. It sends the acquisition command to the stereo camera R_1 every k steps of the virtual receptor behaviour. The updated stereo images will be available in the next behaviour iteration at the earliest. To minimize the time between image acquisition and image processing, a frequency of the virtual receptor behaviour is higher than the image acquisition frequency. As soon as the stereo image will be available, usually in the step $k+1$, the *visual odometry* starts the estimation process and sends the resulting transform matrix $A_{4\times4}$ to the agent control subsystem. It is paramount that the delay introduced into transferring this data to the control subsystem be low, thus the frequency of the virtual agent functioning is much higher than the frequency of image acquisition. After that transfer the virtual receptor waits till the next stereo image will be captured.

The contents of the virtual receptor buffers are the following.

- The internal memory contains:
  - ${}^{r}r_{1,1}[k]$ – the number of steps which remain to the next image acquisition,
  - ${}^{r}r_{1,1}[odom]$ – the structure containing information relevant for the visual odometry algorithm,
- The input buffer is composed of:
  - ${}^{R}_{x}r_{1,1,1}[img_{lr}]$ – the stereo images.
- The output buffer holds:
  - ${}^{c}_{y}r_{1,1}[A_{4\times4}]$ – the current transformation matrix,
  - ${}^{R}_{y}r_{1,1}[cmd_{R1,1}]$ – the acquisition command for the stereo camera.

4. Implementation

The general concept of the distributed architecture implementing the specification described in Sec. 3 has been presented in Fig. 9. As the computational load of the investigated algorithms cannot be judged a priori the architecture of the implemented system has to be extensible – thus the anticipated multitude of PC computers. The hardware can consist of several computational devices such as PCs or ARM based computers and a number of custom embedded controllers. The number of computational devices and custom controllers is not limited. The absolute minimum is one computational device. The intention is that the computers should perform high level control tasks that require high computational power and many resources. However this is not obligatory, so low level control algorithms can be hosted as well. Computers can be equipped with any number of internal or external devices that extend their functions and communication abilities, e.g. cameras, range finders, multifunctional I/O cards and WiFi network cards. Computers are supervised by real-time Linux with Xenomai [29] extension. Xenomai provides hard real-time support to the user-space applications. It is well integrated with the GNU/Linux environment and implements many programming interfaces including the native one and POSIX. The software stack is based on two well known robotics frameworks: ROS [30] and OROCOS [31]. No time critical components should be implemented as ROS nodes, while time critical as OROCOS components. This distinction is due to the fact that OROCOS is fully integrated with Xenomai. OROCOS components are associated with real-time threads and utilize Xenomai infrastructure. ROS has not been designed as a real-time framework, however it provides a rich set of tools, services and ready to use application building blocks. OROCOS and ROS frameworks are well integrated, each framework provides common interfaces and transport layer for communication between components located within one machine as well as when they are spread over several machines.

Custom devices with embedded microcontrollers can be easily adopted to any specific requirements regarding functionality, resources and dimensions. Such devices are dedicated to performing low level control tasks such as: motor control, signal conditioning and sensor fusion. Usually this kind of tasks require specialized resources, real-time response and high stability. For this reason custom embedded controllers, are managed by FreeRTOS [32], a tinny footprint, hard real-time operating system dedicated to small embedded devices. FreeRTOS has very portable source code structure. It is predominantly written in C – typically its kernel binary image is in the range of 4k to 9k bytes, and it support 34 different architectures including the very popular ARM Cortex-M. Applications hosted by custom controllers are implemented mainly as RTOS tasks. Critical application parts can be implemented as bare metal procedures running directly on hardware. Number of applications running on each device is limited only by its resources.

