Data and Event Handling for the MARIE Vehicle

A. Visser, G.D. van Albada, L.O. Hertzberger, Faculty of Mathematics and Computer Science, University of Amsterdam
and
G.A. den Boer, CMG (Computer Management Group), The Netherlands

95RO063

Abstract

In autonomous control systems the sensor-actuator control loop plays a dominant role. It is essential to give the system the capability to react to changes in its environment. A flexible autonomous system needs to reconfigure its sensor-actuator control loops depending on its surroundings, the task at hand, and the available sources of information. In the system architecture of the MARIE vehicle there is a clear difference between control and data flow. In this article we describe our general approach to realise the data interfaces between the several sensing- and actuation-modules. Information users can dynamically select their sources and monitoring is transparently supported. Also we describe how the modules can be controlled (initialised, parameterised, activated, suspended) from a higher abstraction layer through the so called elementary operation interface.

Introduction

System autonomy requires interaction between the system (robot) and its environment. On the one hand, the robot can move about and act on the environment, on the other hand, it observes the state of the environment and the effect of its own actions. These observations play a crucial role in the selection and execution of future actions. The robot will observe the environment and its own internal state through one or more sensor systems. Sensor data processing modules will process the raw observations, yielding a variety of data to be used for various purposes. Some data are used almost immediately for low-level actuator control, some data will be used as input for further analysis, some by high-level decision making processes, and some data are recorded for monitoring and performance analysis purposes only. The bulk of these data constitutes a continuous stream of information.

An autonomous system must be able to execute a range of tasks under various environmental conditions. For each task, a certain combination of sensory and control modules will be optimal in a given range of environmental conditions. When a task has been completed, whether or not successfully, or when a significant change in the environment has been detected, the components of this sensor-actuator control loop must be reconfigured. Such a transition must be effectuated by the high level task scheduling control components of the system on basis of information received from the sensory and control modules. This type of information does not constitute a continuous stream, but consists of individual distinct events.

The reconfiguration of the sensor-actuator control loop can result in short-term changes in the data flows between sensory modules and actuator control modules. Sensor failures, modifications in the system, etc., can cause similar changes on a longer time-scale. In such situations the correct operation of the system should not depend on the connection of a module to a particular source of data: as long as the appropriate information is available in the system, this information may come from different sources.

The above illustrates the two principal types of interaction between the components of an autonomous system - continuous data exchange, and event/control type interactions. For the MARIE vehicle two sets of interface functions have been implemented that support these interactions in a distributed processing environment. We have found that the use of these functions greatly simplifies the integration of new modules into the system. In this paper we will describe both sets of functions and our experiences when using them in the MARIE vehicle [1].

For the continuous data exchange, we have opted for the use of a structured intermediate storage, which we have called “data-managers”. For the event/control type interactions we have implemented an “elementary operation” interface. Both sets of routines were written for the vxWorks™ operating system. For the data-manager routines a compatible version was also implemented to run on the UNIX host, so that e.g. monitoring tasks can easily be implemented.

The data-manager interface

The planning and execution of actions by the robot system is based on the robot’s perception of its environment. This perception will be based both on static data provided by the operator at the start of the mission, and by dynamic data, acquired during the execution of earlier parts of the mission. These two together constitute a ‘dynamic environment database’. The data in this database should be organised in such a way as to facilitate access of similar data, e.g. obstacles, in a uniform manner, even when obtained from different sources.

Every perception task can conceptually be split into three parts:

1. obtaining the required input data,
2. the actual data processing, and
3. publication/distribution of the results.

The actual data processing is certainly specific for each task. The distribution of results, on the contrary, is quite similar for all tasks. Obtaining data can be a very process specific, if the input is gathered from real sensors, or quite general, if the input is gathered from processes that have pre-processed the data. In the latter case the perception modules are called ‘logical sensors’ [2]. The effort required to realise these input and output streams in the general case can, to advantage, be relegated to a specific data-management task leading to a blackboard-like communication structure [3]. In our system, the processes providing these data-management services are the data-managers. Each logical sensor requests and obtains the required data from these data-managers and publishes its output data as one or more records through a data-manager.

