Improving control of the MARIE robot

Improving control of the MARIE robot

Master Thesis

Koen Böinck, 9443266
University of Amsterdam
kboinck@science.uva.nl

September 2003

ABSTRACT

This thesis describes an idea to enhance the control over the mobile autonomous robot MARIE. MARIE is a complex, software controlled autonomous robot, that is capable of versatile behavior, such as maneuvering in a complex environment. Control decisions are made by a central process. This process controls the execution of a variety of tasks, that the robot can do, by instructing, differently controlled, elementary operations. This requires flexibility on part of that central control process. The downside of this flexibility is that control is enforced in a generic fashion, and the central controlling process can't take too detailed decisions, because it lacks the information to do so.

This is solved, in this thesis, by introducing location independence of all the software components of the robot, and introducing a property interface that allows for the communication of the properties of all these modules.

Location independence makes it easier, for the developer and user of the robot, to carry through changes in the software or its configuration. Extending location independence with the ability to talk about properties of the robot's controlling elements creates room for more autonomous behavior. The property interface provides the required control details about the actuation, sensing and reasoning components of MARIE, in generic and broadly applicable way, so that high-level functionality such as self-configuration or self-optimisation become possible.

PREFACE

Dank...dank...

INDEX

1 INTRODUCTION

1.1 General

1.1.1 Autonomous Systems

Autonomous systems are computational devices that need to operate in an environment that is similar to that in which humans operate. The devices' ability to adapt to unforeseen situations or changes in the environment make it autonomous.
The fundamental goal of research on autonomous systems is to have the best behavior with as little human specified directions as possible. This is, generally, accomplished by specifying the task a robot must perform in terms of parameters or properties of the world the system resides in. The system will need to translate the task description into internal parameters, that it can evaluate or reason about and that control the robot.
There are three main processes that lie at the basis of a (mobile) robots autonomy:

- Sensing; perceiving the environment the robot resides in.
- Reasoning; interpreting and analyzing perceived data.
- Actuation; undertaking actions to change the world state, based on perception and reasoning.

An autonomous robot's behavior is the result of the relationship between these three processes: actuation changes the world state, which the robot senses, which in its turn influences reasoning. An autonomous mobile robot continuously loop through these tasks, allowing it to perceive, act on, and adapt to its changing environment.
These processes are performed by software that controls the robot's hardware. The robot's software is usually divided in different functional components, the cooperation of which causes the robots behavior. One of the developments in this area is involved with the structuring of these software components, i.e. the software architecture. In literature on autonomous systems a number of proposals for the software's architecture can be found:

- Functional; hierarchical decomposition of software components, based on functionality.
- Behavioral; components are separated on basis of specific behavior they should exhibit.
- Hybrid; a combination of functional and behavioral architectures.

The hybrid form is currently the most acceptable architecture, in literature. During the nineties the focus on research on autonomous robot software moved more towards the cooperation of, and controlling the various software components within a given software architecture. This master thesis is about enhancing the control of the sensing, actuation and reasoning components in a mobile robot, called MARIE.

1.1.2 The MARIE autonomous mobile robot

The research in this master thesis concerns an autonomous mobile robot called MARIE. MARIE is an acronym for Mobile Autonomous Robot in an Industrial Environment. MARIE is a mobile autonomous robot, that was developed in order to research computer-integrated manufacturing. The project was started in 1989. The primary goal of the research was to investigate symbolic and numerical control methods for autonomous systems.
The integration of symbolic and numerical techniques enabled the robot to meet three major control objectives:

- Comprehending a complex environment,

- Maneuvering within this complex environment, and
- Carrying out tasks in exceptional situations.

The robot is built from a Vesa Trekka cart for disabled people. It contains a steering and a driving motor, an on-board computer with the controlling software and, recently, a wireless network connection to other computers.

At present there is no research being done to have MARIE operate in an Industrial Environment, which was the original purpose of the MARIE project. Despite the project not being finished, it has left the department with a fully functioning, state-of-the-art, mobile autonomous robot and thus there is still plenty of opportunity to perform research on it. Throughout the years different people have conducted research using MARIE as a test bed. Ph. D. George de Boer has investigated the architectures for intelligent robots, and described in detail its application on MARIE [1]. M. Sc. Frank Terpstra [2] has developed an on-line planner for MARIE, with considerable success. M. Sc Erik de Ruiter [3] has finished work on structuring input and output of the system combined with version management, in order to learn future control parameters and configuration settings.

1.1.3 Flexibility

MARIE was developed in a modular sense. Meaning that some kind of abstraction was thought of to divide the desired functionality into several components, additionally, interfaces were developed to have the components interact. The success of this modular approach was (besides usual software design demands) needed because the development of MARIE took place in different countries, with the intention to merge (the best functioning) developed components. At every research site different hardware was used. For the planned merging to take place all software components had to operate with 'foreign' software and be abstract enough to be applied on the other hardware platforms. This approach led to success; it has resulted in a fully functional mobile autonomous robot system. However, the flexibility also had a price; controlling the software of MARIE suffers from its size. Control decisions are made by a central module. This module is flexible, because it can deal with the high-variation in specific control details of the actuation, sensing and reasoning software components, allowing for MARIE's versatile behavior. The downside of this generic control method is that it is not able to adjust to changes in the components that this central module controls, as will be explained in the next section.

1.1.4 The downside of the current system

The software of the MARIE robot unifies the best methods available on the subject of software architecture at the moment. Both a hierarchical and a behavioral architecture philosophy are applied, resulting in a versatile robot, capable of solving various 'complex' tasks [1].
The software of MARIE can be divided in two parts. One part is concerned with translating the human supplied mission description into subtasks that the robot can understand. The other part of the software controls and executes these subtasks. The execution of these subtasks is done by a virtual representation of the robot. The actual behavior of MARIE is a direct result of the behavior of this virtual robot. All sensing, actuation and reasoning of MARIE is performed in this software layer. It contains a collection of relatively simple software elements, called elementary operations, that each perform one specific tasks. For example, there is an elementary operation that reads and stores ultra sonic sensor values, or there is another elementary operation that assembles line segments from the ultra sonic data or yet another that plans a path.

The presence and location of each of the modules of the virtual robot has to be known at the time when task execution takes place, else how can they be controlled in the first place? This was accomplished by having locations and configurations be constant and that compromises the systems extensibility and maintainability, because as a result the software had to be treated as a monolithic whole, making it much harder to treat processes separately; aspects such as changing configurations or debugging (new) modules require a lot of effort.
Another issue, that arises when one has to control the various elementary operations, is that all the operations are controlled differently. This was partly solved by introducing an interface that *generalizes* control. It defines certain functionality that every controlling component has to posses. The components will thus communicate in an abstract 'instruction language'. This language is suitable enough to control the components, but the operator has to supply the factual instruction-details and since there is a big amount of instructions this makes operating the robot a time-consuming job, consisting of a repetitive and simple task. As a result, since the higher-level modules of the robot do not have the means to determine the specific control details for the virtual robot, high-level functionality, such as self-configuration, self-healing or self-optimization are harder to realize.

1.2 The proposition of this thesis

MARIE is capable of accomplishing high-level tasks such as finding a docking a spot and parking in it, while avoiding obstacles. These high-level tasks are described, by a human operator, in a mission description. Such mission descriptions contain high-level instructions, that tell the robot when to perform what task, it also contains specific control details that tells how a task is to be performed.
The ultimate goal for MARIE, or autonomous mobile robots in general, is to have the system perform complex tasks, with as least human interference as possible. In case of MARIE, it should be sufficient for the human operator to just tell the robot to "park in the nearest docking spot", without supplying any extra information. This however is far from how it is actually done; besides high-level task descriptions, the human also provides control specific details, such as the docking spot size. Rather than having a robot, that relies for goal satisfaction, on detailed, human supplied control instructions, the robot would be more 'intelligent' if it were able to obtain these specific control details on its own.
This thesis should bring MARIE a step closer to this ultimate goal; a method that increases the robots *self-knowledge* is introduced by extending the robots controlling software elements with functionality to supply their properties to higher-level system modules, so that on a higher-level it is known how the virtual robot is controlled. In this way more autonomous behavior becomes possible.
The instruction-details are different for every controllable software element. This means that, if one wants to communicate the capabilities of these elements, the interface has to be generic enough to accommodate all of them, but specific enough to communicate all details. Furthermore, publishing ones capabilities doesn't necessarily mean they can be accessed. That has to be accomplished as well.
A new interface will be designed to facilitate these demands. As a result the system is expected to decrease the work the operator has to spend on creating and providing mission descriptions, without the system performing poorer, of course. Furthermore the robot is expected to have a higher degree of autonomy since a system, which is aware of its control details can better adjust to unforeseen situations (environments), and perform better.

This leads to the formulation of the following hypothesis:
The property interface and location independence will

- Decrease the human interaction with MARIE, and
- Increase MARIE's autonomy

In the rest of this thesis the problem is described in more detailed, a solution is devised and the hypothesis are tested by means of experimentation.

1.3 Structure of this document

The theoretical background of this thesis will be treated in chapter 2 by discussing four articles that aim at using heterogeneously controlled elements in a distributed environment, just like the controlling elements of MARIE. Chapter three will explain how MARIE works. It gives a description of the various development stages of MARIE's software.
In chapter 4, the drawback of controlling the robot, will be described. Chapter 5 considers existing solutions to this problem, it discusses the applicability of the literature examples from chapter 2, and proposes a final solution.
Chapter 6 describes the solution in more detail, explaining how it is will be developed.
Chapter 7 gives the design for location independence, of which, in chapter 8, the implementation will be provided.
Experiments were devised, that test the hypothesis, by investigating if the proposed changes to the robot actually improve the system, this is described in chapter 9.
Chapter 10 contains a general discussion concerning the (improved) robot, and chapter 11 concludes this thesis.

2 RESEARCH CONTEXT

2.1 Introduction

The subject of this thesis is to improve the control over a collection of distributed components of the MARIE robot that require specific control information. When distributed systems become sufficiently large and versatile the need arises for a communication framework that defines how to obtain knowledge about distributed components and have them cooperate. This, and its relevance to MARIE are explained briefly in the overview of section 2.2. Section 2.3 gives an example of a system that specializes in locating and using distributed services. Section 2.4 gives an example of a system that controls heterogeneously controlled, statically located, elements. Section 2.5 describes a specific language for component description and section 2.6 describes, in more conceptual form, a design methodology and architectural concepts for collections of systems to manage themselves more autonomously.

2.2 Overview

The increase of MARIE's autonomy is expected to be accomplished by extending the robot with more self-knowledge, so that it has the information to make high-level decisions, that are now made by the human operator of the robot. The emphasis is primarily on self-configuration, but self-management in general is kept in mind.
This issue is a typical AI issue [8], [13]. In literature various software systems are developed that deal with decreasing the role of humans in controlling and managing systems. The papers that are discussed in this chapter were selected because they deal with the specific issue of controlling heterogeneous components in a networked environment That is (as will be seen in the next chapters) an important obstacle to overcome because the controlling components of MARIE are also heterogeneously controlled and are spread out over a network. In order to use these components, they must be identified and located and, because each component is controlled differently, control information needs to be obtained.
Despite the difference in scale (MARIE being relatively small) the examples are applicable because of the similar problem they intend to solve. The first two examples focus on localization of distributed components in a system. The other two examples deal with system control; LARKS is an example of the use of middle-agents and the capability language that comes with such an approach. And in the fourth example, a more towards the future view will be described, that shows a general approach to designing self-managing, complex control systems.

2.3 Jini

The goal of Jini is to turn a network into a flexible, easy to use tool on which resources can be found by clients and users [9]. Resources can be anything from hardware resources to programs.
Jini is an extension on Java, bringing along the machine independence and the ability to communicate data and code from machine to machine.
The Jini system consists of the following parts:

- A set of components that provide an infrastructure for federating services in a distributed system
- Services that can be made part of a federated Jini system and which offer functionality to any other member of the federation
- (a programming model that supports the production of reliable distributed services)

The most important concept within the Jini architecture is that of a service. A service may be a computation, storage, a communication channel to another user, a software filter, a hardware device, or another user.
Members of a Jini system share access to services. A Jini system consists of a collection of services that can be used together to perform a task.
Services are found and resolved by a *lookup service*. The lookup service provides the major point of contact between the system and the users of the system.
A service is added to a lookup service by two protocols, called *discovery* and *join*. The first locates an appropriate lookup service, the second joins the service. Communication between services is done by remote method invocation or RMI.

The infrastructure of Jini consists of

- The discovery and join protocols, that allows services to discover, become part of, and advertise its service to other members of the federation.
- The lookup system, which serves as a repository of service

The entries in the lookup system are objects that can be downloaded as part of a particular lookup operation and act as local *proxies* to the service that placed the code in the lookup service.

Dataflow
At the heart of the Jini system are three protocols: discovery, join, lookup. Figure 2-1 depicts the dataflow.

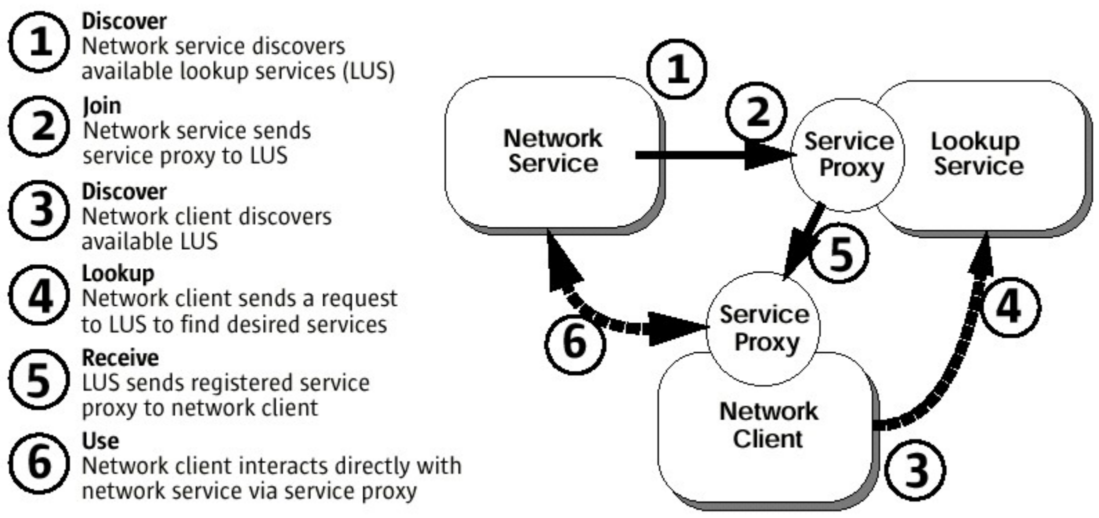

1 Discover
Network service discovers available lookup services (LUS)
2 Join
Network service sends service proxy to LUS
3 Discover
Network client discovers available LUS
4 Lookup
Network client sends a request to LUS to find desired services
5 Receive
LUS sends registered service proxy to network client
6 Use
Network client interacts directly with network service via service proxy
Network Service
Service Proxy
Lookup Service
Service Proxy
Network Client

Figure 2-1: Jini dataflow

Discovery occurs when a service needs to find a lookup service with which to register, join occurs when a lookup service is located and it wishes to join. Lookup occurs when a client or user needs to locate a service. This requested service is described by its interface and possibly other attributes. The data that is send to the lookup service consists of a service object. The service object contains the interface for the service, including the methods that applications will invoke to execute the service, along with any other descriptive attributes.

2.4 Hive

2.4.1 Overview

Hive is a distributed agents platform, a decentralized system for building applications by networking local system resources [15]. Its architecture concentrates on the idea of an 'ecology of distributed agents'.

Hive provides

- ad-hoc agent interaction,
- ontology's of agent capabilities,
- mobile agents, and
- a graphical interface to the distributed system.

Hive consists of three components: cells, shadows, and agents. In Hive an agent is located in a particular place, called a *cell*, and uses various local resources, called *shadows*. These shadows encapsulate capabilities such as a screen display or a digital camera. Agents are hosted on a cell and are meant to communicate with each other to share information and to access resources. An application is made from the communications and actions of agents. The ecology of distributed agents is a decentralized system, this means that agents are responsible for locating the resources they need, finding each other, and negotiating their relationships.
A cell is like a kernel, shadows are like device drivers, and agents are like processes.

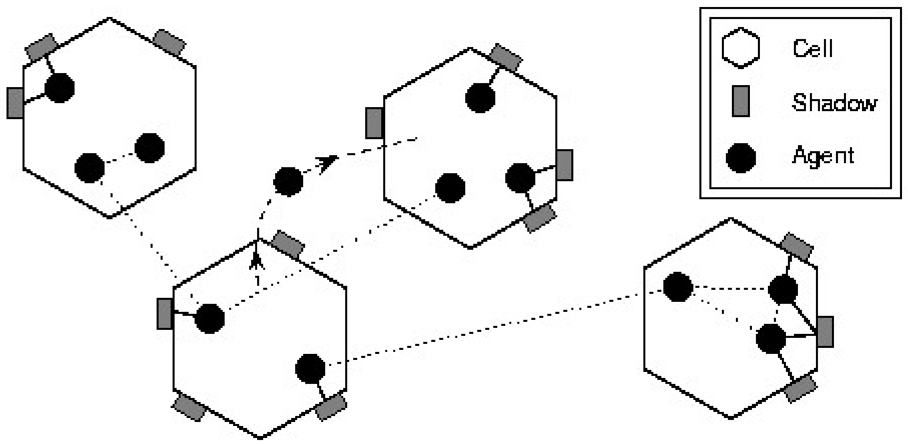


Figure 2-2

Of concern is the ability of the agents in Hive to represent the ontology of their capabilities.

2.4.2 Cells

A cell is a program that performs two tasks: hosting software agents and managing access to local resources through shadows.
Hive has a *location dependent* model of a distributed system. Hive cells are different from each other: each cell has a specific set of shadows and a specific population of agents. Devices are accessed by contacting an agent on the cell that has access to the shadow of that

device. The HIVE model does not ‘remove’ the concept of location, in contrast to other distributed systems.

2.4.3 Shadows

Physical devices are `shadowed' in the Hive cell. For example, a Hive cell may have a digital camera plugged into it, a shadow then contains the functionality to use that camera, simply by providing an API to access it.
The collection of shadows are the static part of a Hive cell. The behavior of a device is represented in the shadow, by the developer. The flexibility of the system comes from the mobile agents; the shadows provide the functionality which the agents access.

2.4.4 Agents

Hive agents embody the network interface and policy for resources.
Agents live on specific cells, accessing shadows for the resources they need. Agents export selected functionality to the network and communicate with each other to share those functions. For example, a camera agent can export the picture taking functionality of a camera shadow to remote agents. An image displayer agent can then invoke this method over the network, implementing a simple remote picture taking application. Hive applications are built out of a collection of interacting agents.
There is no requirements for agent communication, it is up to individual agents to decide how to talk to each other, making agent interaction ad-hoc.