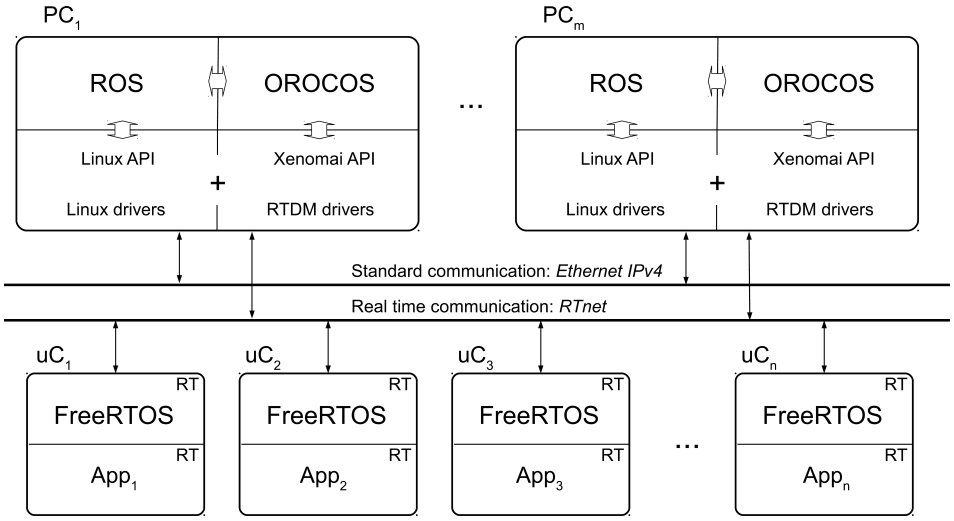

Fig. 9. RobREx general implementation architecture

The communication between components running on different machines is made possible with a ROS publish-subscribe message passing mechanism and also with an OROCOS CORBA framework. Those transport layers are based on standard Ethernet IPv4 protocol and can be used for non-time-critical communication. Time critical communication between computational devices, as well as custom embedded controllers, is possible with the RTnet [33] framework, a hard real-time network protocol stack for the *Xenomai* and recently also for the FreeRTOS. The RTnet operates on standard Ethernet hardware, implements common Ethernet protocols in a deterministic way, and provides a standard POSIX socket API. In the future, it is planned to implement a publish-subscribe protocol over the RTnet.