For a robot system it is attractive to use a distributed approach for the information architecture [4], with data areas supported by separate ‘data-managers’. Similar data will be stored together and can be queried in a single operation. This does not necessarily mean that all data in a single data area can be interpreted in quite the same way.

As an example, we will consider the data produced by an ultrasonic sensor system and by a stereoscopic vision system. The ultrasonic system produces distances to ‘features’ in the environment in a certain direction relative to the vehicle at a certain point in time [5]. The stereoscopic vision system does essentially the same, although the nature of the features is described in somewhat more detail (as 3D line segments) and with a better accuracy, at least in angle.

Both can be classified as features and can be used as such as input to the collision avoidance module (after a selection for relevance) [6]. Both can also be used as input for a process that builds a global map of the environment. Yet the data are different. These differences can be hidden in the data-manager. General features can be stored in a general class of sensor data, additional information can be stored in subclasses. For the latter, one can think of the 3D information provided by the stereo-vision system. Although various fundamentally distinct classes of sensor data may be recognised, the total number of such classes can be kept reasonably small.

As will be evident from the nature of the robot system, the data in the data-managers will not only be low level features, but also high level objects as walls, parking-spots, etc. The knowledge about these object has to change based on the observed features. The data-managers will not effectuate this change themselves, this task is done by various logical sensors. The data-manager supports the logical sensors with functionality to add, update, delete and select data on basis of various criteria. In order to facilitate a flexible retrieval of the data, each record is characterised in a number of ways: by a sequence number, by the task that stored it, by the time of storage, the time of the original observation, and by a ‘data class,’ describing the information contained in the record.

Data classes

As stated, the information contained in the data, is more important for the recipient than is the source of the data. Therefore, a classification method for the data was set up that allows a - limited - classification of the data by content. This classification is based on the assumption that there is a limited number of basic categories - or classes - of information relevant to a particular autonomous system. One such a basic class of information would e.g. be information about features in the environment. Such a feature would have an extent and a position at a given point in time, but much more information would not necessarily be available. For many purposes, such as collision avoidance, this information suffices. For other purposes, a more detailed description of the feature may be needed, without invalidating the basic information on the feature's position. In order to allow such information to be specified, two levels of sub-classes are allowed for every class.

The data records corresponding to each data class start with a general header, common to all records stored in a data-manager. The header contains a field describing the total length of the data fields. It is followed by the information particular to the class. When the record belongs to a sub-class, the sub-class specific information is stored after that for the larger class, etc. In this way routines that require only information belonging to the larger class can always access these data, whether or not more data belonging to sub-classes is present.

A single data-manager process can store data belonging to multiple classes. However, when the total amount of stored information becomes too large, it will be more efficient to create multiple data-managers, each responsible for a limited number of data classes. Data belonging to a single class should not be distributed over more than one data-manager.

The data-manager interface functions

Within one autonomous system, and even on a single processor node, there may be more than one data-manager. All data-managers run the same code and can have exactly the same interface and functionality as they do not interpret their data. The data-manager code is multiply re-entrant so that multiple data-managers can run as separate threads in a common memory space. At the operating system level, data-managers are distinguished by their processor node and port numbers; at a higher level they can be identified by a single system-wide identification number.

The software for the data-managers is contained in two parts. One part contains the private code for the server task, the other the public library functions for access to the data-managers, callable from client tasks.

For the communication between the data-manager servers and the client processes we have opted for a connection-oriented approach, as usually a repeated exchange of data will take place. All communication is initiated by the client tasks, and is implemented as a remote procedure call. After setting up a connection with a data-manager, a client can exchange data with that server using a connection descriptor. A function is available to switch the connection to another server.

For the exchange of data, functions are available to add a record, read a record by sequence number, and to delete a record. More powerful functions make it possible to replace a record by a new record and to purge all data older than a specified age in a given class. However, the most powerful function is the ‘dm_select’ function for retrieving data from a data-manager. It searches for all records matching specified criteria of class membership (inclusion and exclusion), source, age and sequence number and returns the number of records extracted or a negative error value. The client process can specify how many records it wants to receive maximally and how long it wants to wait if no matching records are available. This simplifies the construction of data-driven sensor and monitoring tasks.