2.4.4.1 Agent Description

Every agent in Hive is described in terms of two ontologies: syntactic and semantic.
The syntactic ontology of agents in Hive is simply its Java type.
Hive uses a second ontology to describe ‘semantic’ information about agents. This ontology utilizes the Resource Description Framework (RDF) [14]. RDF provides a structured way to attach nouns and verbs to agents. For example, an agent's semantic description might state its physical location, a human readable nickname, the owner of the device it is using, and a description of the meaning of its data. It is expected that application designers will develop their own schemas to make agent communication semantically consistent.
In HIVE, the agents capabilities and their meaning are explicitly stated in an ontology.
On request an agent can communicate its capabilities. The requester agent can then use that information to make decisions.

2.5 LARKS

2.5.1 Overview

Because the amount of services and deployed software agents in the internet is increasing rapidly there is an growing demand for automated search and selection of relevant services or agents. In order for heterogeneous agents to cooperate effectively across distributed networks of information, they must be able to communicate with each other using a common language.
A paper, called “Dynamic Service Matchmaking Among Agents in Open Information Environments”, describes research done, to solve this problem. It introduces a language and the process of matchmaking of agents in an open environment, like the internet [10], [11]. By means of a capability description language, called LARKS (“Language for Advertising and Request for Knowledge Sharing”), agents are meant to locate particular services for other services or agents.

The proposed approach defines three categories of agents:

- *Service providers*, proving some kind of service, such as finding information
- *Service requesters*, that need some service provider to perform some service for them
- *Middle agents*, agents that help locate others

Matchmaking or *brokering* is the process of finding and appropriate provider for a requester through a middle agent. This has this general form:

1. Provider agents advertise their capabilities to middle agents
2. middle agents store these advertisements
3. a requester asks a middle agent if it knows a provider that can provide certain information
4. the middle agents matches the request against the stored advertisements and returns the results

A common language is used by matchmaking agents to pair service-requesting agents with service-providing agents that meet the requesting agents' requirements.

2.5.2 The Language

The process of matchmaking is complicated by the fact that providers and requesters are heterogeneous, spread out over the internet and don't understand each other. This is solved by creating a common language that makes it possible to describe capabilities and requests in a uniform manner.
Three of the main features this language posses are:

- *Expressiveness*. The language should be expressive enough to represent data, knowledge and meaning
- *Inference*. Inference must be possible with a description in the language, enabling automated reasoning and comparison of descriptions.
- *Ease of use*. Description should be easy to use, read and write by the user.

The agents capability description language developed in this case is called Language for Advertisement and Request for Knowledge Sharing, or LARKS. Below is what a specification in LARKS looks like:

Context	Context of specification
Types	Declaration of used variables
Input	Declaration of input variables
Output	Declaration of output variables
InConstraints	Constraints of input variables
OutConstraints	Constraints of output variables
ConcDescription	Ontological description of used words
TextDescription	Textual description of specification

The slots have the following meaning:

- Context: context of the specification in the local domain of the agent
- Types: Optional definition of the data types used in the specification
- Input and Output: Input/output variable declaration.
- InConstraints and OutConstraints: logical constraints on the input/output variables. Described as Horn clauses.

- ConcDescription: Optional description of the meaning of words in the specification. The description relies on concepts in a given domain ontology and links words, occurring in the above slots, to this concept defined in the ontology.
- TextDescription: optional textual description of the specification.

2.5.2.1 Local domain ontology

LARKS uses local domain ontologies to describe the meaning of words from the specification. This way it offers the possibility to use domain knowledge in advertisements and requests. What concept language to use to write the ontology in is up to the developer. The benefits of using domain ontologies are twofold:

- The user can specify in detail what is being requested or advertised,
- The matchmaker is capable of making inferences on descriptions

2.5.3 The Matchmaking Process

The matchmaking process uses several methods for computing syntactical and semantic similarities between agents capability descriptions. It does this by providing 5 filters. The different filters provide different techniques to calculate the relevance between specifications, when the resulting value exceeds a certain threshold the specification pass the filter. Different combination of filters give different results. The 5 filters are *Context Matching, Profile Comparison, Similarity Filter, Signature* and *Constraint filters*.

LARKS and the matchmaking process present a good balance between quality and efficiency of matchmaking between heterogeneous software agents on the internet, due to the language's expressiveness and its efficient matchmaking with filters.

2.6 Autonomic computing

IBM has taken the integration of various complex and differently controlled systems to a next level by introduction of a concept called autonomic computing, that consists of a more general approach towards design of system integration and system cooperation in a dynamic, open environment [4], [6], [7].
In 2001 the corporation published a manifesto describing an increasing demand for systems to become better at self-management.
IBM suggests an approach involving *autonomic computing* and *self-repairing systems*. Systems must be developed that can run themselves, can adjust to varying circumstances and changing demands, and are robust with respect to damage and degradation.
The essence of this autonomic computing approach is self-management. The goal is to create a system that decreases the system administrators interaction with it and provide users with an optimal performing system.
The manifesto sites four aspects of self-management:

- **Self-configuration**, the automated configuration of components and system, following high-level wishes
- **Self-optimization**, components continually seek to optimize their (cooperate) performance.
- **Self-healing**, components automatically detect and repair from errors.
- **Self-protection**, system automatically defends against malicious attacks.

These autonomic systems will consist of autonomic elements, that are individual components that deliver services and contain resources and that will interact to accomplish the above mentioned goals, with and each other and the users of the system.

They will manage their internal behavior, as well. Such an autonomic element will consist of an autonomic manager that manages elements contained therein, that monitor, analyze, plan, and execute the managed element. See figure 2-3.

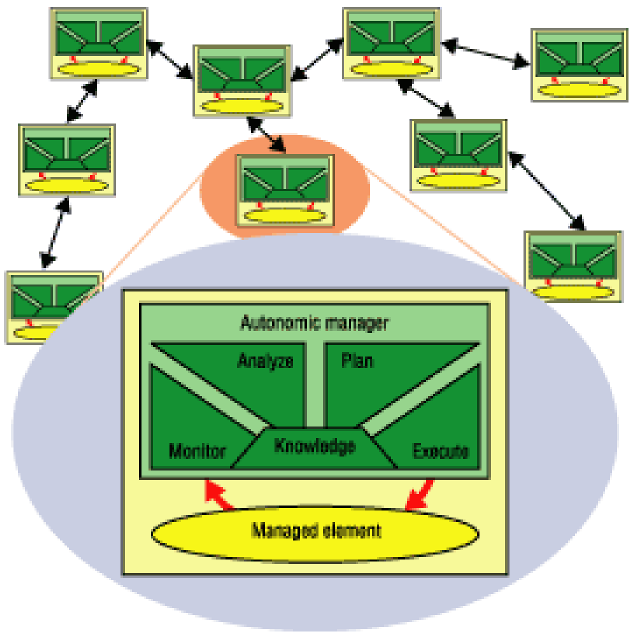


Figure 2-3: Structure of an autonomic element

The managed element could be something high-level as an application service or as low-level as a hardware component.
Each autonomic element is responsible for managing its own state and behavior with its environment, possibly by interacting with other autonomic elements.
Higher level autonomic elements will use low-level autonomic elements to achieve their goals. Goals that may be described in terms of the goals of these low-level elements.
Viewing autonomic elements as agents and autonomic systems as multi agent systems makes it clear that an agent-oriented approach will be important. There are no further details about the cooperation of the various elements, besides the recommendation that this should be done by some existing agent-software system.

2.7 Conclusion

Throughout the years more and more systems have been developed for varying purposes. In the last decade, or so, the need for these systems to cooperate has increased drastically. Considering the fact that most of these system reside somewhere on the internet, the above observation presents two problems, that are also relevant for MARIE. In order to control the different components (or systems) one needs their location and control information. This issue is a hot topic in the world of computer science and more specifically in the field of artificial intelligence. Connecting and cooperation of various, differently controlled, distributed systems is tackled subject of a vast number of software applications. This chapter described four of these. The first two dealt primarily with the distributed nature of the to-be-controlled components. The third example deals with describing components. The fourth example looks at self-management of systems, in general.

Jini and HIVE are interesting because of the way they deal with the location information of distributed components and are less concerned with providing specific control details for these components. In Jini services are located by looking them up with a lookup service, where the service has registered its location and additional control information.
HIVE uses fixed locations for its distributed components. Mobile agents serve as representatives of the component to interface with other agents. HIVE makes use of a (clearly separated) syntactical and semantic descriptions of its agents.
LARKS is an agent capability description language and serves as an example of how control information of (distributed) components may be represented. Great care was taken to include semantic and reasoning information in the components' descriptions. It is considered useful because it gives a nice picture of how a component's control information may be represented.
IBM's autonomic computing concept is a good example of one of the major goals of AI, namely making systems manage themselves. The self-management of a system is defined by its ability to configure, heal, optimize and secure itself. This would be accomplished by developing or extending a system (or its components) with monitoring, analysis, planning and executing elements.
Section 5.3, of the solution chapter, will discuss what concepts of the above mentioned techniques are useful for MARIE.

3 MARIE

3.1 Introduction

The autonomous nature of a given system is derived from the ability to solve problems in a varying environment. MARIE is autonomous in that sense, since it is capable of finding solutions to unforeseen situations. Solutions for relatively simple problems, like obstacle avoidance, have been shown to work and solutions for more difficult problems like maneuvering in a changing environment, are also performed successfully.
This chapter will explain how MARIE works. The three architectures that lie at the base of the system of the system are explained in sections 3.2.2, 3.2.3, 3.2.4. Section 3.2.5 explains the interfaces of MARIE, section 3.3 shows how MARIE is used.

3.2 MARIE Architecture

3.2.1 Overview

The goal of these sections is to explain how MARIE works. The software architecture that was designed for MARIE will be described. Some explained aspects bare more relevance on the issues of this thesis, than others, but it is assumed that an over-all perspective of the system is essential for understanding the motivation that led to the proposed changes.
There are three architectures, a functional, operational and implementational.

- The functional architecture defines the requirements for the robot. It is of a conceptual form, allowing it to be reused for other projects.
- The operational architecture describes how to realize the requirements from the functional architecture. In this step the over all problem is divided in components that must solve sub-problems, consequently, this is where design decisions are made.
- The implementational architecture describes in detail how these components must be implemented. It adds details regarding the technical limitations of the practical circumstances the robot and its software will be used in.

In this thesis some changes will be proposed that, mostly, affect the operational and implementational architecture.

3.2.2 Functional Architecture

To achieve its overall goals two basic systems were defined for MARIE, a Plan Generation system and Plan Execution system.
The Plan Generation system must produce a plan or task tree. A *human operator* and a logical module are contained in the system. Together, they interactively construct a task tree.
The operator supplies symbolic information, structured in a tree. The module translates this to a tree in a numerical format, suitable for the Plan Execution system. The process of decomposing symbolic instructions into a series of activities consists of task planning, scheduling and parameter planning.
The Plan Execution system receives the numerical tree with activities. It is responsible for the trees executing. Each activity consists of zero or more perception, on-line reasoning or control operations. To coordinate the proper execution of the activities and handle exceptions, execution control functionality was included. Figure 3-1 depicts the systems and their interaction.

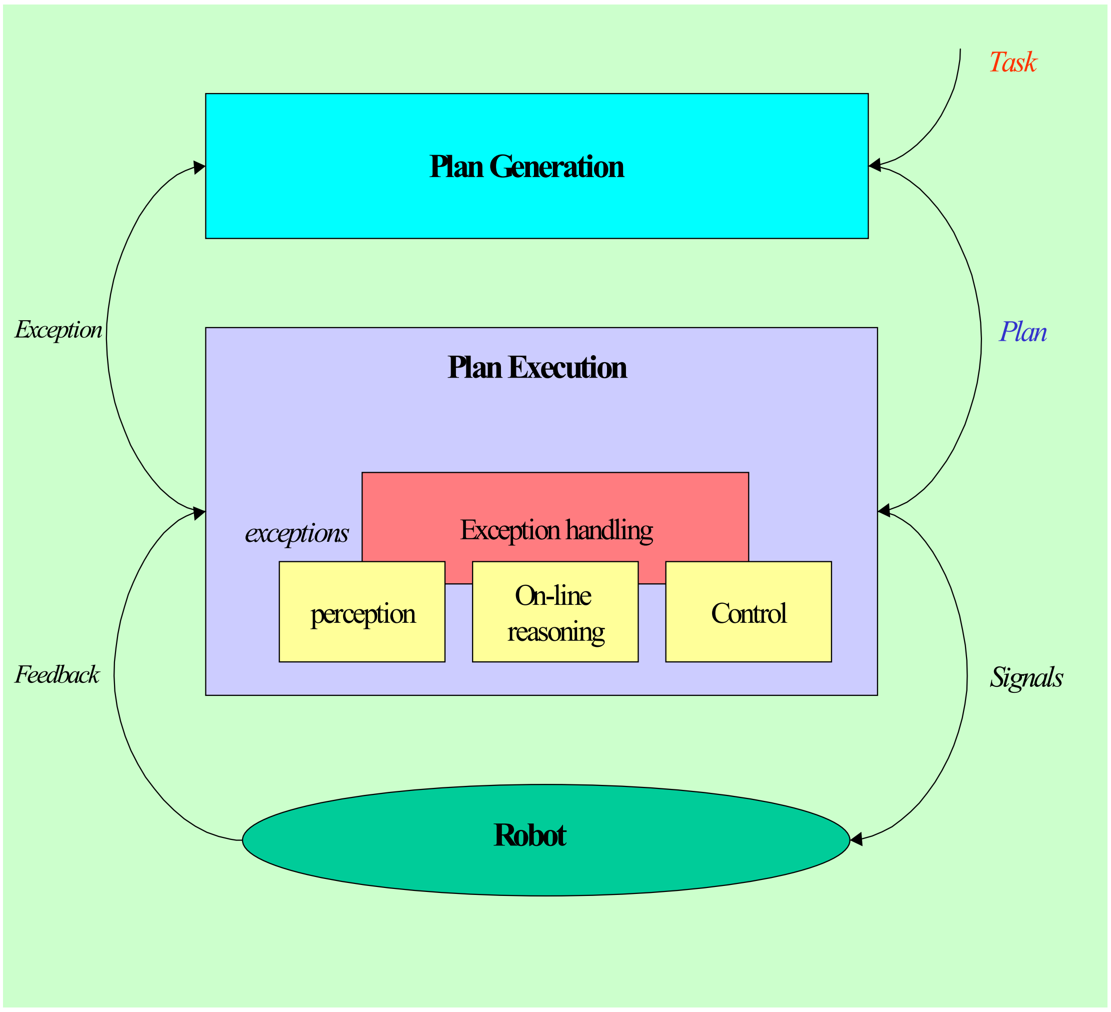


Figure 3-1: MARIE's functional architecture

3.2.2.1 Plan Generation System

The Plan Generation system must produce an executable task tree. There are two ways to do this. One is done interactively by a human operator, the other is automatic.
The operator supplies a collection of tasks and execution control information. A logical module interprets this data, verifies it, and builds a numerical task tree from it. It supports the operator by prompting for missing information.
The generation of a task tree is done off-line, because it is a time consuming job. The operator would not be able to keep up with the speed at which tasks are executed.
The preparation of the mission can also be performed by an automatic, on-line task-planner [2]. It is able to plan tasks from a collection of predefined, numerically formatted sub-trees. Based on initial and final conditions of each sub tree, it tries to connect them, into a complete task tree. Because it uses a fast algorithm this job can be performed on-line.

3.2.2.2 Plan execution System

The task tree that was built by the plan generation system is sent to the plan execution system. The plan execution system is responsible for executing the tasks of the missions and the handling of exceptions.
If an exception occurs it can't handle, it is passed to the Plan Generator. The execution of a task is accomplished by routines that are capable of perception, reasoning and control.

This system *controls* the robot. This means that the execution of the task tree is done on-line.

3.2.3 Operational Architecture

In the operational architecture the role of the plan generation system and the plan execution system are divided in logical modules that together perform the required functionality. Each of these modules are so-called Virtual Machine.

A Virtual Machine layer is capable of handling the instructions it receives, without any help from the higher layers. This approach makes it possible to hide system-dependent information from higher layers. Furthermore, dividing the system in virtual machines defines a hierarchy of processes and provides structured exception handling.

> *A virtual machine layer is defined as an instruction set and a virtual machine. The instruction set fully defines the capabilities of that particular layer, while the virtual machine interprets and executes them. A level N+1 virtual machine will express a task that it wants to accomplish, as a sequence of level N instructions. The level N machine will then interpret each of those instructions, and produce a number of instructions for level N – 1 [1].*

The virtual machine levels and their hierarchy are depicted in figure 3-2:

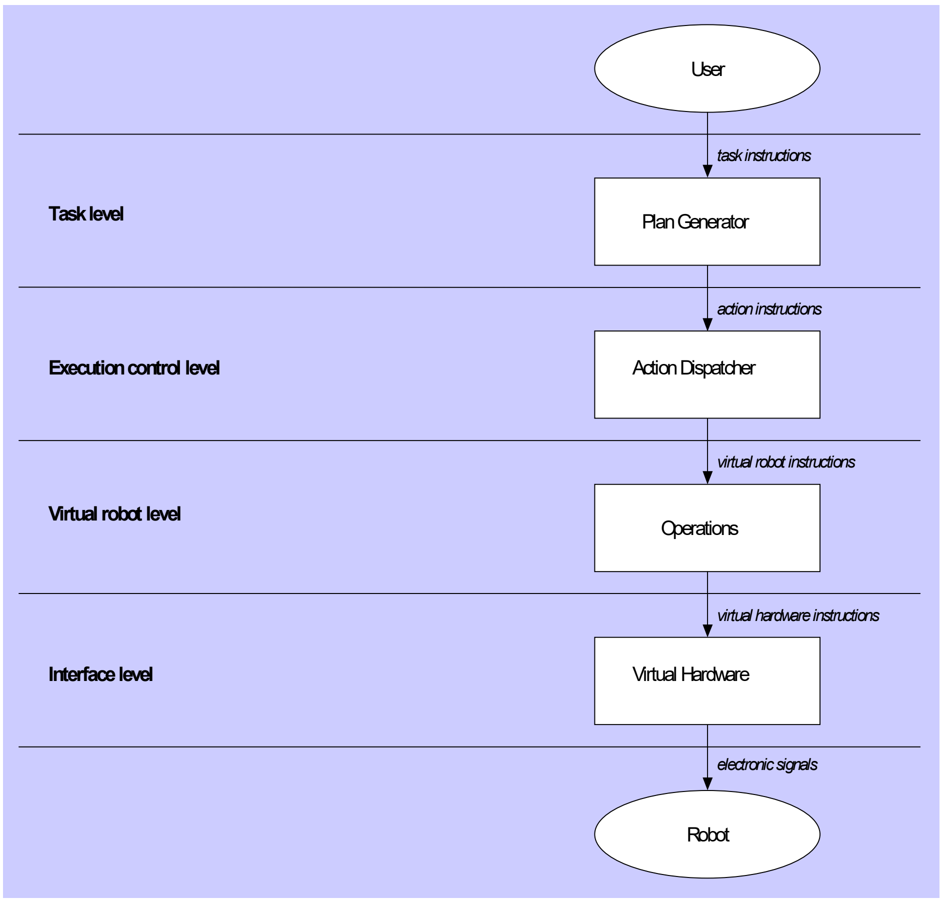


Figure 3-2: MARIE's operational architecture

The name of each virtual machine level is given on the left side of the figure. The virtual machine belonging to that level is given in the rectangle and the words in italic identify the instruction set that is produced and interpreted.