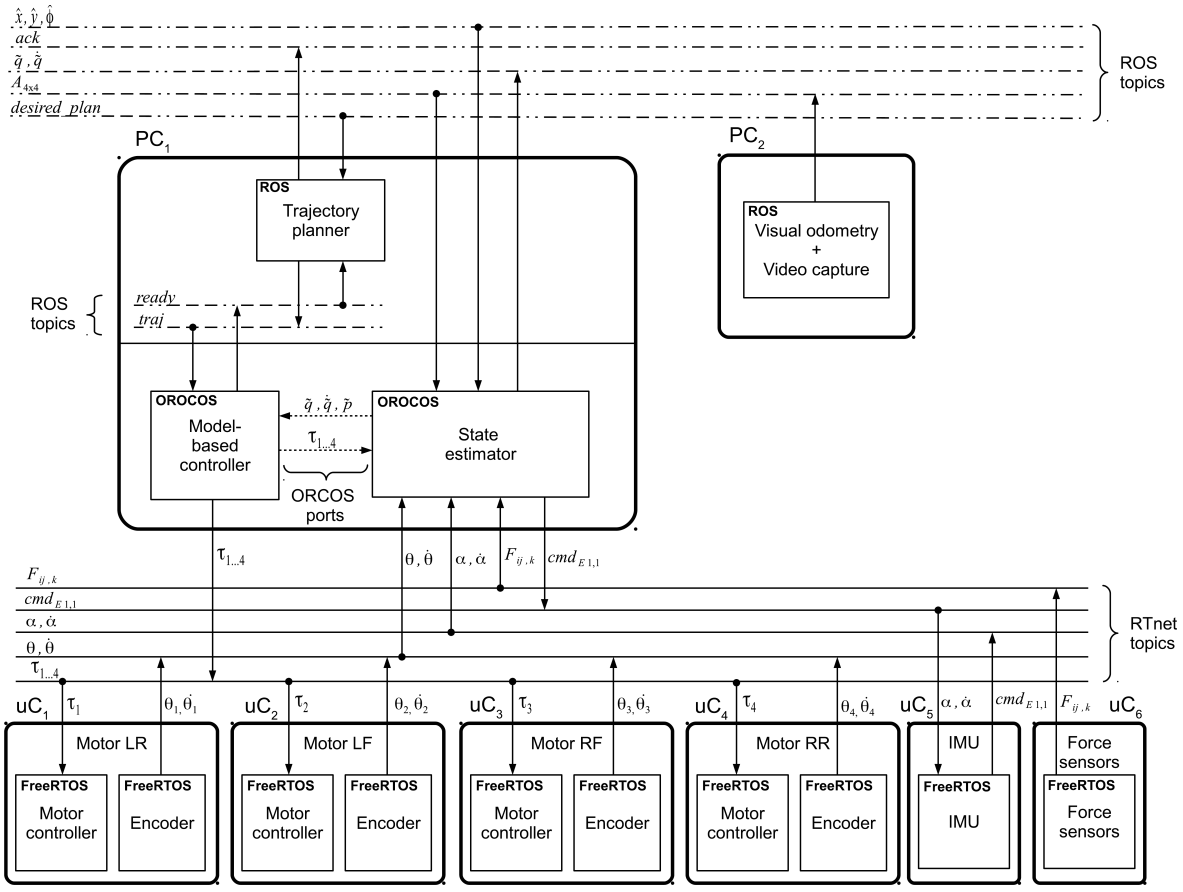

Fig. 10. Implementation of the locomotion agent

With the presented toolchain, an implementation of the locomotion agent on real hardware is straightforward. The final structure of the *Rex* platform control system, is presented in Fig. 10. This distributed control system consists of two computational devices $\mathtt{PC}_1$ and $\mathtt{PC}_2$ as well as six custom embedded devices $\mathtt{uC}_{1\ldots6}$. $\mathtt{PC}_1$ is an industrial grade mini-ITX PC with Intel Core i7 processor and 8GB RAM operational memory, running Linux with Xenomai, ROS and OROCOS. The control subsystem c_1 and the virtual effector $e_{1,1}$ of the agent a_1 are hosted by this machine. The *trajectory planner* is implemented as a ROS node, while *model-based controller* and *state estimator* as OROCOS components. The virtual receptor $r_{1,1}$ is hosted by the second computational device the $\mathtt{PC}_2$. This is an industrial grade small PC/104 embedded PC with Intel Atom processor, 2GB RAM and IEEE 1394 FireWire interface. This machine is supervised by the standard Linux with ROS, thus the *visual odometry* is implemented as a ROS node. Communication between the two machines as well as between the agents is realized by the ROS transport layer.

The real effector E_1 of the locomotion agent is realized by six custom embedded controllers $\mathtt{uC}_{1\ldots6}$, all supervised by the FreeRTOS operating system. Devices $\mathtt{uC}_{1\ldots4}$, associated with the respective platform wheels, provide motor control and encoder measurements. The device $\mathtt{uC}_5$ handles the communication with the IMU, and the device $\mathtt{uC}_6$ provides support for the force sensors. Real-time communication between computers and custom devices is handled by the RTnet interface.

5. Conclusions

The paper describes the specification and the resulting architecture of the *Rex* mobile platform control system. The methodology relying on the decomposition of an embodied agent into several standard subsystems, employing transition functions for the description of their behaviours, has been used for the designed system specification. The internal data structures of each of the subsystems have been defined. Finally implementation architecture has been proposed relaying on a set of well integrated hardware and software tools.

Within resulting control system architecture, motion planning and trajectory tracking tasks have been isolated and closed into separate components with well defined interfaces. This allows for independent development and testing of various algorithms, without disturbing other system components. Replacing one system element does not enforce changes in the others. Through the use of proposed integration tools, time needed for prototyping algorithms has been significantly reduced. The *Trajectory planner* and *Model-based controller* components will be a subject of further exchange. A trajectory planner implementing ECSA and OPCM motion planning algorithms, as well as a model-based controller implementing NMPC and AFM trajectory tracking algorithms will be investigated in the future. Other system components remain unchanged, unless the control paradigm is entirely changed.

Acknowledgements. This work has been done in the framework of the *RobREx* project funded by the Polish National Center For Research and Development grant PBS1/A3/8/2012.

REFERENCES

[1] C. Zieliński, "Specification of behavioural embodied agents", *Fourth Int. Workshop on Robot Motion and Control (RoMoCo'04)* 1, 79–84 (2004).

[2] S.H. Kaisler, *Software Paradigms*, Wiley Interscience, London, 2005.

[3] K. Sacha, *Software Engineering*, PWN, Warszawa, 2010, (in Polish).

[4] C. Zieliński, T. Kornuta, and M. Boryń, "Specification of robotic systems on an example of visual servoing", *10th Int. IFAC Symp. on Robot Control (SYROCO 2012)* 10, 45–50 (2012).

[5] T. Kornuta and C. Zieliński, "Robot control system design exemplified by multi-camera visual servoing", *J. Intelligent & Robotic Systems* 1, 1–25 (2013).

[6] C. Zieliński and T. Kornuta, "Diagnostic requirements in multi-robot systems", *Intelligent Systems in Technical and Medical Diagnostics*, pp. 345–356, Springer, Berlin, 2014.

[7] C. Zieliński, T. Kornuta, and T. Winiarski, "A systematic method of designing control systems for service and field robots", *19-th IEEE Int. Conf. Methods and Models in Automation and Robotics, MMAR'2014* 1, 1–14 (2014).

[8] C. Zieliński, "Transition-function based approach to structuring robot control software", *Robot Motion and Control*, *Lecture Notes in Control and Information Sciences* 335, 265–286 (2006).

[9] C. Zieliński and T. Winiarski, "General specification of multi-robot control system structures", *Bull. Pol. Ac.: Tech.* 58 (1), 15–28 (2010).

[10] C. Zieliński and T. Winiarski, "Motion generation in the MRROC++ robot programming framework", *Int. J. Robotics Research*, 29 (4), 386–413 (2010).