On top of these functions, libraries supporting more complex data structures than single records can easily be constructed. One such library allows the processing of lists of records, another the processing of maps, consisting of an index record plus a collection of records describing poly-line segments.

The elementary operation interface

Perception and control tasks in an autonomous system can usually be described as data-driven, event-driven, or time-controlled. In our opinion, autonomous robot systems should be designed and implemented as a series of virtual machine layers [7]. One of the essential aspects of this approach is that, though the higher level layers control the activities of the lower levels by sending instructions, the lower levels execute these instructions autonomously. Figure 1 illustrates the levels in the operational architecture for the MARIE vehicle.

The operations are active components at the virtual robot level. They define the basic capabilities of the complete autonomous system Therefore, they were originally called ‘elementary operations’. At this level we find the control operations that allow the robot to follow complex paths in various environments. At this level we also find the network of (logical) sensors that together provide the required information about the environment and monitor the effect of the robot's actions. The exchange of information between the processes executing at this level is facilitated by the data-managers described in the preceding section.

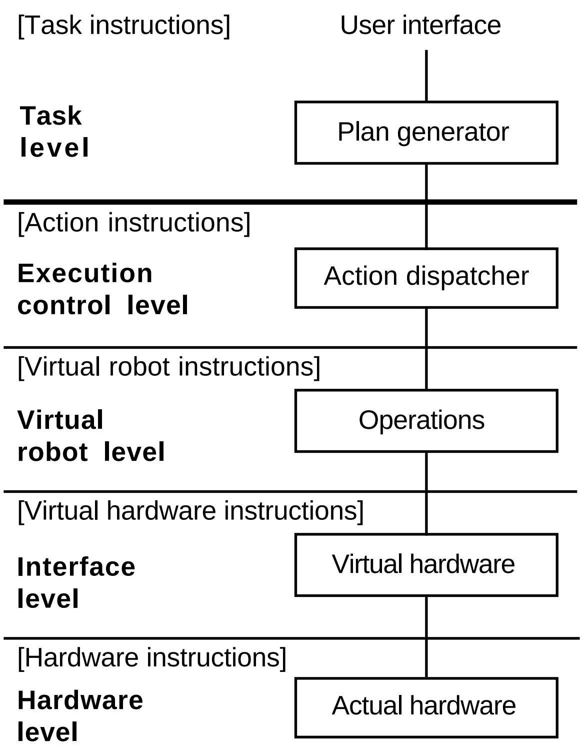

Figure 1. The operational architecture for the MARIE vehicle.

Virtual robot instructions constitute the instruction set of the virtual robot level. Its virtual machine is an aggregate of several modules: the operations. Each operation actually is a virtual machine in its own right, able to execute a subset of the virtual robot instruction set. Those instructions are executed concurrently with other operations. Each virtual robot instruction maps directly onto an operation. Furthermore, each has a corresponding set of parameters. The parameters define the exact behaviour of the operation they are associated with. At the virtual robot level there is no awareness of how operations are manipulated. Each operation runs without knowing about others that are also active. If information is needed to accomplish a goal, it simply is requested from a data-manager under the assumption that an operation that produces such information, is active. A set of concurrently executing operations is called an action.

The instruction set of the next higher level in the hierarchy – the execution control level – consists of actions, together with a set of conditions that must hold for the action to be executed and a prescription on how the outcome of the action affects these conditions. The virtual machine of the execution control level – the action dispatcher – will repeatedly evaluate the state of the system, select an action to execute, initiate its execution and update the state information on basis of the result. The action dispatcher is essentially event-driven: most of the time it is waiting for an action to complete. An action is considered to have completed when the first operation in that action signals completion.