3.2.3.1 Overview

The operator and the task level together represent the plan generation stage. The operator is the top-level, the task level is the lowest level of the plan generation level. The operator creates a task tree with the plan generator. Additionally the plan generator converts the task tree in appropriate instructions for the next level.
The first level of the task execution level is shown as the execution control level. The instructions it receives also consist of a task tree. It gets parsed here and executed. Alternatives are present in the task tree to facilitate exception handling. When such an event occurs execution is stopped and an alternative is selected.
The next level of the task-execution level is the virtual robot level. At this level perception, reasoning and control are responsible for the robots behavior. This is where the operations are present. These operations are standalone units; they have no over-all knowledge, contrary to the higher levels. They instruct the virtual machine of the interface level. The virtual robot directly interacts with the hardware. Its instructions control the actual robot.

3.2.3.2 Task Level

The virtual machine of the task level, the plan generator, is the only software representation of the plan generation system. There are two different task planners available for MARIE; one is an off-line task planner, the other an on-line task planner.
The instructions the off-line planner accepts come from the operator and consist of elements of a task tree. The task planner combines these elements into a complete task tree, by prompting the operator for missing details. The instructions the operator gives are of a symbolic nature. To get the next layer to understand the tree, the task planner first translates it, before sending it. I.e. the plan generator performs task decomposition; it transforms high-level instructions into low-level activities that can be understood and executed by the plan execution system.
The on-line task planner, was recently created by Frank Terpstra [2]. The on-line planner is capable of combining sub-trees it has available in a repository, into a complete task tree. If execution of such a tree should fail, the planner is again consulted to supply an alternative tree, thus on-line planning can be done.

3.2.3.3 Execution Control Level

The Execution Control level, is performed by a virtual machine called the Action Dispatcher. It accepts a series of activities from the task planner. They are structured in the form of a task tree; linking actions with conditions for execution.
The action dispatcher parses the tree. When it encounters an action and its conditions are satisfied, it executes it by dispatching the necessary operations of the next level, by sending virtual robot instructions to them. When an action is finished it dispatches the next one, until the whole tree has been processed or the mission is completed. Feedback from the operations, influences the next dispatched action.
The action dispatcher takes care of activation and deactivation of operations. It does not execute actions itself. Since it has knowledge of the active operations, it is capable of stopping them, when, for example, the action is finished or an exception occurs. Autonomy is implemented at this level, because only here the combination of knowledge and control capabilities is present. The control capabilities are limited, though, because the system lacks the explicit knowledge on how to control the operations. This problem is the subject of this thesis and will be discussed in the following chapters.

3.2.3.4 Virtual Robot level

The operations reside in the virtual robot level. Each operation is a virtual machine, more precisely called an *elementary operation*, and is controlled by elementary operation specific instructions from the action dispatcher.
At this level perception, reasoning and control are implemented; specific tasks like detecting a docking spot, finding a wall or reading sensor input happen here.
Each Action Dispatcher instruction maps directly on an operation. An instruction consists of a command and parameters. The command can be 'execute', 'suspend' or something similar. Additionally it contains parameters specifying the settings of the task. The parameters are operation-specific, meaning that each operation has its own set. The parameters 'tune' the operations behavior and makes them adaptable to different circumstances.
An important feature of the operations is that they can run concurrently; for example, while sensor data is being read, the features of a wall can be extracted and a path can be planned. Operations are completely unaware of one another, they have no idea how other operations are manipulated and can't manipulate each other. Over-all knowledge is, to a limited extent, present at the action dispatcher level, and originates from the task tree, that was created by the human operator.

3.2.3.5 Virtual Hardware

The interface level implements the virtual hardware; this layer is a software representation of the hardware and directly interacts with the sensors and actuators. It is highly dependent on the hardware, for example, for each sensor and actuator one interface is defined. I.e. one for the ultrasone sensor, one for the shaft encoder, etc. The virtual hardware provides the hiding of details, like low-level input-output control, but also provides some basic safety measures like an emergency stop.

3.2.4 Implementation Architecture

In the implementation architecture specific hardware and software modules were implemented.
In this section the additional details of the virtual machines will be specified. The development of the software components was governed by basic software design considerations, such as modularity, expandability, adaptability and flexibility.
This has lead to two interfaces that supply generic means of interaction. A control interface defines the way in which operations have to be controlled and an intermediate data manager interface defines the data exchange between operations. This prevents the modules from directly interacting with one another, allowing modularity and flexibility. Expandability is achieved by the interfaces, because they enforce a standard on new modules. The data exchange interface makes modules more adaptable to different situations.

Figure 3-3 denotes the implementation architecture.

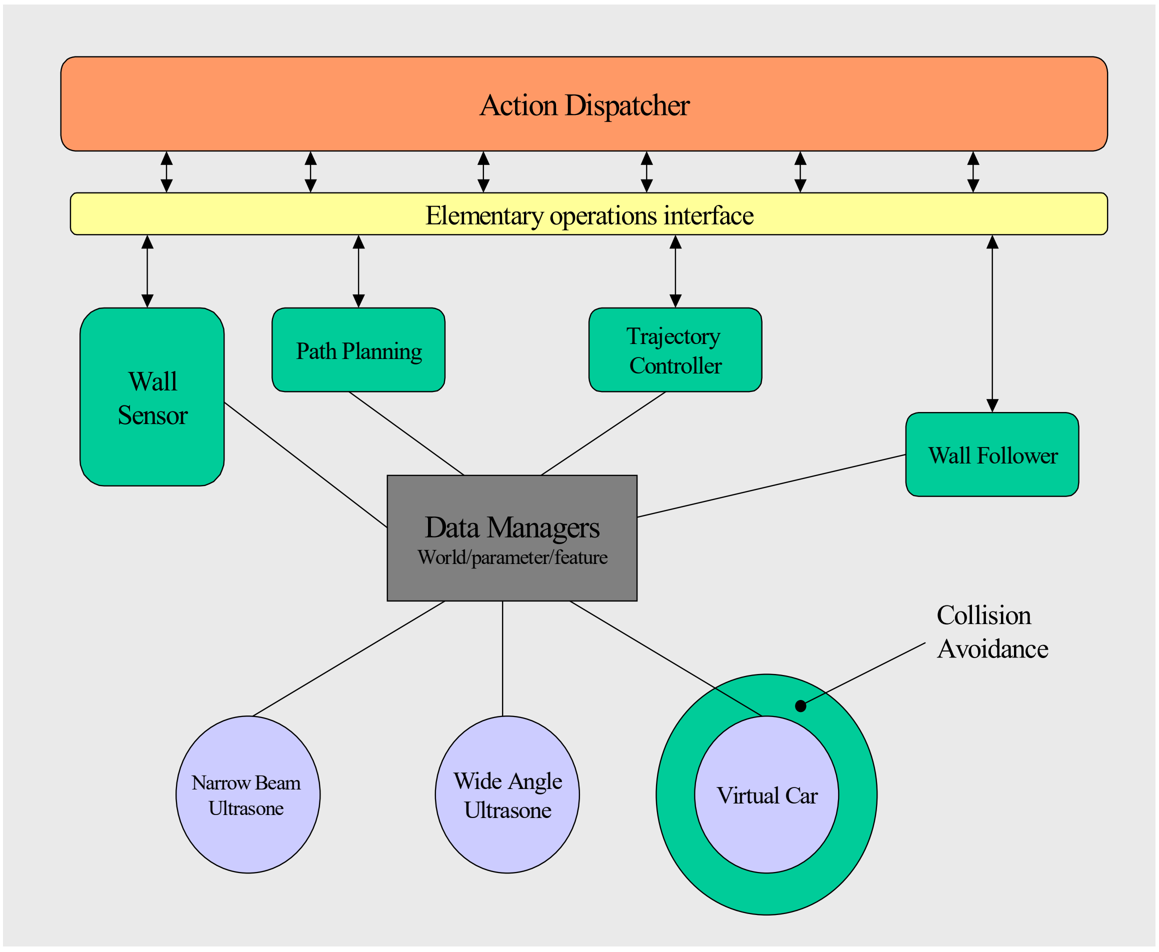


Figure 3-3: MARIE's implementation architecture

3.2.4.1 Action Dispatcher

The action dispatcher receives an instruction from the task planner in the form of a task tree. The task tree describes when to execute what task. The action dispatcher must interpret and realize this.
Tasks get executed by dispatching the appropriate operations in the virtual robot level. To do this the action dispatcher sends a command telling the operations what to do, this can be something like activate, suspend, etc. Additionally it sends parameters specifying the control for an operation more precisely.
When the action dispatcher has dispatched all operations in a task, it waits for one or more of the operations to finish, before dealing with the next task.

The action dispatcher does the following tasks:

- Receive and parse a task tree
- Activate and suspend operations
- (Re)parameterize operations
- Wait for completion of an action
- Interpret status flags

Tasks in the task tree consist of actions and conditions under which they can be executed. 32 bit flags represent these conditions. When an activity is finished executing, all operations return their status. The action dispatcher interprets this returned information and combines it into status-flags. A next action is suitable for execution if its conditions (i.e. its flags) are implied by that status-flag.

For example, one of the 32-bits is reserved for the operation that is responsible for finding a docking spot. When this operation is finished executing and has found a docking spot, it returns a flag with the specified bit set to one. The action dispatcher then combines that feedback with that of the (possible) other operations of the action, and thus produces a status flag. If it then finds an action with a condition flag that has the 'docking spot found' bit set (and all other relevant bits) it executes that action.

3.2.4.2 Elementary Operations

The capabilities of the operations define the capabilities of the complete system. Elementary operations perform the actuation, perception and control functionality of the robot. This means that all operations do different things and are controlled differently.
Examples of elementary operations are: Trajectory controller, Path Planner, Docking Spot detector, wall sensor.

The basic 'frame' for each operation is relatively simple: to provide optimal capabilities, it must provide functionality for control and to exchange data. This functionality is defined by a control interfaces and a data interface. Elementary operations must conform to these interfaces.
To cooperate with the rest of the system an elementary operation must contain the following functionality:

- Repeated activation/deactivation by the action dispatcher
- (Re-) parameterization, the settings that control the operation must be of a dynamic nature
- Signal status-flag
- Task completion leads to suspension, instead of termination
- Socket based communication, connect and disconnect functionality

This functionality is defined by the elementary operations interface, covered in the next section.
The action dispatcher controls the operations through two types of information. One is a command, telling the operation to start, stop or retrieve its configuration information. The second way of control is by providing settings. These settings are parameters for the functionality the operation exhibits. An example of parameters is the depth and breadth of the docking spot, passed to the docking spot detector-operation. The parameters are important for the exact function of an operation. Parameters need to be fairly accurate; a small deviation often results in completely different behavior.

3.2.5 The Interfaces

Essential for communication between execution control and operations are interfaces. There are two interfaces implemented in MARIE; one to communicate Virtual Machine instructions from the Action Dispatcher to an operation and one to facilitate communication between individual operations. The first is done over the *Elementary Operations Interface*, further called the eo-interface, the second uses the *Data Manager Interface*, referred to as the dm-interface. The eo-interface communicates virtual machine instructions, the dm-interface data-records.

3.2.5.1 The eo-interface

The eo-interface defines the functionality for interaction between the execution control layer and the operations. The operations and the controller of the operations must implement this functionality, but they both must conform to the standard it defines.
The elementary operations provide reasoning, control and actuation as services. The action dispatcher is the client, and uses them. To give control over an operation, the interface defines a generic method for communicating instructions, which tell the operation what to do and how to do it and give the operations feedback on their performance.
The control information that is passed between client and server consists of:

- A command,
- Parameters, and
- Status flags.

This information is exchanged as follows. The Action Dispatcher connects to an operation it wants to use[1]. It sends an instruction to the Elementary Operation and waits for a reply.
This instruction consists of the command and the parameters. The command specifies to the operation what to do, for example activate or suspend, the parameters tell the operation how to perform its task.
The answer the operation responds with, to the action dispatcher, after it is finished, is a 32-bit flag. The flag denotes the status of the system, according to the operation. I.e. in case of the docking spot, a bit corresponding to a found docking spot is set, or, in case of the trajectory controller, a ‘path has been driven’-bit is set. The action dispatcher processes this information and decides, based on the task tree, what action to take next.

3.2.5.2 The dm-interface

To prevent operations to become to reliant on one another, communication is done through an intermediate data manager. Data managers are designed to store data. When an operation produces data it can store it there so that other operations can retrieve it.
Using data managers must be done through the data manager interface. The data manager interface defines functionality to store, retrieve and delete data entries from a data manager.
The data managers are powerful, because of their simplicity. No complicated data structuring is performed. The idea is that both supplier and retriever of data are supposed to know what the structure of the data is they want. For example, the ultrasone sensor produces data, in the form of points. It is stored in the data manager. It is then retrieved by the line segment extractor, whose task it is to ‘see’ a line in this data. Both operations don’t know of each other’s existence and the retriever assumes the data to be present.
This means that any data may be stored. To make it recognizable some structure on the data, namely a division in a class and up to two sub-class levels is defined.
The operation that is the source of the data defines the classes, the operation that will need the data needs to know that structure. On storing the data, the manager adds additional context information, like a sequence number, a time stamp and the source.
The interface defines one very powerful function to retrieve data, in particular. Based on almost any criteria (time, sequence number, class, sub-class) data may be retrieved. This

[1] All connections between action dispatcher and elementary operations are set up on start-up of the system and only get closed when the action dispacher or the specific operation terminates.

broadness is very important, because the stored data is (almost) only relevant to the operator of that data, and because of that broadness it can be very meticulous about retrieving it.
There are three data managers defined in the system. A world datamanager, containing world information, a feature datamanager, containing world features and a parameter datamanager, these operations plan parameters for an operation in a consecutive task.

3.3 Utilizing MARIE

3.3.1 Functionality of the system

On initialization of the system all processes that are going to be used, will be started . The action dispatcher will set up a communication channel with each operation using the eo-interface and all datamanagers through the dm-interface.
The system will then built a task tree. This is done by one of the task planners.
The on-line planner is very resourceful in letting MARIE maneuver. This means that when testing one is never sure if success was due to the tested component or the on-line planner. This same line of reasoning holds for explaining odd and unexpected behavior. This planner is too good at solving problems. It is useful for demonstration purposes, but for serious experiments the old planner is better usable, because it allows for a better examination of results.
The alternative is to create a task tree by the operator and the (original) task planner. This proceeds off-line, all processes that are not involved, wait until the task tree is finished. Prompted by that task planner, the operator must supply the elements for the structure of the tree. This way the operator provides the control details, through the task planner, for the execution control system. The operator can use scenario's, which contain pre-processed sub-trees, in an acceptable format for the task planner. The task planner does not know if it receives information directly from the operator or out of a scenario; it has no effect on the robots functioning.

Once the job of building a task tree is done, it is sent to the execution control system. Here the action dispatcher parses the tree, when it finds an action node, it dispatches the appropriate operations, through the eo-interface, by sending an activation command. Then it will wait for an operation to finish, upon which it will decide what to do next. Some operations must finish their execution, instead of being told to stop. The Trajectory Controller is such an operation. It drives a path and must be able to finish, regardless of the result of other operations.
When all active operations are suspended or stopped, the action dispatcher interprets their returned statuses and based on those, the parsing of the tree continues.
Execution control is enforced by the action dispatcher. It is capable of this thanks to a mechanism that involves condition flags. By means of these flags the action dispatcher decides what actions it takes. Based on the systems status the action dispatcher executes nodes of the task tree. When that event occurs and there are no more nodes to process, the mission has succeeded.

3.3.2 Example

To illustrate how MARIE processes information and how its components interact, an example mission will be explained. The goal of this mission is to position MARIE in a corner of the room. See figure 3-4.

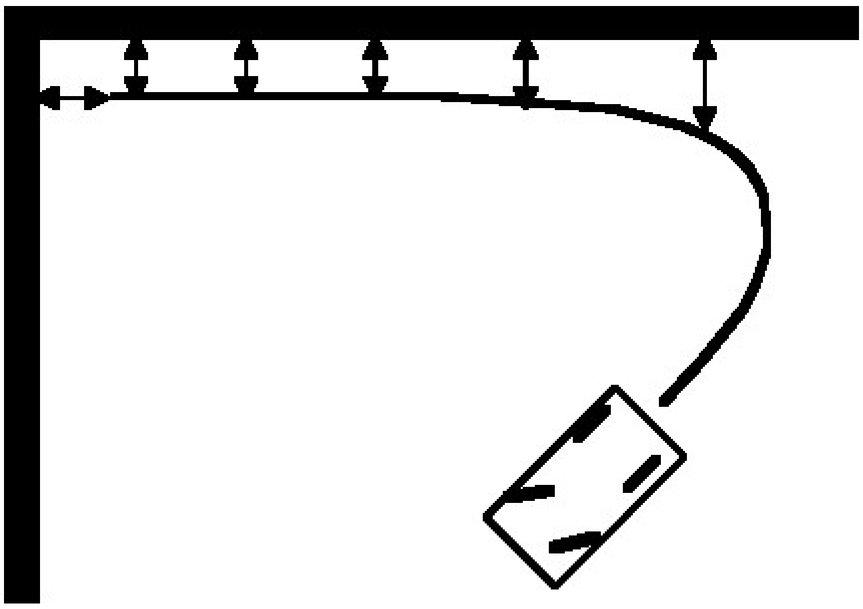


Figure 3-4: Driving backwards into a corner

The car knows the layout of the room and its own position relative to the corner. First a path will be planned to the corner, then the wall must be found after which the cart will follow it and drive to the corner. These four steps have to be performed:

1. Planning a path into the corner
2. Driving the path until a wall is found
3. Aligning the vehicle with wall
4. Follow the wall until the corner is reached

The task tree consist of a node (the root) that has four actions corresponding to four steps. The operations of each action are:

1. Path Planner
2. Wall Sensor, Trajectory Controller and Collision Avoidance
3. Wall Sensor, Wall Follower and Collision Avoidance
4. Wall Sensor, Wall Follower and Controlled Stop[2]

The first action is done by one elementary operation, the other action by several at once.
The operations interaction and data flow:

[2] The controlled stop is actually a special instance of the collision avoidance; when an obstacle is detected it signals full stop.