[11] K. Tchoń and J. Jakubiak, "Endogenous configuration space approach to mobile manipulators: a derivation and performance assessment of Jacobian inverse kinematics algorithms", *Int. J. Contr.* 76 (14), 1387–1419 (2003).

[12] D. Paszuk, K. Tchoń, and Z. Pietrowska, "Motion planning of the trident snake robot equipped with passive or active wheels", *Bull. Pol. Ac.: Tech.* 60 (3), 547–555 (2012).

[13] M. Janiak and K. Tchon, "Constrained motion planning of non-holonomic systems", *Syst. Contr. Lett.* 60 (8), 625–631 (2011).

[14] A. Ratajczak and K. Tchoń, "Multiple-task motion planning of non-holonomic systems with dynamics", *Mechanical Sciences* 4 (1), 153–166 (2013).

[15] K. Tchoń, M. Janiak, K. Arent, and Ł. Juszkiewicz, 'Motion planning for the mobile platform Rex", in R. Szewczyk, C. Zieliński, and M. Kaliczyńska, eds., *Recent Advances in Automation, Robotics and Measuring Techniques*, pp. 497–506, Springer, Berlin, 2014.

[16] M. Janiak and K. Tchoń, "Constrained robot motion planning: Imbalanced jacobian algorithm vs. optimal control approach", *Methods and Models in Automation and Robotics (MMAR), 15th Int. Conf.* 1, 25–30 (2010).

[17] B. Houska, H.J. Ferreau, and M. Diehl, "ACADO toolkit – an open-source framework for automatic control and dynamic optimization", *Optim. Control Appl. Meth.* 32, 298–312 (2011).

[18] M. Diehl, H.J. Ferreau, and N. Haverbeke, "Efficient numerical methods for nonlinear MPC and moving horizon estimation",

Nonlinear Model Predictive Control, Lecture Notes in Control and Information Sciences 384, 391–417 (2009).

[19] G.V. Raffo, G.K. Gomes, J.E. Normey-Rico, C.R. Kelber, and L.B. Becker, "A predictive controller for autonomous vehicle path tracking", *Intelligent Transportation Systems, IEEE Trans.* 10 (1), 92–102 (2009).

[20] M. Cholewiński, K. Arent, and A. Mazur, "Towards practical implementation of an artificial force method for control of the mobile platform Rex", *Recent Advances in Automation, Robotics and Measuring Techniques, of Advances in Intelligent Systems and Computing* 267, 353–363 (2014).

[21] A. Mazur and M. Cholewiński, "Robust control of differentially driven mobile platforms", *Robot Motion and Control*, *in Control and Information Sciences* 2011, 53–64 (2012).

[22] M. Diehl, H.G. Bock, H. Diedam, and P.-B. Wieber, "Fast direct multiple shooting algorithms for optimal robot control", *Fast Motions in Biomechanics and Robotics*, *Lecture Notes in Control and Information Sciences* 340, 65–93 (2006).

[23] C. Zieliński, A. Rydzewski, and W. Szynkiewicz, "Multi-robot system controllers", *Proc. 5th Int. Symp. Methods and Models in Automation and Robotics* 3, 795–800 (1998).

[24] J.V. Frasch, T. Kraus, W. Saeys, and M. Diehl, "Moving horizon observation for autonomous operation of agricultural vehicles", *Control Conf. (ECC), Eur.* 3, 4148–4153 (2013).

[25] M. Zanon, J.V. Frasch, and M. Diehl, "Nonlinear moving horizon estimation for combined state and friction coefficient estimation in autonomous driving", *Control Conf. (ECC), Eur.* 3, 4130–4135 (2013).

[26] H.J. Ferreau, T. Kraus, M. Vukov, W. Saeys, and M. Diehl, "High-speed moving horizon estimation based on automatic code generation", *Decision and Control (CDC), IEEE 51st Ann. Conf.* 2, 687–692 (2012).

[27] A. Geiger, J. Ziegler, and C. Stiller, "StereoScan: Dense 3d reconstruction in real-time", *Intelligent Vehicles Symp. (IV), IEEE* 1, 963–968 (2011).

[28] S.A. Mahmoudi, M. Kierzynka, P. Manneback, and K. Kurowski, "Real-time motion tracking using optical flow on multiple GPUs", *Bull. Pol. Ac.: Tech.* 62 (1), 139–150 (2014).

[29] Xenomai, *Real-Time Framework for Linux*, http://www.xenomai.org.

[30] ROS, *Robot Operating System*, http://www.ros.org.

[31] OROCOS, *Open Robot Control Software*, http://www.orocos.org.

[32] Real Time Engineers Ltd., *FreeRTOS*, http://www.freertos.org.

[33] RTnet, *Hard Real-Time Networking for Real-Time Linux*, http://www.rtnet.org.