The operations at the virtual robot level are mostly time-controlled and sometimes data-driven. After activation, they repeatedly do their thing and wait for a fixed time-interval or for new data to arrive. When they have completed their task – e.g. when a path has been completed, or when a specific object has been found in the environment – they signal the action dispatcher and wait for further instructions. Alternatively, their execution may be suspended by the action dispatcher when another operation in the same action has completed. The functionality required for the interaction between the action dispatcher and the operations is provided by the elementary operation interface. Different from the data-manager interface, the required functionality cannot be provided by a remote procedure-call paradigm. A more general message-passing paradigm is used instead. Even so, we will call the action dispatcher the 'client' and each operation a 'server'.

The implications for the functionality of operations are as follows:

- All operations must allow repeated activation/suspension by a different process.
  This is not the same as creation/destruction, which may be necessary only once for most operations.
- All operations must support reparameterisation in such a way that the behaviour of the operation is affected immediately.
- Every operation must be able to send a signal to the client process that activated it. This signal will indicate successful completion, specific conditions (such as detection of a specific feature in the environment), or failure. The signal should be sufficiently informative to provide the client process with the required information about the state of the system.
- Task completion in an operation should lead to suspension of that operation, not to termination. I.e. the operation should be able to receive new input and resume execution.
- To allow a distributed computer system to be used, the elementary operation interface functions have been implemented using sockets. The use of sockets implies the need to set up, and possibly to close, the connection between client and server.

The software for the elementary operation interface is also split in two parts: the client-side and the server-side functions.

On the client side, functions are available that allow the client to set up a connection to an operation sever and to sever that connection. Other functions send a new parameter record to the server, query its state, or activate or suspend the server. When all operations belonging to an action have been parameterised and activated, the action dispatcher will call the function 'eo_term_wait'. This function waits for one of the operations to signal completion or an error. This function also allows a time-out period to be specified and allows the user to send an interrupt from the console. Normally a call to eo_term_wait will be followed by suspend calls for the other operations in the action.

The functions implementing the server side of the elementary operation interface include 'eo_init', which initialises the server data structure for the operation, 'eo_get_params', which obtains the parameter record passed by the client, and 'eo_finish', which sends a message to the client task to signal that the assigned task was completed or aborted. All three routines return immediately - they do not require a reply from the client. The function 'eo_wait' provides the mechanism through which the operation receives instructions from the client process. It will cause the execution of the operation to be temporarily suspended, usually for an interval specified when the function is called. However, when the call is preceded by a call to eo_finish, or when the client has sent a 'suspend' instruction, the operation blocks until it receives a new parameter record or an activation command.

Experimentation

The two available interfaces to our software modules, one for instructions and one for data-flows, are thoroughly tested at the robot vehicle at our department. Several experiments were set up to test the integration and co-operation between different modules [8]. In this article we will walk through one of those experiments to illustrate the actual use of the data-managers and the elementary operation interface.

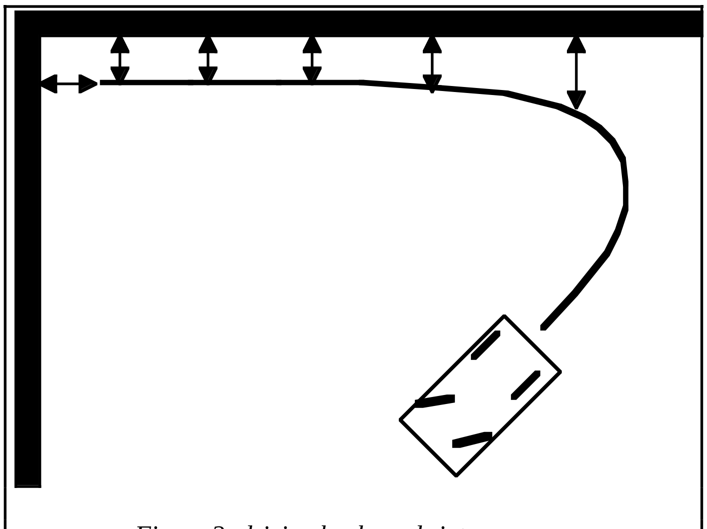
Figure 2: driving backwards into a corner