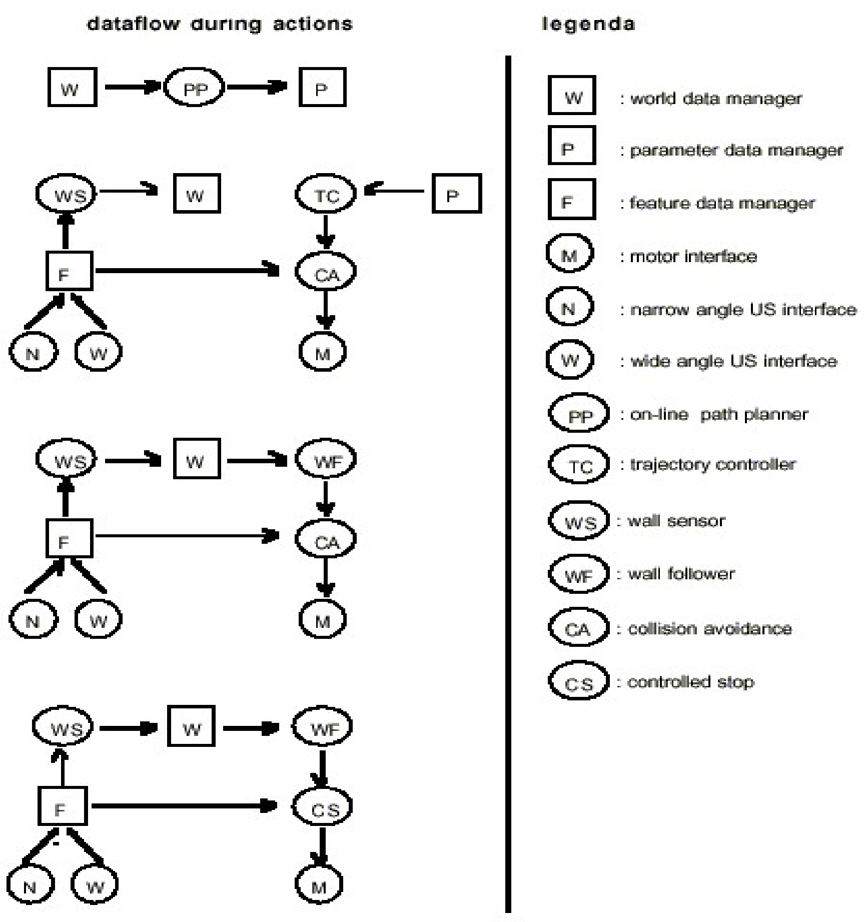


Figure 3-5: interaction and data flow of elementary operations

The first action plans a path from the world map that it obtains from the world datamanager and stores the path in the parameter datamanager.
The second action consists of the trajectory controller drives the path that it obtains from the parameter datamanager, while the collision avoidance makes sure that it does not bump into obstacles. While doing this the ultrasone sensor acquires data from the surroundings, it does this with narrow and wide beam ultrasonic sensors and stores their findings in the feature datamanager. Simultaneously, the wall sensor looks for a wall, from the ultrasone data in the feature datamanager.
Next, when a wall is detected, the wall follower will align the cart along this wall, using the ultrasone data in the world datamanager, while the wall sensor continues to update that data. When the cart is aligned the wall follower will drive the cart backwards, along the wall, until the collision avoidance detects the other wall, resulting in full stop; the cart is parked in the corner.

3.4 Conclusion

This chapter explains how the MARIE robot is capable of solving complex tasks. A mission is given to the robot, in the form of a symbolic task tree, that is executed, and results in the robots versatile behavior.
MARIE's software is a so-called hybrid software architecture, a combination of a functional architecture and a behavioral architecture. The software's high-level modules first create or gather a symbolic task tree and then translates it into a numerical task tree. This task tree

consist of actions, that are executed when certain conditions are met. A module called the action dispatcher is responsible for processing the task tree. Actions are executed by activating simple processes, called elementary operations, they do things like following a wall or reading sensor input. The combination of these operations are a virtual representation of the robot; the robots behavior is a direct result from there processing. Actuation, sensing and reasoning is done by these elements. A strong characteristic of the elementary operation is that they can run concurrently; for example, while sensor information is being gathered, a wall representation can be constructed from it.
Information is exchanged by means of two interface; the datamanager interface (dm-interface) and the elementary operations interface (eo-interface). The dm-interface allows for the sharing and storing of data, produced by the elementary operations. By means of this interface the elementary operations make use of each other. The action dispatcher enforces control by means of the eo-interface. The interface generalizes control for all the elementary operations. Control instructions for elementary operations and state information are communicated between the action dispatcher and the operations, through the eo-interface.

4 PROBLEM DESCRIPTION

4.1 Introduction

With respect to adapting to changes or extensions of the robots controlling software, MARIE can be improved. This chapter describes this improvement.
Section 4.2 gives an overview of high-level observation that led the conclusion that an improvement is due. In section 4.3 out-lines the general area in which MARIE may be improved. It describes that the robot is unable to adapt to modernized or new software components. This makes much desired, high-level functionality, such as self-configuration and self-healing impossible. After the domain of the problem is narrowed down, in section 4.4, it is possible to describe exactly what information the system lacks. This description is divided over two sections; section 4.5 deals with the location information of the components, and section 4.6 deals with the controlling elements and their specific control details.
Section 4.7 will look ahead towards a solution and section 4.8 will conclude the chapter.

4.2 Overview

The research and development of MARIE is meant to, eventually, increase its autonomy. The autonomy of a robot can be derived back to its ability to do things 'on its own'. For MARIE, this has to quite an extend been accomplished already. However, functionality can still be improved.
It is inevitable that a complex system, like the control software and hardware of MARIE, suffers from its size; control decisions that work fine in a historical context, have to be re-evaluated in situations where the system is extended with new functionality or modernized in general. This is reflected in the work-intensive job it has become, to prepare experiments and adapt to changes in the software. The general cause for this is that the robot lacks means to deal with information about the system as whole, and can therefore not reason about itself on a higher-level. Currently, it relies on the operator to make system-wide decisions. The variation, high quality and amount of control information that the operator has to take care of in order for MARIE to function appropriately, is the information the system should have in order to be able to reason on a grander scale.

In general, this thesis is concerned with solving the lack of self-knowledge around configuring and knowing how to control the elementary operations. A solution will be sought that bridges the knowledge gap between the operator and the robot and provides the system with additional functionality that should give it more high-level knowledge.

4.3 Room for Improvement

MARIE has extensive means to accomplish goals, because a lot of high-level, machine independent software that it has available. This demands flexibility on the part of MARIE; it must be able to plan a multitude of tasks and execute and vary control, interactively. The action dispatcher is the one module that is responsible for this, facilitated by the eo-interface. That flexibility of one module controlling all elementary operations comes with a price, because controlling the elementary operations requires elementary operation specific information, such as their location on the network and its control details, i.e. the information that tunes the functionality of the elementary operation. If these change, for example when a different configuration is set up, or the control structure of an elementary operation is improved, these changes have to be manually brought to the attention of the system by the operator or even the developer. If the execution control system were able to obtain the knowledge about locating or controlling elementary operations, it would have the means to

become much more autonomous. Characteristics of autonomy, such as, parameter optimization, predicting results or self-analysis would be easier.
Currently, the only means the robot has to control the operations are defined in the eo-interface, which is generic enough to *communicate* all commands but gives no insight into the structure of the control information, let alone its contents. The eo-interface is not able to deal with the high-level information about the system as a whole, such as locations or control structures for elementary operations.

Practically, the focus of this thesis will be on configuring MARIE. This is, at the moment, made more difficult by the static configuration of MARIE's software and the inability to obtain information about the elementary operations capabilities. In theory, dynamically configurable software and the ability to obtain such control information gives the robot means to configure itself. Such self-configuration increases the autonomy of the robot and decreases the operators concern with the system.
The design of the software architecture of MARIE does take distribution into account, but has never been implemented, due to practical reasons. Solving this location dependence issue of MARIE's modules, as an initial step in providing self-knowledge, should be realized first. When this has successfully been accomplished the supply of elementary operation specific control information should be developed. For this a design will be presented.

4.4 Domain of the Problem

Before going into more detail, it must be pointed out that the subject here is centered around versions, settings, and all other elementary operation specific information that describes how a certain elementary operation must be used and *not* the work spent on devising tasks-trees as far as the sequence of tasks is concerned.
Or, in other words, what is explained here describes *how* an operation needs to be controlled, it does not describe the functional side of the elementary operations or their cooperation. This is motivated by the opinion that the operator's job to supply a collection of tasks, that together solve a problem, is the job of the human. The robot must be able to execute the tasks and control the elementary operation therein, in as much an autonomous fashion as possible, without help from the operator.

4.5 Location Dependence

Historically all software of MARIE ran on one computer. Since a wireless network was added to the hardware of MARIE this is no longer required. Processes of MARIE can now be run on any computer within the network. In order to make use of this, MARIE should have the ability to obtain and reason about the location information of its processes. The system has no such mechanism, and to make matters worse, locations are hardcoded.
The introduction of distributed computation in the system increases the computational capabilities of the system and allows for some evolution in the way the software configurations are realized.
Historically all processes ran on the on-board computer, limiting the amount of possible configurations. These limitations no longer hold in the distributed version of MARIE. So that makes it possible to establish configurations more 'to the need' of the robot. For example, selecting between different versions of one process becomes an option. To be able to do this kind of reasoning, knowledge about elementary operations, i.e. where they can be reached and how they are controlled, is required. This knowledge is different for each elementary operation and can currently not be obtained by the system in a generic fashion.
In all circumstances it is the operator that is responsible for configuring the system and supplying the control information. In the world of AI this type of control and configuration being pre-defined is somewhat out-dated. Compare this for example to autonomous agents;

they rely heavily on the ability to communicate their capabilities and are independent of the agents location.

4.6 The Applicability of Elementary Operations

The issue of the robots inability to obtain information on how the elementary operations are controlled, and the fact that it can't deal with the dynamic location of its processes, is the result of the non-uniform information that controls the elementary operations.
To understand this one must realize that if the structure and content of the control information for the elementary operations were of one format for each elementary operation, there would be no problem, because if the system were aware of that structure it would automatically have the potential to enforce control more autonomously. The fact that for each elementary operation, location and control information is not generic or pre-determined, means that specific knowledge is required for each elementary operation in order to make use of it.

The following example will explain the problem when developing and applying operations. It should demonstrate how the system would improve if it had information about the location of elementary operations and their control information.

- A Development Example: wall follower vs corridor follower

  MARIE contains an elementary operation called the wall follower. As the name suggests, this operation is able to follow a previously found wall. Because MARIE often drives through a corridor, a corridor follower was desired. It was decided not to make this a new operation, but instead, to extent the wall follower with extra functionality; a parameter was added to the operation to indicates if a wall or corridor is to be followed.
  To be able to use old task trees, that don't take this new parameter into account, and new task trees, that do take the new parameter into account, the operator could have the old and the new wall follower active on the network. MARIE's software should then decide 'on its own' which of these two instances its should use, based on the compatibility with its task tree .
  There is a desire for the system to be able to determine the proper configuration with its task tree, based on the available elementary operations and the way they are controlled

Note that such configuration issues would be no problem if MARIE could locate and choose between available elementary operations based on their properties such as, control information and version.
At that moment MARIE cannot choose between two instances of elementary operations, because it lacks functionality to distinguish between multiple instances of an elementary operation and even if it could distinguish two instances of an elementary operation, the robot could not use them, because the software relies on hardcoded locations.
The desired situation is that the robot could assemble a working configuration from processes that reside somewhere in the network, based on the elementary operations characteristics, instead of the operator specifying what elementary operations to use and then re-linking and restarting the entire system .

The role of the operator
Dealing with the flexibility of controlling and connecting the elementary operations is currently done by the operator that has to fill in all the 'missing' knowledge. The operator must configure the system and explicitly tell the action dispatcher where, which elementary operations can be found. So, because the system is not aware of the locations or capabilities of

elementary operations a static configuration is enforced. When during the development of new elementary operations these properties change, a manual reconfiguration of the entire system must follow to make the changes known to the 'users' of that elementary operation. Control of each individual elementary operation requires operation specific information. This is the concern of the operator, because the system cannot obtain these details and cannot handle them because it uses a (too) generic interface to communicate them.

This thesis will introduce location independence and a method that gives the robot more information about how to control the elementary operations and deal with variable locations. This will be done by enabling it to obtain elementary operation specific information on the location and capabilities of its operations.

In order to explain what information controls the elementary operations and to debate what information should be made available by the system itself, the next section will elaborate on the properties of elementary operations.

4.6.1 Operation Management

The operator supplies the information that tunes the functionality of the elementary operations. The 'tuning-information' consists of settings that specify the behavior of an elementary operation. By means of these settings a variety of control can be enforced, that allows for MARIE's broad applicability.
The robot takes care of starting the processes that will be used, activates them when needed, supplies them with control parameters and reacts on their feedback.

The location and 'tuning' information of the elementary operations are part of the elementary operations properties. Identifying and describing an elementary operation (in the context of configuration and control) can therefore be done by these properties.
The information for controlling the elementary operations is listed below.

Information due to the functionality of elementary operation:

- Amount of operations
    - Functionally different operations
    - Different versions of operations
- Settings of Operations
    - Different Settings per operation
    - Prediction and Accuracy of Settings

Information for the configuration of the system:

- Static Configuration
    - Existence of operations
    - Static Location of operations

The contribution of each aspect to the over-all problem differs, but it is the combination of these points that cause a great amount of possible control information.
Next, these aspects will be clarified.

4.6.1.1 The Amount of Operations

Approximately, 8 operations have so far been implemented. They can be divided into three types: operations that sense, reason and control.

Sensing is done by:

The Ultrasone –operation
Wall Finding –operation
Docking Spot Detector –operation
Segment Extractor –operation

Reasoning is done by:

Path Planning –operation

Controlling is done by:

Wall Follower –operation
Collision Avoidance –operation
Trajectory Controller –operation

♦ *Functional Range of Operations*

The operations are the 'simple' units that perform reasoning, activation and sensing. Thus a path planner is an elementary operation, but also the ultrasone sensor reader is an elementary operation. High and low level functionality is mixed at the operation level. The operator has to supply control information for both and thus must know information ranging from low-level details, such as the direction of the ultrasone search beam, and high-level functional knowledge, such as the width and breadth of the docking spot.

♦ *Different Versions of Elementary Operations*

Throughout the years MARIE has evolved. This advance is the result of development of new modules and enhancement of old modules. Both from a functional and operational point of view changes took place. Functionality of modules was improved, technically modules were made better applicable, for example, the entire system was ported to Linux. Although most individual changes are documented, version management was neglected. There used to be a version management of the system, but this has not been used for some years[3], resulting in a variety of different versions of modules. It is unclear what module is the most recent one. This gives rise to the following problems:

- Old modules are used by mistake in (new) experiments; in experiments old modules have been used, when newer version were available. This could result in failure, and consequently a time consuming and *wrong* problem analysis
- Old experiments become less informational; suppose a later performed execution of an identical experiment gave different results; is this caused by differences in version or lies the discrepancy somewhere else? Again, this costs time to figure out.
- New modules can become too advanced, in comparison with other modules; such as the on-line task planner that is unpractical to be used in experiments.

4.6.1.2 The Settings of Operations

Elementary operations are not designed for a particular moment; rather, they are designed to be applicable on as many occasions as seems fit. This requires that an operation has functionality that can be configurable, so that it can be adapted to a specific circumstance. This adaptation is done by means of parameters. These specify the exact behavior that is expected from an operation.

- *Different Parameters for each Elementary Operation*

There is no framework for parameters or any other directive that specifies what they must look like; every elementary operation can have its own set of parameters. This means that

[3] This is not primarily due to laziness, but lack of system overview and tests for compliance with the rest of the system.

each elementary operation is controlled differently. Both from a semantic and a syntactic point of view the parameters of each elementary operation differ. The system itself cannot reason about these parameter structure, it doesn't even have them explicitly available.

- *Prediction and Accuracy of Settings*

All parameters have to be predicted before hand, for each elementary operation in every situation. They can't be changed during execution.
The system doesn't contain methods to estimate are supply default values. Some operations are sensitive to their parameters values and some are not. This further increases the complexity of supplying parameters.
From experiments done for this thesis with the docking spot detector it was shown that the values for these parameters had to be very precise. The parameters of the docking spot are very simple: one must supply a breadth and depth of the park-spot. The elementary operations is sensitive to its settings. A slight deviation in their value from the real docking spot would have the operation fail[4] [21].
On the other hand, the Trajectory Controller is pretty flexible in handling its path-parameters. When an object is present on the path to drive, the collision avoidance makes sure no collision happens.
To get insight into the parameter settings that were used in experiments Eric de Ruiter has created a database that links this information to the software version that was used and some additional information [3].

4.6.1.3 Static Configuration

During execution additional hardships arise concerning the control of the operations. This is due to the limited capabilities of the system to manage its own configuration. Currently, the over-all system configuration is only implicitly present.
This deficit exhibits itself by the absence of the capability to manage the following information:

♦ *Existence of Operations*

MARIE was designed to be a multi tasking system, running on a single machine. To function correctly the system must know all components that will participate in the execution of a task. However, the system only knows of the existence of the elementary operations it started. When the operator or a developer starts an additional operation it cannot participate in the rest of the system or replace an existing elementary operation, because there is no mechanism to announce that it is there. Additionally, during execution the system assumes all started operations are still there, but never knows for sure, because there is no mechanism to check this. When an operation crashes, for instance, the remaining software isn't informed and therefore cannot recover.

♦ *Location Dependence of Modules*

An additional problem has resulted during development of modules of MARIE. This is the result of the practical situation the robot was used in. It caused two different, but very much related, problems:
One needs to remember that there is one main computer aboard the MARIE robot. This computer runs the VxWorks operating system. Although the software of MARIE is set up to run distributed, all was run on that VxWorks machine.
It resulted in hardcoded locations of modules, i.e. when a module needed to find another module it just connected to the 'local machine', being sure the module would be located there.

[4] One could very well justify blaming the docking spot implementation for this. However, the fact remains that adjusting the parameters solves the problem.

This caused an additional change that became possible because of some memory management features of VxWorks. To avoid getting too technical, and drift off topic, the explanation will be kept brief. In short: Different processes and threads share global memory and a single name-space on VxWorks, this, practically, means that global variables are known throughout the system. In a programming context this is very 'dirty', in the context of MARIE it was gratefully exploited. Various forms of data were shared between all processes e.g. software modules. And that's what's wrong: data sharing. That's bothersome, because modules cannot be properly compiled, tested or debugged without having to deal with all the other modules that run on the system. One could say that these two changes led to a decrease in reusability and maintainability, which means that modularity, is compromised.

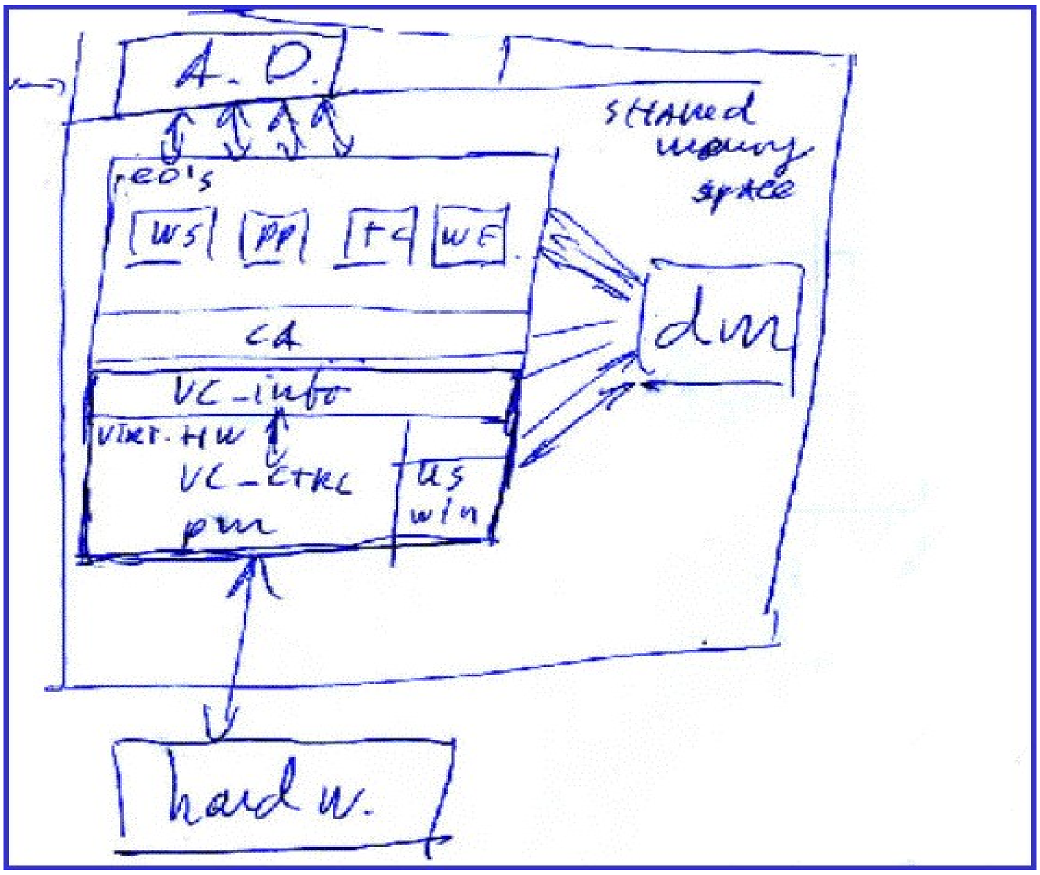


Figure 4-1: As it is

An example of the sharing of system wide knowledge will clarify this.
On start-up the action dispatcher connects to all elementary operations and data managers. A handle identifies the connections. This handle is globally known throughout the system. This makes it possible for all clients of a datamanagers to use the established connections, *provided* they run on the same system. The modules that use this shared information cannot choose to do otherwise, this means that they must run on the same machine as the action dispatcher. Goodbye distribution, farewell modularity.

4.6.2 Concluding

The need for improvement in controlling the operations are on the one hand the result of MARIE's behavioral architecture; the collection of operations and their different way of control are the result of the demand that all behavior should be exhibited by one virtual

machines. Because they must be able to run concurrently, they reside in the same operational layer, even though from a functional perspective the operations are very different. The functional difference of operations demands operation-specific control, which leads to a great variety of control information.
Operations are controlled by their parameters. There are no development limitations on the parameters; there can be as many as the developer seems fit. This leads to the disadvantage that there is no straightforward way to know how they are controlled and that makes it the job of the operator to equip them with the correct control details.
Additionally parameters are liable to change during run-time. The operator has to foresee this and supply the correct values for the expected situation. Combine this with the sensitivity and amount of parameters to realize the meticulousness and extent of this job.
The problems with respect to version management and location dependence, on the other hand, are the result of a discrepancy between the design and implementation of the system. By design all modules must be able to run on different machines and their communication must be performed over a network. In the implementation this has not been fully developed this way; modules are to such an extent dependent on one anther that they must run on one machine.

4.7 Towards a solution

The existence of a behavioral architectural layer in MARIE's software, that contains a variety of functionally different elements and that are controlled from a central process enforces a flexible but general control mechanism. The cooperation of the different modules of the robot is facilitated by two interfaces; the dm-interface, for data, and the eo-interface, for control values, but cooperation on a higher level is not possible, because the system lacks both means and knowledge to reason on a level that is comparable to that of the operators'.
The current interfaces don't have the capability to deal with information about the system in general. No knowledge is communicated *about* the systems components, but 'just' the information that controls the robot. A solution should be devised that introduces the ability to communicate about the system as whole. A meta-interface on self-knowledge. The goal of the thesis was previously formulated as 'bridging the knowledge gap between the system and the operator'. An interface that communicates *properties* is expected to do this, because it provides the information that allows for reasoning from a more birds-eye view of the system, just like the operator does.
Such a property interface would be useless if there is no application that uses it. For this reason, as the example from section 4.6 shows, self-configuration could be an application that improves the system, when using the interface.

4.8 Conclusion

MARIE is an up and running, fully functional system. The robot has means to communicate data between processes, it has means to communicate control information to processes, but has no means to communicate more general information about itself, making it more difficult for the robot to reason on a higher-, operator-like level. An example situation in which this becomes apparent is self-configuration. This is impossible due to the lack of knowledge to identify individual instances of elementary operations.
Two aspects were identified that can be improved. One improvement is providing the system with self-knowledge about the detailed control knowledge for its versatile control mechanism. The second aspect that can be improved is the way the robot deals with location information; since the introduction of the wireless LAN there is no longer any need for a static, pre-defined configuration of the robots software.
This thesis seeks to improve the system with regard to controlling and using elementary operations, independent from their location. The general goal will be to find a solution that

provides MARIE with more autonomy. It should be able to acquire information about the properties of its elementary operations, so that it knows where they can be found and knows how to use them.

5 SOLUTIONS

5.1 Introduction

This chapter describes how to the previously formulated improvement could be realized. First, in section 5.2 some existing methods that deal with decreasing the operator involvement in creating and executing task trees and testing new modules will be described. Section 5.3 will describe the applicability of aspects of the research that was discussed in chapter 2, by evaluating the four literature examples, with respect to how they deal with location information and specific control details. In section 5.4 the proposed solution will be presented; a functional description will be given that should improve MARIE.

5.2 Existing Solutions

The system contains a collection of methods that helps the operator control MARIE. In this section these methods and their pro's and con's will be discussed. The methods that will be dealt with are:

1. Pre-defined scenario's that can be used when producing a task tree,
2. online parameter planners during task tree execution,
3. the on-line task-planner

1. Pre-defined scenario's

Why do the same thing twice? The idea behind scenarios is that task trees can be stored as a 'script', facilitating reusability. One can store a task tree in a so-called scenario-file type format. The system, the task-planner in particular, has been adapted to accept these scenario-files as input, and incorporate them into a task tree. This means that tasks, that are described in a scenario, can be reused in different experiments. This decreases the amount of work for the operator, especially when experiments are repeated.
However, scenario's contain static control information. When, for example, the control information for an operation changes then all scenario's that use that operation, are out of date and can no longer be used, unless they are updated. Additionally, scenario's are in an unreadable format, which makes it hard to understand what a scenario precisely does, and that makes them hard to incorporate into other trees. On top of that, scenarios still have to be made by the operator.

2. On-line parameter planners

The system allows for the use of on-line parameter planners. These are themselves elementary operations that are meant to parameterize other elementary operation during execution. Using parameter planners has the benefit that it can adapt the behavior of an elementary operation to the 'current' situation, and thus provide great adaptability for the operation that needs the parameters.
Currently, there is only one parameter planner available for MARIE. This is the path planner, whose task it is to plan a path which serves as a parameter for the trajectory controller.
There are two problems with this approach. Parameter planners will have to be specifically designed for each elementary operation, this is not straight forward and requires a lot of work. Secondly, the planners also need control information, so this approach would just move the problem from the elementary operation to the on-line planners.

3. The on-line task planner

The on-line task planner takes the trouble of creating a task tree out of the hands of the operator all together. This planner, created by Frank Terpstra [2], is capable of composing its own task tree from a collection of sub-trees, given a start- and end-state. Although this means

that the operator is not needed anymore to supply all control information, this is still not a satisfactory solution.
Using the online task planner circumvents the tediousness of settings, but was, obviously, not developed for this purpose. The problem with the on-line planner is that it is not very useful when testing the software, because the on-line task planner greatly influences the result. Examining the results of a test is quite some work, because besides the tested component, the behavior caused by the on-line planner has to be taken into account as well.
Additionally, the macro's used by the planner assume the software to be static, so changes in the software, require manual adjustment of the macro's.

Concluding
The major drawback of all these solutions is that they cannot automatically adjust to changes in the software. For an existing configuration most work fine, but, when adding new operations or conducting new experiments just as much work is required. The methods assume the system is fixed. Since MARIE is a test-bed and as a result changes in its software are not uncommon, the system would benefit from a method that automatically updates the information about its components.
There are no methods to manage the systems configuration, nor does the system have methods that deal with the availability of operations, the correctness of its parameters, etcetera. A successful solution will need to obtain and use such knowledge.

5.3 Alternative Solutions

These sections investigate to what extent the presented examples from chapter 2 can be integrated in, or used for, MARIE's software with respect to the improvement mentioned in chapter 5. As stated there, an improvement is due concerning the fact that the specific control information for elementary operations is only known by the operator and not by the system. A solution that allows for that knowledge to become available is needed.
Systems that consist of various, functionally different, components that together cause the behavior of the entire system are common in literature, because this is the holy grail of agent technology [8].
The presented systems are all born from a desire to control (heterogeneous) components in a distributed computing environment that is becoming more complex. Both, MARIE and the presented systems, need to ascertain a components control information.
This section discusses the possible application of the discussed systems and methods, with respect to locating and using distributed, differently controlled, components.

5.3.1 Jini

In Jini location and control information is actively published by the service. A client identifies a service in Jini by java types, that identifies the object, as a reply it gets sent the specific control information. In Jini this consists of possible (remote) methods that can be called to the service. Communication proceeds directly with the service

Jini has been used as the bases for various systems [22], but using Jini for MARIE would be very elaborate, because Jini relies heavily on Java. The concepts of Jini, however, such as the way that the identification, localization and control issues are solved, could be learnt from.
MARIE's elementary operations are the services that clients want to locate and control. If the concepts of Jini would be used for MARIE, then the elementary operations should register their locations and control information at a specially devised lookup service. It would add additional attributes to this self-description with version information, and such.
The client (the action dispatcher, for example) can request this proxy from the lookup service, so that it knows where the elementary operation is located and knows how to control them.

This presumably solves the location and control issue for MARIE, but at the cost of quite some additional design and implementation. Besides that, one would have to design the proxies for each elementary operation, and make it transferable, executable code. Applying the proxy concept with C, means that CORBA [19] would have to be used, and, to communicate code between processes, a special interface would have to be devised to describe the proxies in and thus the elementary operations' control information.

Advantages of Jini approach:
- solution to localization issue
- communication of attributes allows for better description of elementary operations

Drawbacks of Jini approach:
- concept of proxies unpractical
- unacceptable amount of additional design and implementation

5.3.2 HIVE

A system setup with HIVE consists of *cells*, *shadows* and *agents*. The cell contains services on a system, shadows are interfaces for using these services and agents are meant to communicate between shadows (and thus cells). The agents communicate and share resources with each other. Programs in HIVE are represented by a collection of cooperating agents.

The idea of HIVE is to let heterogeneous components in a system cooperate. This is exactly what we want. Because of the structure of separating the system in services (cells), interfaces (shadows) and agents HIVE could be used in almost any distributed system.
On the surface, HIVE provides a nice solution to the static location of modules. This is solved, or rather, worked around, by leaving them in a fixed location, and have agents by means of shadows interface with them[5]. It is the agents' task to locate the process then.
This cannot be used as a solution to introduce distribution in MARIE's software, because desired benefits from real distribution cannot be met by HIVE's methods. I.e. the monolithic appearance of the software is not solved; debugging, and ease of configuring the system are thus not realized, unless the agents facilitate in this as well.

Functionality from a system that uses HIVE is derived from the cooperation of the agents. In analogy the functionality of MARIE is derived from the cooperation of its elementary operations. The big difference here, is that MARIE requires a central component (the action dispatcher) that governs this cooperation, that would also have to be represented as a collection of agents. But if all (controlling) components are represented as agents, MARIE's operational layering is gone and because the operational layer is a cornerstone of the MARIE robot, this makes HIVE an inappropriate approach.
What may be considered, though, is the representation of the communicated knowledge in HIVE. In HIVE the agents are described in two manners; a semantic description and a syntactic description. The semantic description is required to give agents the ability to determine what other agents are capable of, the syntactic elements then describes how to make them do it. In analogy for MARIE a semantic description of the elementary operations would describe what they can do (obtain line segments, follow a wall), a syntactical how to make them do it. This last aspect is exactly what MARIE requires, since it gives insight in the

[5] The components in HIVE are fixed, physical objects, such as digital cameras or printers, and as such really have a fixed location. For MARIE, this is only the case for the on-board computer; it also has a permanent location. However, introducing HIVE just so that the on-board computer can communicate with the rest of the software components of the robot seems rather extravagant.

specific control details of the elementary operations. A semantic description would also be useful, but at what cost? The next section will go into this in more detail.

Advantages of using HIVE:

- Separation of semantic and syntactic descriptions
- Fixed location of elements is compensated by dynamic elements that interface with them (useful for the on-board computer of MARIE)

Drawbacks of using HIVE:

- As a concept not usable, because it would require a complete conversion of almost every module into an agent
- The overhead of having a representation of a service on each machine.

5.3.3 LARKS

LARKS is interesting because it makes differently controlled components understand each other by having them 'talk' a specific language. This is accomplished by defining the components in a syntactic and semantic form, together with logical statements, that allows for reasoning about the possibilities and capabilities of the components. Separate software entities, called matchmakers, reason with these capability descriptions, obtained after the components are located, in order to satisfy certain requests. Thus providing services on demand.

The semantic aspect of ACDLs faciliate reasoning about agent information such as an agent's intentions and capabilities. The elementary operations are no agents and do not have (easily) defined intentions or capabilities. This gives the problem that, to make a semantic definition of them, a lot of research is required as to find a good representation of the elementary operations and because a semantic representation is only useful when it is generic enough to be used for multiple goals (on-line parameter planning, self-configuration, self-optimization, etc) generating a semantic ontology for MARIE is simply too much work.

Considering the fact that MARIE was never designed to describe its own capabilities or communicate about them, it seems more practical to focus on the particular properties that should be made available, in stead of providing means to allow the system to choose elementary operations in order to solve an action or have it accomplish a task, i.e. no matchmaking and no semantics, but just syntax should be defined.

Advantage of using LARKS

- Allows for sophisticated decision making

Drawback of using LARKS

- Not applicable for elementary operations
- Difficulty in creating a semantic language

5.3.4 Autonomic computing

The IBM manifesto speaks of controlling complex, heterogeneous systems and components by developing systems with self-managing elements that provide the ability to self-configure, self-optimize, self-heal and self-protect. Except for self-protection (protection has never been an issue for MARIE), self-configuration, self-optimization and self-healing are definitely improvements to MARIE, and are actually made more realistic by the changes this paper introduces. Providing the robot with more knowledge about the specific control details of its

components brings these improvements closer. However, the way to increase the autonomy of the individual system elements, by transforming the elements into autonomic elements with monitoring, analyzing, planning and executing parts, is not usable for MARIE, because elementary operations are designed to be simple units and were not meant to be autonomous. Adding autonomy to the individual elementary operations would render the action dispatcher useless for the greater part and would require a totally different way to enforce control on a higher-level.
The manifesto does provide a glance at how useful elements with more autonomy would be, but sadly does not provide means to make it so; the manifesto deals too little with how capabilities of components are to be published and used, a strong recommendation for an agent-oriented approach is all that is mentioned.

So, although the characteristics of autonomic computing, such as self-configuration and self-healing are much desired features for MARIE, autonomic computing is only useful as a very general concept. The reason for this is, as said, that IBM's approach is too heavy for the simple and small design of the elementary operations.

5.3.5 Concluding

Agent technology with ontologies, capability languages and some middleware provide means to solve problems in an autonomous fashion, with using a (potentially big) collection of agents, although this is useful for MARIE (as it would be for *any* autonomous system) it is not so easy to implement. Considering the fact that MARIE was never designed to describe its own capabilities or communicate about them and that choosing a particular service (i.e. an elementary operation) is done once per execution, it seems more useful to focus on the particular properties that should be made available, in stead of providing means to allow the system to choose elementary operations in order to solve an action or have it accomplish a task.
In general, there are two reasons why it is not recommended to use the existing systems for MARIE:

- The existing approaches produce too much overhead. Using middle-agents or brokers or something similar, requires too much communication and computation, especially for the on-board VxWorks machine of MARIE (the VxWorks machine is 14 years old). Note that such methods are meant to work in environments with a lot of agents. MARIE has only ten, potential, agents available.
- Managing control information is not just a matter of dealing with properties of agents, but also handle states, intention, goals, context, et cetera. Because elementary operations are very different from agents, since they do not have these typical agent characteristics, they can't talk about them. Defining semantic information, that allows for reasoning about how to solve a goal, such as agent-based systems do, is needlessly complex. MARIE is, presumably, better off with syntactical definitions and a human that does the task planning.