The vehicle is a four-wheeled robot, with kinematics similar to a normal automobile. The goal of the mission we will describe is to recalibrate the position of the robot relative to a certain well-known location in the room: one of the corners, as shown in figure 2. For this calibration the assumption is made that the orientation of the vehicle is known with an accuracy of 10˚, and that the position error perpendicular to the driving direction of the car is larger than the position error in the driving direction. The mission can be divided in the following steps:

- first a backward path is planned from the current location to the corner.
- the planned path is driven, until a wall is found on one of the sides of the robot. A collision avoidance module is active to turn the vehicle away, if the wall or an other obstacle appears behind the vehicle.
- the vehicle aligns itself along the wall, the collision avoidance module is still active to improve the performance of this algorithm.
- the vehicle follows the wall, until an obstacle is found at a precise distance in the back of the car.

Each step is called an action. During the first action only one operation is active, an on-line planning operation. During the last three actions multiple operations are active concurrently. One operation is responsible for the control of the vehicle; several sensing operations are active simultaneously. While driving the locations of obstacles and walls relative to the vehicle are updated continuously. The operations called ‘collision avoidance’ and ‘wall sensor’ are responsible for the maintenance of this information. The ‘collision avoidance’ operation not only gathers the information about obstacles, it also modifies the control signals for the vehicle. It does not actually control the vehicle, but guarantees that the control signals sent to the vehicle are safe with respect to obstacles.

The collision avoidance and wall sensor do not acquire their information directly from the physical sensors, but read it from a data-manager. Two other operations are responsible for the acquisition of that data; one module controls a set of 8 narrow beam (20˚) sonar sensors, and the other module controls a set of 12 wide beam (60˚) sonar sensors. The wide beam sensors have good characteristics to detect the presence of objects in a range from 0.2 to 2 m in a full circle around the vehicle. The narrow beam sensors have good characteristics to measure the location and shape of big objects in the environment of the vehicle, although there are not enough sensors available to do this in all directions. The two sonar systems have completely different characteristics, but for the collision avoidance and wall sensor this is hidden by the data-managers interface.

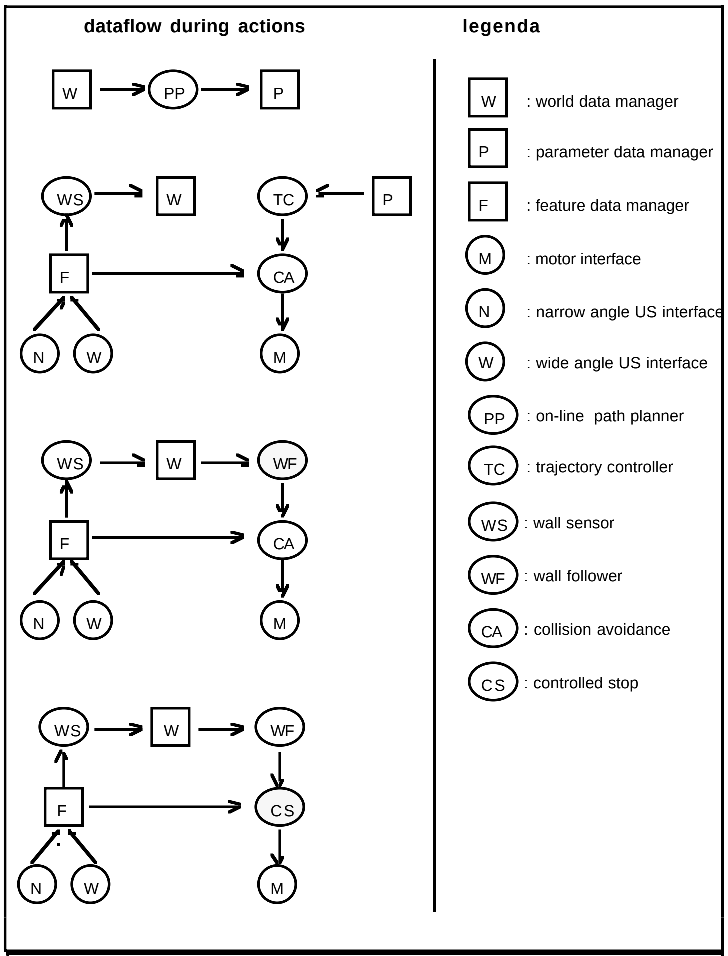

Figure 3: the different data flows during the actions in our example

Experiences

The combination of data-managers and the elementary operation interface provides the programmer with a set of tools for constructing perception and control operations in a straightforward manner. As the tools have been designed from a certain view on how such operations should function and co-operate, they also more or less enforce adherence to this view.