The presented frameworks also deal with the way the various components locate and deploy each other's services, by having some type of middle agents resolve requests (HIVE, LARKS). In MARIE locating is not dealt with. It is best to keep following in mind when developing a solution:

- Identification, as a combination of predefined, fixed data, and further information on request. (a client actively has to identify a service)
- Location information must be stored in a fixed location table
- Control information of a syntactical form, provided on request

5.4 Proposed solution

5.4.1 Introduction

The proposed solution will consist of an interface that communicates information about the elementary operations, so that the system has means to determine such knowledge more autonomously. A property that is certainly required is the location information of elementary operations.
In these sections the hi-level functional requirements of the solution will be defined. Described is *what* the proposed solution will do. These descriptions form the bases for the next chapter, that describes *how* the requirements are fulfilled.
Its is expected that this knowledge can be used in much more situations to the systems advantage, not just for efficiency. Section 5.5 will describe some possible future applications for the interface.

5.4.2 Overview

Chapter 4 described that MARIE lacks knowledge about the elementary operations location and control details (and the consequent problems with either of these changing). This chapter describes a solution to fill this ‘knowledge gap’. It will consist of a way to communicate information about its components by means of their properties. This will become possible by designing an interface that allows other components to obtain and publish property information of elementary operations. The idea behind the construction of another interface is that this new interface introduces knowledge on a different, higher, level than the other two interfaces of MARIE.
The first stage of development of this interface will consist of making the systems modules location independent. Locations of modules are thought to be a property that nicely demonstrates the usefulness and functioning of the property interface.

5.4.3 Functional Description of the Solution

Knowledge about elementary operation will be made available, in a generic fashion, that surpasses the other generic ways of the system to share or communicate information.
The traditional way this kind of knowledge is characterized is by defining a semantic framework that captures the inter-relationship between knowledge and the meaning thereof, resulting in a description of capabilities and a ‘language’ to describes these capabilities in, so that they can be reasoned about, see section 2.5 or [10], [12], that deal with such a languages. However, this is not what will be done in this thesis. The existing software architecture is not designed to function with such a mechanism that (in essence) uses decentralized control and reasoning, in order to fulfill goals. Developing and using it would create too much overhead for MARIE.

Properties
Identifying and describing characteristics of an elementary operation (in the context of locations and control information) can be done differently. The use of a semantic representation of the relationship between elementary operations will be omitted and instead there will be a description of the *properties* of elementary operations, of which the user, or consumer of this information, must know the interrelationship with other information.
Using properties, instead of capabilities, allows for much broader (although less structured) collection of information to be made available; properties, such as up-time and historical information can also be taken into account.

An Interface

Sharing this information between processes could be done by extending the eo-interface. But, because of the nature of the property information (its *about* elementary operations, instead of controlling the elementary operations), this will not be done. Instead, an interface will be designed that is specifically meant to communicate properties of elementary operations in a predefined manner, according to a certain protocol. Such an interface defines how information is made available and can be obtained This means that any process can make use of it.

5.4.4 Self-configuration: an application of the property interface

The more information that is known about the elementary operations, the better control can be enforced. One particular instance where this is expected to be the case is when the system can determine its own configuration. Self-configuration, however, is not the only enhancement that could result from more self-knowledge. What was described in section 5.3.4 about autonomic computing and [5] deal with typical autonomous system improvements, that essentially require self-knowledge.
Having explicit knowledge about the whereabouts of elementary operations in combination with knowledge about the specific control information for each elementary operation provides not only means to set up a system, but can also be used to recover from crashing processes or failing machines or it could be used in assembling task trees, or to check the (syntactical) correctness of a particular mission, etc.
All such application require special functionality, and this paper limits itself to the design of the property interface, not using it. However, the self-configuration application will be kept in mind as an example application of the property interface. It is considered a very useful application that can be viewed as a proper test case for the improvements that will be made to the system.

The self-configuration example
To give an idea of what properties should be communicated by the interface, self-configuration will be taken as an example. It describes how the system would benefit from the ability to chose between multiple instances of an elementary operation, and thus determine on its own what its proper configuration should be like:

> When the system is starting up one of the initialization steps is setting up connections to elementary operations. The action dispatcher is, at startup, expected to connect to every elementary operation it will use in the experiment. Currently the developer (!) must specify which elementary operations the action dispatcher will connect to and the operator must make sure that they are available.
> The new property interface, however, must make it possible for the action dispatcher to find out what elementary operations are available so that it can setup a connection, without the help from the operator.
> The idea is that, in that case, the action dispatcher invokes functions from the interface to determine the available elementary operations. Based on that information it can reason about which ones to use. If there are multiple instances of a particular elementary operations active at the same time (for example an old and a new version of an elementary operation), then the interface provides means for the action dispatcher to obtain information to distinguishes the multiple instances. This can be done by providing elementary operations specific information, such as information about their parameters, their version, up-time, etc.

Summarizing this: in order for MARIE to set up an appropriate configuration by itself, it first of all, needs a source for information on all *available* modules within the network and their *location*. Additionally, in order to select between multiple instances of an elementary operation, it needs to obtain information about other properties of these elementary operations, such as previous parameter values, version information , et cetera.

5.4.5 Functional requirement

In general, the interface is a source for properties of elementary operations, to any module of MARIE. It allows such modules to obtain up-to-date, elementary operation specific property information.

The property interface must be the source for properties, such as:

- The availability of elementary operations; so that the system knows what elementary operations it has at its disposal
- The location of the elementary operations
- The version, parameters count, parameter values, default parameter values, up-time, and various other properties.

Implementation of this functionality leads to a system where all modules can contact all available elementary operations and find out how they are controlled. The property interface assures the availability and enables the communication of properties of elementary operations. It does in no way dictate how the obtained information should be used. That is up to the developer of the particular application that uses the interface.

In order to have up-to-date and accurate information the elementary operations will be the source for their own properties. The elementary operations are required to take care of the following:

- actively supply availability information
- actively supply location information
- supply all other properties on request

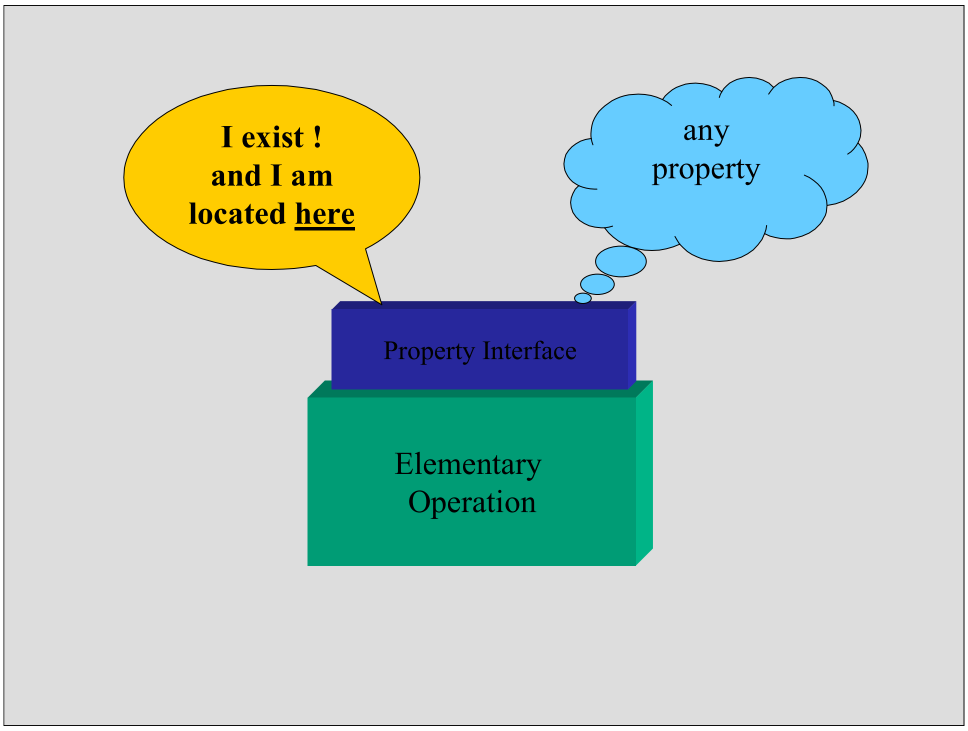


Figure 5-1: Locations are actively published, other properties are available on request

Because the elementary operations are the source of their own properties and because the location of the elementary operations should not be considered static, the requester of properties of such an elementary operations must first locate it, before it can submit its particular request. The following is required from the requester of property information:

A requester of property information must

- Locate and contact the elementary operation it wants property information from

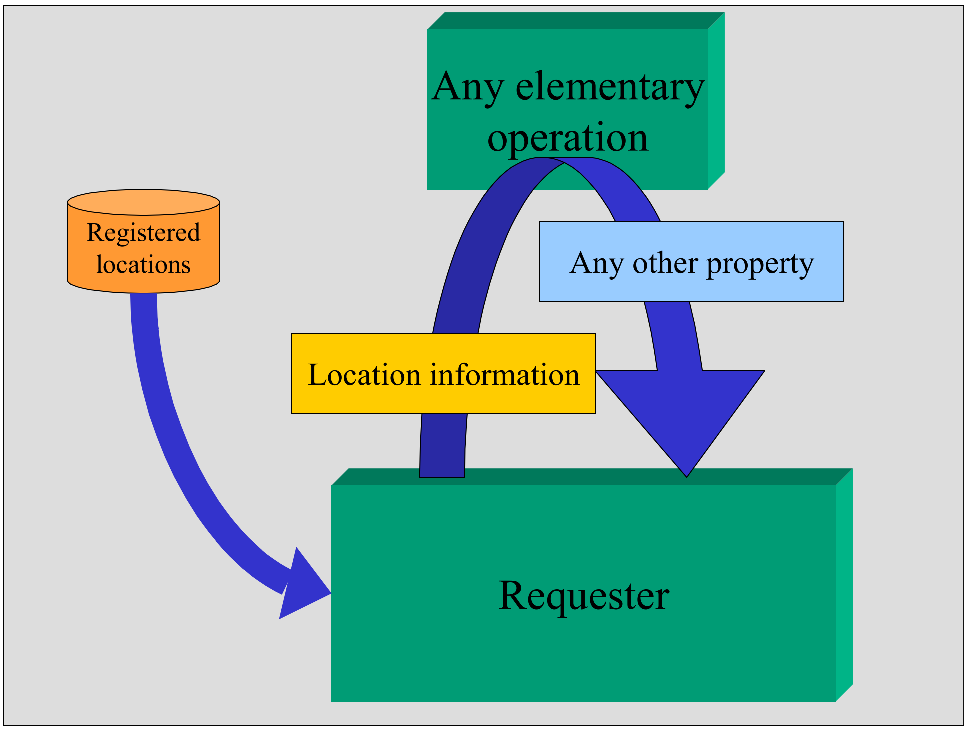


Figure 5-2: location information is required for obtaining properties

In figure 5-2 the blue arrow represents interface functionality. The requester is required to locate and contact the elementary operation for that operations property information, before it can obtain any other property of that elementary operation.

5.4.6 Towards design and implementation

At this juncture an important decision must be made concerning what will be designed and implemented and what not. It was chosen to, first of all, design and implement location independence, with the property interface in mind. This means that location independence is going to be implemented and the property interface will be designed in some detail.

5.4.7 Location Independence

Providing the system with knowledge about the location of elementary operations and making the system location independent are two different, but closely related, subjects.
Providing the location information of elementary operations does not imply location independence. Location independence, however, does require some way of obtaining location information of modules, in order to set up connections.
The locations of MARIE’s modules are hardcoded in the system, although by design all its modules are location independent. This means that MARIE, currently, runs with a static configuration, but has the potential to function with a distributed configuration. To accomplish dynamic configurations location independence of MARIE’s modules needs to be designed and implemented.

Location independence was dealt with first, for the following reasons:

- the system as a whole improves with real distribution,
- the location of elementary operations is needed, in order to obtain their properties,
- the location property can serve as an example property for the interface

These points will be briefly motivated:

- *System as a whole improves*

A location independent system is easier to use, extend and maintain. Both the developer and operator of the system will have less interaction with the system. The developer needs to pay less attention to integrating a new module, the operator will have less work configuring the system.

- *location of elementary operations is needed, in order to obtain their properties*

In order to obtain an elementary operations properties one needs that elementary operations location. Although, this in itself, is easier when locations were hardcoded, it would provide additional problems, when in the future location independence were to be implemented, because, then, also the property interface would have to be adjusted to deal with dynamic locations.

- the location property serves as an example property for the interface

For demonstration purposes the location property is a good example; it is a property that each elementary operation posses. Its gain, as extra, higher-level knowledge that can be reasoned with, is apparent in the context of determining configurations.

For the self-configuration application location independence is required. The application would have to be able to set up a configuration with two differently located elementary operations. How would this be possible if the system can not deal with dynamic locations?

5.4.8 Future applications of the property interface

Availability of properties of elementary operations is useful in various circumstances. This section will give some example applications in which property information can be useful, although tests should be performed to confirm this.
Note that the property interface assures the availability and enables the communication of properties of elementary operations, but it is up to the developer of the application to use this information.

1. Self-configuration

Depending on the available modules that are active, a proper configuration could be set up. Deciding which configuration to use, if multiple are available, could be based on information from the database of Erik de Ruiter.

2. Recovery from failure due to a new configuration.

Recovery from wrong configurations requires the ability to communicate the properties of the elementary operations in order to check the appropriateness of the send parameters and recover from errors.

3. Parameter simulation during task tree development

One step in the development of a task tree is to simulate the nodes in order to query the operator for parameters. This simulation can be augmented by the interface.

4. Monitor/request Configuration

The interface would make it possible to query the current system configuration. This information could then be used for different things ‘on the side’. I.e. evaluating a certain configuration, for example.

5. Eo-interface message debugging
A detailed log file can be generated that reports all information received through the eo-interface, by an elementary operation. This can be useful when the system did something wrong.

6. Eo-interface supervision
The eo-interface could be extended with a check of (and maybe correct) the information it passes.

5.4.9 Conclusion

The system will be extended with an interface that provides information about the elementary operations. The information about elementary operations is represented by their properties. The location and availability property will be dealt with in more detail than the others, because the expected benefit of knowing location information exceeds that of other properties. Being able to deal with dynamic location information will improve the system as a whole and is an important property for the functionality of the property interface. Because the elementary operations are the source for their own properties, the location of an elementary operation is required before properties can be requested. The elementary operations must actively publish their locations, in order for the rest of the systems modules to be able to contact them. All other properties are delivered on request, by the elementary operations.
It is expected that MARIE will benefit from the self-knowledge about its controlling components; possible future application such as self-configuration, self-recovery or self-analysis are potential users of this data.

5.5 Conclusion

The chapter looked at improving MARIE, with respect to obtaining knowledge about its controlling components, by first evaluating existing solutions. It was concluded that these do not (sufficiently) solve the problem. The drawback of the existing ‘solutions’ is that they can’t deal with changes to MARIE’s controlling software components.
The alternative solutions that are (abundantly) available in literature are either to vast for MARIE or simply not applicable. So a solution is proposed, that uses the interesting features from the alternative solutions, but does not have the overhead that comes with those systems. The solutions consists of an interface that communicates properties about the elementary operations. This interface must be developed such, that, on request, the elementary operations publish their properties. Among these properties are the operations specific control details, but also version information, or any property imaginable (and implemented).
The property interface makes the development of smart extensions to MARIE’s software possible; in the future applications that enable self-configuration, self-healing, parameter checking or facilitating task tree construction can be developed.

6 OPERATIONAL DESIGN

6.1 Introduction

This chapter motivates the decisions that were made in designing location independence with the property interface in mind. Following an overview in section 6.2, this is approached by formulating general design demands for the property interface, in section 6.3.
Section 6.4 describes in detail the collection of properties that can be retrieved with the interface. Among these are, of course, the specific control details of the elementary operations, but also other useful properties. Section 6.5 describes how the communication of properties proceeds. Section 6.7 concludes the chapter.

6.2 Overview

The information that is provided by means of the property interface gives MARIE more high-level knowledge about its own controlling software components. Any module within the configuration may request properties from the interface. The elementary operations provide the properties.
Figure 6-1 below depicts how the property interface is integrated with the other system components.

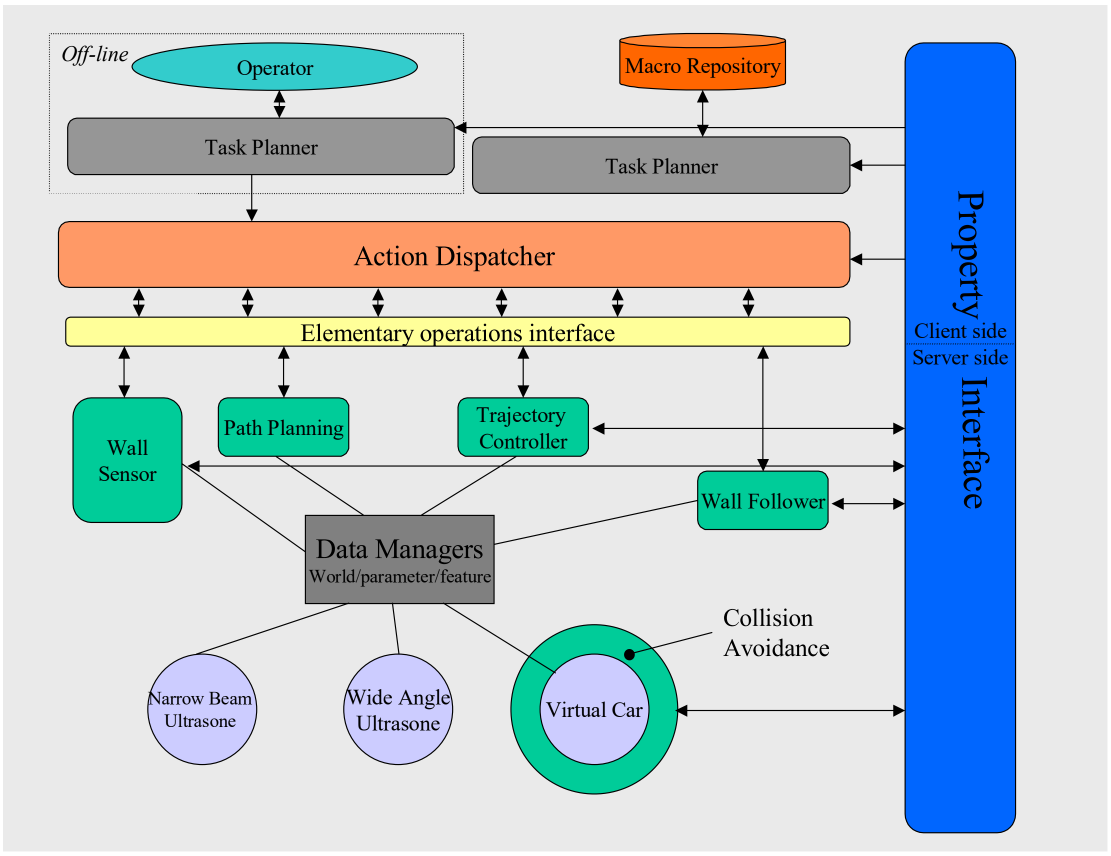


Figure 6-1: MARIE's software architecture, with the property interface

The interface is concerned with two aspects; a definition of the functions that take care of the communication of some data and a definition of that data.

The data of the property interface are the properties of the elementary operations. These include information such as versions, parameter information or up-time and location information of modules.
Properties are obtained real-time, from the elementary operations. This means that they are always up-to-date, but this also means that there is no functionality to edit or delete properties. The interface will contain a function for each property.
The location property of an elementary operation will be treated differently in this regard, because, in order to contact an elementary operation, its location must be known, beforehand. Locations are therefore reported to a database. This makes it possible to deal with the dynamic location of all elementary operations.

6.3 Design Demands

The new interface should not burden the systems or the future development on MARIE. Some general guidelines for the development of the interface are taken into account that should keep the interface manageable. They are formulated below.

- Location Independence

Location independence will be defined as a design demand. It is expected to serve the property interface well, because it demands a more generic approach towards communication. As a result of location independence the property interface will be able to deal with the dynamic location of the elementary operations.

- Simple and generic

The data communication must require minimum overhead. This means that the data exchange functionality will be efficient and with small overhead, so that the interface will not export an unmanageable amount of extra functionality. Compare this to for example CORBA or middle-agents, that both rely on brokerage of requests.

- Small Scale

Taking scalability into account would needlessly complicate matters. MARIE will always be used in relatively small and modest set up. This means that the traditional issues involved with managing objects in a large configuration are not relevant. Name space, communication load, for example, are not likely to cause any problems.

- Computational load on the operations

In general it seems a good idea to keep computations to a minimum. For the more high-level operations such as the docking spot finder or the wall follower extra computations don't influence the overall behaviour of the operation, because they operate with relatively large time-intervals. The low-level operations, like the ultrasone sensor are of more concern. Experiments should indicate if (especially the low-level) elementary operations suffer from the extra computation that results from property interface activity. If this is the case, then special measures will have to be devised for these operations.

- Generic functions

The interface should be designed such that when a process wants some property of an elementary operations it will have a function at its disposal to obtain that property, regardless of which elementary operation is 'targeted'. So, if that process needs the number of parameters that control a particular elementary operation, it should have a function available that obtains that property. If it wants to know the location of a elementary operation it should have a function for that property as well.

- Requester of the properties

The users of the interface will not be restricted to one particular process. Since the interface is more general than the other two interfaces, this would needlessly restrict its applicability. So, all virtual machines can make use of the interface and demands generic access to it.

- Source of properties

The elementary operations themselves will supply their properties to the interface. This is motivated by the fact that properties of elementary operations are liable to change. These changes can take place *between* executions, for example when an operation is moved to another computer or it has been changed, but also *during* execution properties may change, for example its last received parameters. Although the majority of the properties are static during execution, the interface will have to be able to cope when a property changes 'on the fly'; the elementary operations will contain up-to-date information and will therefore be the sources for properties.

The above mentioned design demands have a particular consequence concerning the way that the location property will be treated.
Just like any other characteristic of an elementary operations, the location information is a property. However, in order to obtain a property of an elementary operation one must know that operations location, because the properties are provided by the elementary operation itself. Therefore, obtaining the location property is done differently.

- Location Information Storage

MARIE is, by design, a distributed system that location is not pre-defined and thus needs to be determined on the fly. All locations of all active modules are stored on-line in a central database (whose location *is* hardcoded). By means of the database all active elementary operations can be found and connected to, because the operations have registered their location there when they were started..
If a virtual machine wants to obtain information about a certain elementary operation, it must first look that operations location up in that central database and then contact the elementary operation to send its request.

6.4 The Properties

Describing elementary operations in terms of properties makes it possible to express practically anything that can be said about an elementary operation and thus it enables the merging of general concepts such as version, location and control information with technical, situation dependent information such as up-time, control values, etc. The interface provides answers to question such as: where is the trajectory controller, what version is this docking spot, what was the last value send to the ultrasone sensor, how is the path planner controlled, etc.
It is not the goal to provide a complete description of elementary operations by their properties. A requester of data therefore, must not rely on the interface to do such a thing, rather, the requester of properties already must know something about the elementary operation and it 'just' needs some additional property, in order to facilitate some reasoning process.

The collection of properties the interface provides can be categorized in four groups:

- properties that together *uniquely identify* an elementary operation, i.e. its name, an identification number and its version. These must be known by the requesting process, before any other property can be obtained.

- properties describing the *location information* of an elementary operation
- *general properties* that each elementary operation has (for example properties of the parameter structure). This group can be extended in the future.
- *specific properties* that describe a particular aspect of an elementary operation and that says something about other properties (for example, the data type of a certain elementary operations' third parameter). This group can also be extended in the future.

Below is an example listing of properties. The identification and location information are generally applicable and needed by all users of the interface.
The other two tables with general properties and specific property details are expected to be useful to various applications, but the exact elements of the list are not fixed.
Distinction between the four tables, is that the information from the first two are always needed when the interface is being used. The third and fourth table are needed in the afore mentioned problem of section 5.4.4, and are, so to say, application dependent. The properties listed in those tables will therefore serve as example.

Type	**Description**	**Example**
Name	Name of the elementary operation	"TC – Trajectory Controller"
Version	A version number	*to be defined*
ID	A unique number identifying a elementary operation	1201, 1202

Table 6-1: Properties for Identification

Type	**Description**	**Example**
Existence of properties	Indicates whether a elementary operation is active in the network	TRUE or FALSE
Host machine	Machine the process is running on	"carol.science.uva.nl" or 146.50.1.20
Port number	The port the service is listening to	> 1024 and < 65535, but in practice it corresponds to the ID-property of the identification table

Table 6-2: Properties for Location Information

Type	**Description**	**Example**
Amount of parameter	Amount of entries in the control structure of an elementary operation	0...15
Up-time	The amount of ticks that a elementary operation is running	A number
...	...	...

Table 6-3: User-defined general properties

Type	Description	Example
Data type of parameters	Specifies the data type for each parameter	‘d’, ‘l’, ‘c’ (double, long, char, etc)
Default values	The default value for a specific parameter	...
Range of values	The range a value can have	...
Previous value	Last value of a parameter	...
Current value	Current value of a parameter	...
...	...	...

Table 6-4: User-defined specific properties

Properties may be described, and communicated, in two manners. One way is to communicate a single property per request, the other way is to communicate a set of properties per request.

What will be done is, that by anticipating on the likely context of the request, a collection of properties or a single property will be provided. A requester that wants the amount of parameters of an elementary operation is likely to also need the data types of those parameters, hence, this expectation means both are provided by the elementary operation in ‘one go’. This will be done in a predefined manner, no reasoning will take place on-line to decide what parameters to send. (this contrary to the way that agents, middle agents and ACDL’s work).

The little meaning that is imposed on the (relationship between) properties in this way is expected to be useful to the users of the interface and cuts down on communication, because instead of the (anticipated likelihood of) sending two property-requests only one is sent.

The following collections are made:

- Name, id, host-machine, port-number[6]
- Parameter names, parameter count, parameter data types
- Parameter names, default values, minimum, maximum
- Parameter names, previous values
- Parameter names, current values

Note that there exists the possibility that a particular parameter consists of a structure. This means that, when implementing the interface, special measures have to be devised to represent and communicate embedded structures.

6.5 Communication

Getting information from one process to another is an issue that occurs often in literature. For example, the use of middle agents as is done in various multi agents system’s [15], [10], using brokers, such as in CORBA [19], using shared memory (as was done in MARIE, but not for properties though), through mapped files, RPC [17]. These methods are either to big (agents, CORBA) or simply not applicable because of the system is distributed (file mapping, etc).
In MARIE the transfer of information has so far been done through the eo- and the dm-interface. This is like remote procedure calls, or RPC; a local function is called that has corresponding functionality on the remote machine that delivers data. Contrary to RPC, function names will not be stored in a central table. The idea is relatively simple to implement

[6] The location is a special case, see design and implementation chapter of location independence.

and fits nicely in the rest of the system. Besides, this is one of the few protocols that is supported on VxWorks.

A request for a property is made by invoking the appropriate interface function. This interface function sends a request to the relevant elementary operation. Upon reception of this request, that operation finds the property that is requested and sends it back.

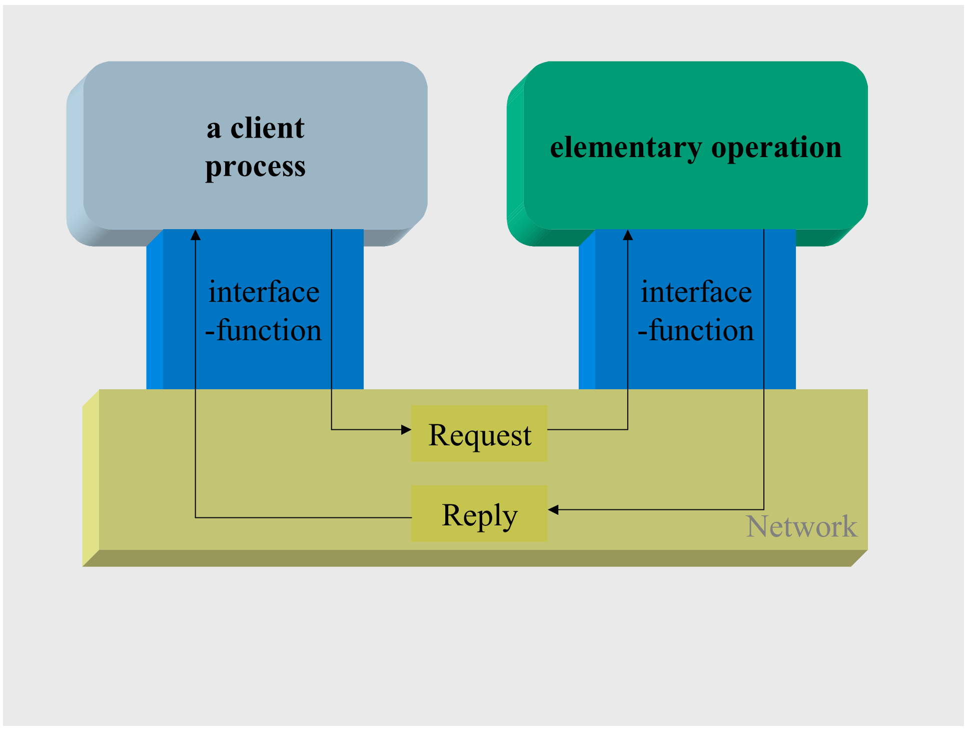


Figure 6-2: Request and reply of properties

This approach requires connections between processes. In setting up connections the need for the location property of elementary operations arises, and therefore locations are treated a bit differently from all other properties. Dealing with location information is explained in detail in the next chapters (chapter 7 and 8).

6.5.1 Request and supply of properties

A request for a property is created by invoking an interface function for that property. This function sends a message to the relevant elementary operation. On the elementary operations side a function is listening for messages. It obtains the information that is requested and sends it back. See figure 6-3.

All the interface functionality is thus represented by functions, and there is not a separate process or data-repository that manages the properties, so the interface does not maintain information. There is no need for functions to create, edit or delete data, because the source of the property information are the elementary operations themselves and their information is real-time and always up-to-data. The only property that can not be dealt with in this way is the location property. How this is dealt with is explained further on.

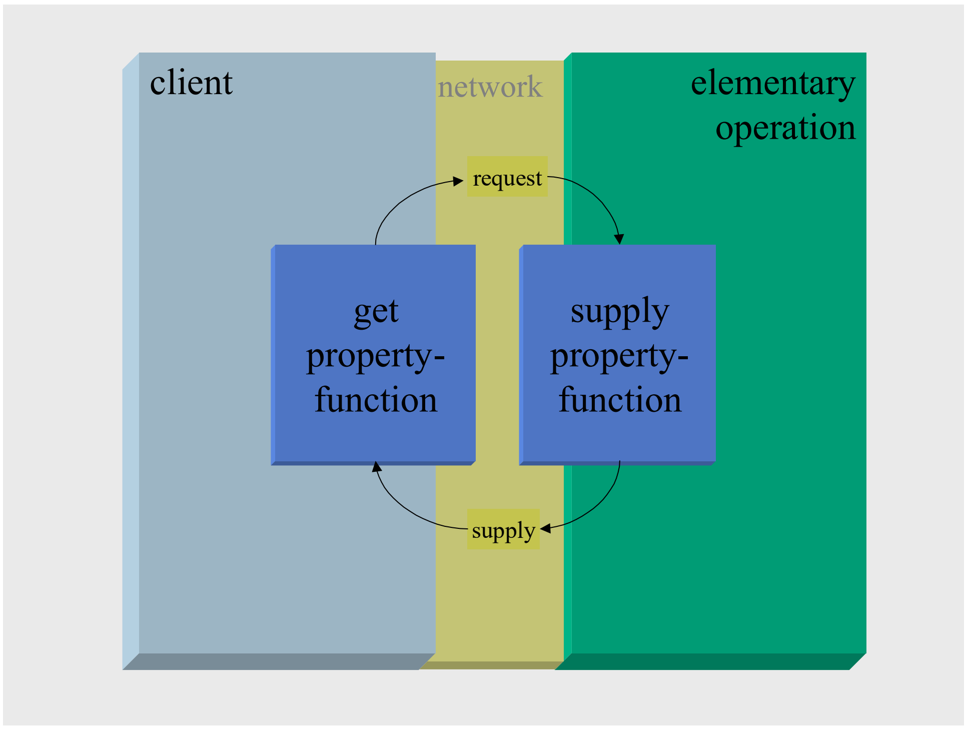


Figure 6-3: A property interface function

An elementary operation needs to provide its own properties, when requested. Because most properties are implicitly present (parameter information, location) property specific functionality is required to make them explicitly known. This makes it more difficult to make the interface generic. It is solved thus:
A certain property is always requested and supplied in a fixed manner. The elementary operations, upon reception of the property request, invokes an internal function that retrieves the property. This internal function is unique for each elementary operation, but the function that received the message is generic; it is the same for all elementary operations.

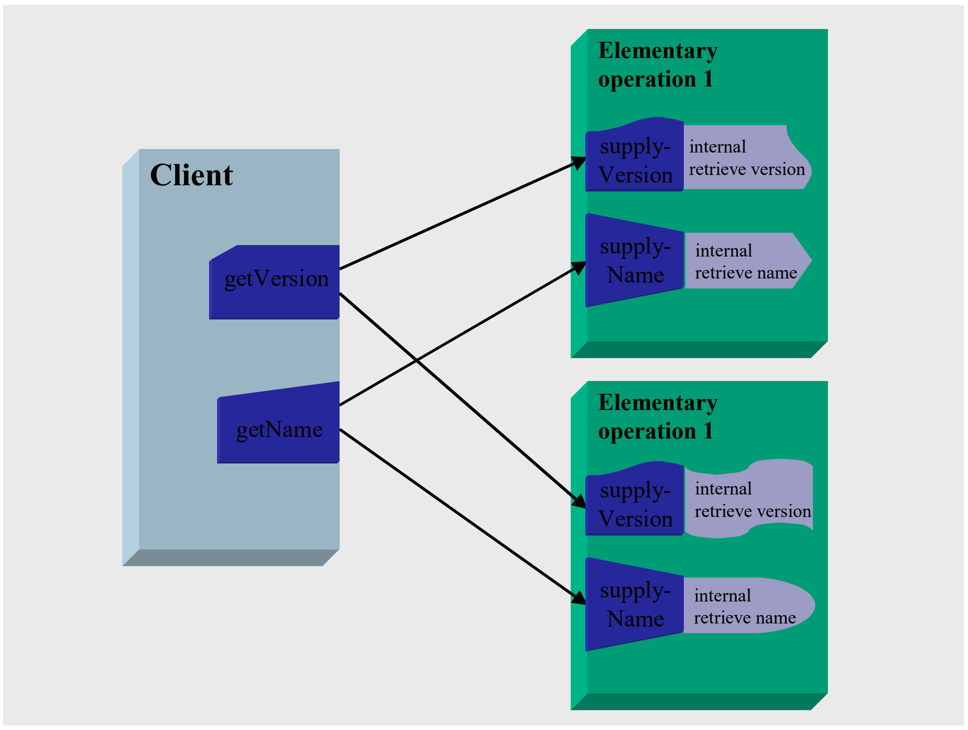


Figure 6-4: request and supply functionality of the property interface

The request and supply functionality of the properties is depicted in figure 6-4 below. Same shapes denote the same functionality, different shapes different functionality. Note how, from the clients perspective, properties are supplied in a uniform manner, but that the internal solution to obtain the properties depends on the property.

6.5.2 Retrieving properties

To retrieve a property from an elementary operation, the requester must first setup a connection to that particular elementary operation. When this connection is successfully established, the requester can invoke a specific function for that property.
This function is property specific, not elementary operation specific; it is the connection that identifies from what operation properties are requested. This approach ensures that, from the clients perspective, the interface can provide in the same properties for all elementary operations, meaning that all operations can be addressed in a uniform manner.
The invoked function to retrieve a property sends a message to the elementary operation, over the established connection, that identifies the specific property, and then waits for a reply. From the reply the necessary information is abstracted.

6.5.3 Providing properties

When a elementary operation receives a property request, the request is handled in an elementary operation independent, but property specific, manner. When the type of request is sufficiently determined, based on the type of message, the appropriate function that obtains the property, is invoked.
These internal functions are characterized by the following aspects:

- it is part of the elementary operation
- there is one for each property,
- each function is specific for all elementary operations

the internal function is part of the elementary operation
The internal function is not provided by a special process. The service of the property interface is completely contained in the elementary operations.

one internal function for each property
For each property, there is an internal function. So when a version is requested a different internal function is invoked than when parameters are requested.

each function is specific for all elementary operations
The internal function that is called to obtain a particular property is unique per elementary operations. The way to address them is the same however, which makes the interface generic.

The result of a property specific, internal functions is encapsulated in a message and returned to the requester.

6.5.4 Retrieving the location property

For the location property special measures have to be taken, since the requester of the location property cannot setup a connection to an elementary operation, before it knows its location. Therefore, when a module wants location information, it must first obtain the specific operations location from the location table. For this the requester of location information has to invoke the interface function, that was designed to obtain location information. In short, this function contacts the database, containing the location records for all active processes of MARIE, retrieves the relevant record with location information and returns this to the requester.
Chapter 7 and 8 discuss this issue in more detail.

6.5.5 Providing the location property

As with all other properties, the location of an elementary operation also originates from the elementary operation itself. However, since the operation cannot be reached, before its location is known, the elementary operation must actively publish its location information. The elementary operations will register their location at a small, central data repository, when they are initiated.
This is fundamentally different from the way that all the other properties become known throughout the system. The location information is actively exported, whereas other properties are only exported on request.
Chapter 7 and 8 discuss the location property issue in more detail.