One of the intended limitations of the elementary operation interface is that an operation is not allowed to activate another operation the services of which it requires. This scenario is not encouraged, because this also means that a suspension command has to propagate down through a chain of operations. In our architectural model, all operations have to be activated simultaneously by the action dispatcher. The rationale behind this structure is that in a modular and possibly changing system configuration, it is unattractive to distribute knowledge of the system structure among all operations. We prefer to limit this knowledge to the action dispatcher or the planning module.

One of the intended limitations of the data-manager interface is the limited query possibilities of the function dm_select. In our approach the client has to know quite explicitly what it wants from the data-manager. This assumption is not as bad as its sounds, because in most cases the information is provided by a limited set of producers, for a limited set of consumers. In many other real time control systems the modules are explicitly coupled to each other. Then not only the format of the data is explicit, but also the connection. Our system is a little more flexible, but still with explicit queries regarding data-formats. The price for a more intelligent query-system will be performance, a price that cannot be paid in a real time system.

A variety of sensor and control functions have been implemented for the MARIE vehicle using the data-manager and elementary operation interfaces. We have found it quite straightforward to implement our own functions in this way. Importing software, which was developed elsewhere, proved to be somewhat more difficult, but yet not too complicated. This gives a good indication of the generality of our approach, because the imported software was not always structured in the way we had intended.

Conclusions

The use of two separate interfaces for the essential interactions between the components of an autonomous robot control system – data-managers for the exchange of information between peer modules, and the elementary operation interface for the initialisation and activation of operations – provides a clear and well structured framework for the implementation of the components at the virtual robot level and the execution control level.

References

[1] G.A. den Boer, G.D. van Albada, L.O. Hertzberger, C. Koburg, and M. Mergel "The MARIE autonomous robot" Intelligent Autonomous Systems IAS-3, F.C.A. Groen, S. Hirose, C.E. Thorpe (eds.), IOS press, Amsterdam, 1993.

[2] T.C. Henderson, and E. Shilcrat "Logical Sensor Systems" Journal of Robotic Systems 1(2), 1984

[3] B. Hayes-Roth, "A blackboard architecture for control", Artificial Intelligence 26, pp. 251-321, 1985.

[4] C. Thorpe, M. Herbert, T. Kanade, S. Schafer, "Towards autonomous driving: the CMU Navlab Part II - Architectures and Systems", IEEE Expert 1991.

[5] B.J.A. Kröse, K.M. Compagner, F.C.A. Groen, "Accurate estimation of environment parameters from ultrasonic data", Intelligent Autonomous Systems IAS-3, F.C.A. Groen, S. Hirose, C.E. Thorpe (eds.), IOS press, Amsterdam, 1993.

[6] G.A. den Boer, G.D. van Albada, L.O. Hertzberger, G.R. Meijer, J.-B. Thevenon, P. LePage, E.J. Gaussens, and F. Arlabosse, "An exception handling Model applied to Autonomous Mobile Robots" Intelligent Autonomous Systems IAS-3, F.C.A. Groen, S. Hirose, C.E. Thorpe (eds.), IOS press, Amsterdam, 1993.

[7] L.O. Hertzberger, G.D. van Albada, G.A. den Boer, "Information architecture concepts for autonomous control", Intelligent Autonomous Systems IAS-4, U. Rembold, R. Dillmann, L.O. Hertzberger, T. Kanade (eds.), IOS press, Amsterdam, 1995.

[8] G.A. den Boer, "A control architecture for the MARIE autonomous robot", PhD thesis, University of Amsterdam, 1995.