6.6 Conclusion

Design demands are formulated that should result in a simple and generic interface. Explicitly programmed properties of elementary operations will be provided to any requesting module within the system. The assessment that the elementary operations are the only sources for accurate property information leads to the demand that they will be responsible for providing their own properties. Because of this, the requester of properties must locate the designated elementary operation, and hence, the elementary operations must register their locations at a central database.
The properties of an elementary operation are divided in four groups; identification properties, location properties, general properties, specific properties. For each property one committed interface function will be present at the server and client side. These two (interface) functions will realize the requesting and deliverance of that property.

7 DESIGN OF LOCATION INDEPENDENCE

7.1 Introduction

This chapter describes how location independence was realized in MARIE. Section 7.2 explains how location information is differently treated than the other properties, followed by an overall view of its design, in section 7.3. The way that the location information is published and retrieved is designed in section 7.4. Section 7.5 concludes this chapter.

7.2 Location independence and the property interface

In the previous chapter the general description of the property interface was described. In this chapter the design of location independence and, in the next chapter, its implementation will be presented.
The handling of the location property serves as an example of how the interface deals with properties. It does deviate from the general method of obtaining properties with respect to some aspects. The location property must be treated differently from other properties because:

- The functionality for location independence must be usable by all modules, not just the elementary operations.
- During initialization of the system, locations of the various modules must be known at an early moment.
- The client functions of the property interface require the location information of elementary operations

This adds up to a different way in which location information is made available.
Making the system location independent concerns two parts; untangling the system modules and publishing and using location information. Since the latter is of more importance to the property interface than the former, it will be discussed first.

7.3 Overview

MARIE was designed to be a multi tasking system, implemented to run on a single machine. In section 4.3.2.3 it was discussed that this causes two problems:

1. The single machine environment that the robots software ran on, caused locations of the various modules to become hardcoded. Configuration are thus static, which is undesired.
2. Due to the way memory is shared on the VxWorks computer, the actual modularity of the system was compromised as well; name-space sharing causes processes to share information through memory and that makes them dependent on one another.

The solution to the two problems is to remove the *name-space sharing dependency*, add *machine independence* and add module *availability information*.
The design demands resulting from the name-space sharing of the various modules are that all such dependencies must be removed. Doing this was rather a tiresome and time consuming activity. The description of all that was done at this stage is discussed in the implementation chapter's section 8.7.
Modules will have to publish their locations actively. I.e. on their own account, without being requested for it. This must be done at an early moment, so that the availability and location can be used as soon as the module in question is started.

7.4 Design of Location Independence

To replace the predefined or hardcoded location information a new mechanism was introduced that makes modules register itself, so that locations are known and that allows the location of a specified module to be retrieved, when a connection is desired.

When a module is started it must register its location information in a location-table. This location table is maintained by the feature datamanager. A datamanager was chosen for this, because, the datamanagers and the datamanager interface exactly contain the functionality to store and retrieve records.
Registration-routines will be embedded in the existing functionality of all modules.
The information that is registered are the name, id, machine and port information.

Figure 7-1 depicts the registration location information.

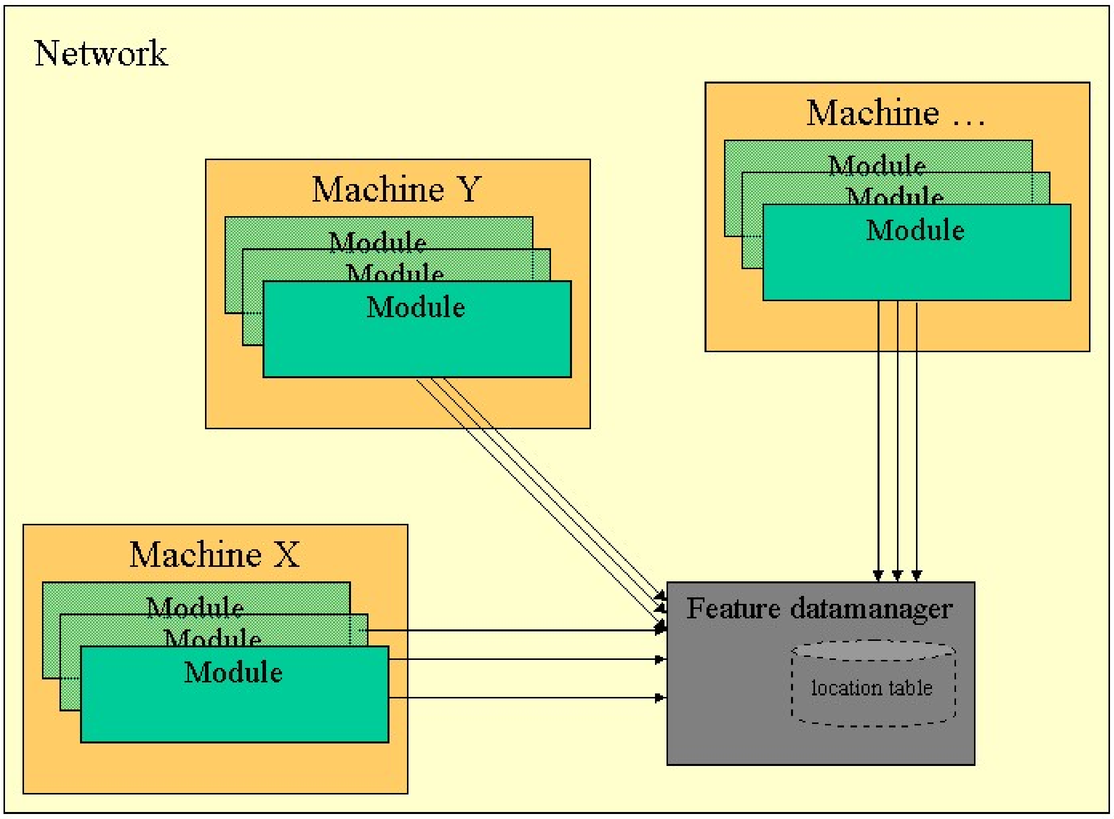


Figure 7-1: Machine independent location registration

When a module needs the service of another module it must obtain the location information, in order to contact the service, from the location table.
Obtaining location information is done by invoking the function to get the location from the property interface. This function first contacts the feature datamanager (which has a fixed location) and then invokes a function from the (existing) datamanager-interface, to select datamanager-records[7].
Figure 7-2 depicts the situation in which module A wants to contact module B. Since module B can be located on any machine within the network, module A needs to obtain the location information of module B from the location table in the datamanager, before it can connect.

[7] The dm-interface was discussed in section 3.3.5.

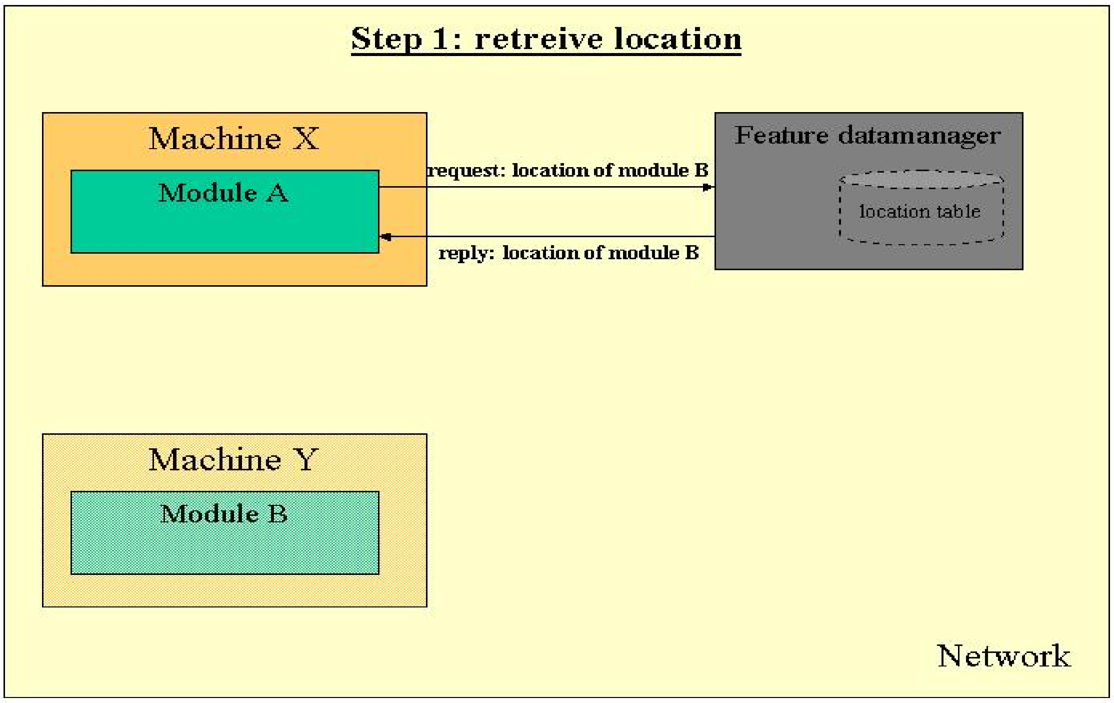

Step 1: retreive location
Machine X
Module A
request: location of module B
reply: location of module B
Feature datamanager
location table
Machine Y
Module B
Network

Figure 7-2: Retrieving a location

When the location information of a module has been successfully retrieved a connection can be set up, as shown in figure 7-3. In the event that no location information is returned, it must be concluded that the desired module is not active.

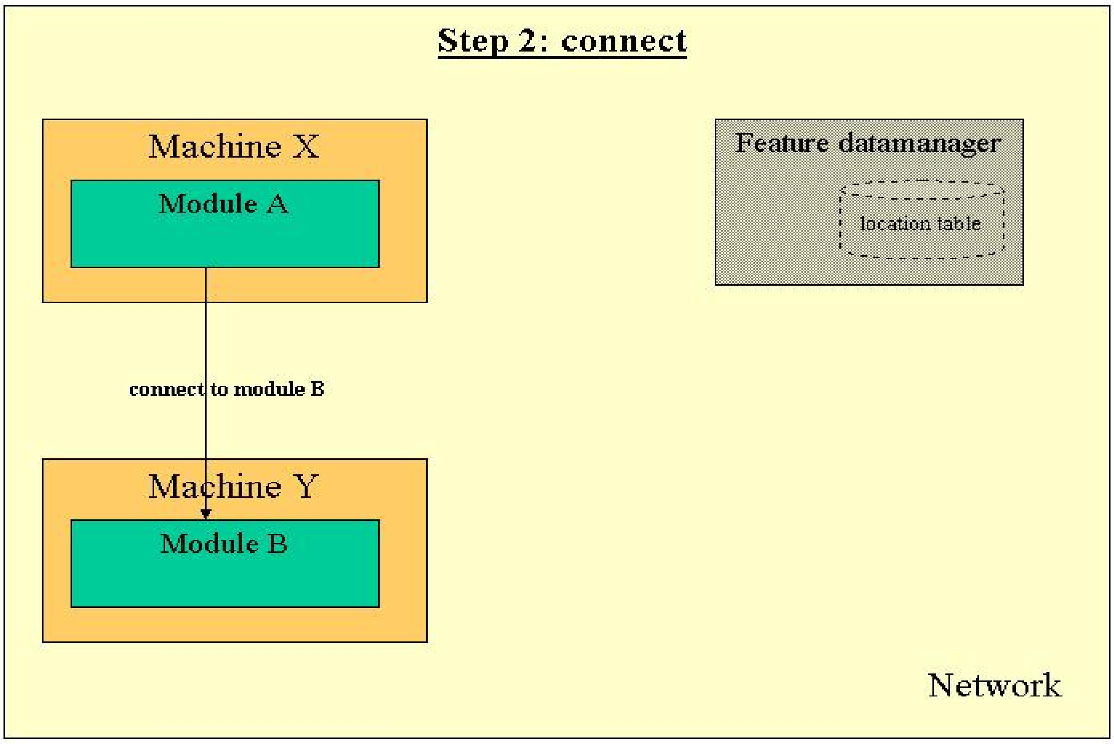


Figure 7-3: Connecting to module

Identification of a module is done by the modules name and id-number. This information could theoretically return multiple entries. For example, when multiple versions of one module are active on the network. In such cases, it is up to the requester to determine what module it needs.[8] Note that the other properties provided by the property interface could be used for this.
When determining what module to use becomes a (too) complex matter, it might be a good idea to introduce a framework of some form, to determine the best module to use, given the desired features provided by the requester. With the current design of the property interface, making decisions is up to the requesting module itself; reasoning is not expected to be done by a third party.

7.5 Conclusion

In order to allow for a more distributed usage of MARIE's software, location independence of the modules was needed. This chapter described the design that accomplishes this.
In the original implementation, locations were predefined; they either were present in a globally available list or were just hardcoded.
The new method, that is designed in this chapter, requires that modules register their location when they are started, so that other modules that want to contact them can retrieve the location information, in order to do so. This mechanism should also be used for contacting the elementary operations when a property is requested.
Registration of modules is done in a location table. This location table is a design concept; in practice locations are stored in the feature datamanager, whose location is the only hardcoded location.

[8] This would be no issue if modules could register their version. However, the absence of version management makes that impossible.

8 IMPLEMENTATION OF LOCATION INDEPENDENCE

8.1 Introduction

This chapter describes the implementation of location independence. Section 8.2 will give an overview. Section 8.3 describes the details of registering location information. Section 8.4 explains where and when, in the original source code, the location information is obtained, section 8.5 explains how this works. Section 8.6 discusses the removing of location records. The various modules were entangled in a number of ways, preventing per-module compilation, or resulting in failure of certain modules. Section 8.7 describes the encountered problems and solutions regarding when untangling the software modules.
Section 8.8 concludes the chapter.

8.2 Overview

To make MARIE's software modules location independent, the following must be taken care of:

- all inter-modular dependencies, that do not use network communication, are removed
- registration of location information
- that the location property is obtainable with the property interface –functions.

Although the first issue was by far the most time-consuming, it does not leave any marks in the final sources. Therefore the explanation of what problems were encountered to restore modularity, are discussed at the end of this chapter.
The second issue, having modules register their location, is explained in detail in the first part of this chapter. The third issue, that makes properties available by the property interface, is described in the next chapter, because it is, by design, functionality of the property interface.

8.3 Registering Locations

On startup modules register their host machine and port number to the location table in the feature datamanager, this is done during initialization of the module.
Since the location information of the feature datamanager cannot be obtained from the datamanager itself, it is predefined, hardcoded, information, making it the only hardcoded location currently in the system.

Registration of locations happens through the use of a generic function, called `location2dm(Id, port)`. This function is embedded in each modules startup routine (and thus called by each module). The 'id' –argument specifies the user-defined unique module number and the 'port' –argument specifies the port where the process is accepting connections on. (The host-machine is determined automatically).
The goal of the function is to send the location information to the location table, that is maintained in the feature datamanager.

The steps `location2dm(Id, port)` takes to register the ID and port are:

1. connect to feature datamanager, if that went OK
2. construct location –record
3. add the location –record to the location table, if that went OK
4. disconnect from datamanager

See figure 8-1.

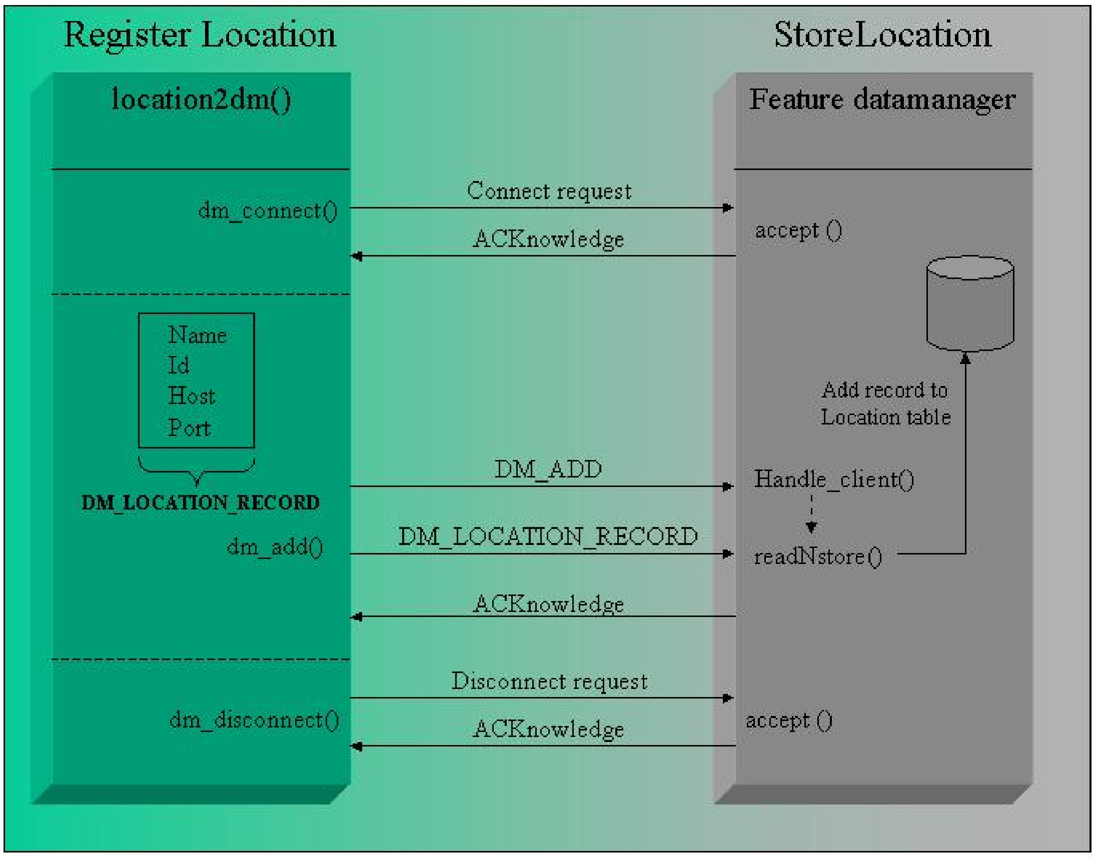


Figure 8-1: location2dm, registering location information

The block on the left depicts the location2dm()-function that is called from a module that registers its location. The block on the right depicts the datamanager, that stores the location information.
Appendix B contains the commented sources of location2dm().

8.4 Using the location property

Originally when a module needed to connect to an elementary operation or datamanager, it just called the interfaces' `eo_connect()` or `dm_connect()` function. These functions looked-up the location information in the predefined global location table and then set up the connection.
In the new situation looking up locations has become part of the `connect()` functions: instead of getting the location information from a predefined table, it is gotten on-line from the datamanager.
Below the pseudo-code for the connect-function is shown (the dm-connect function works similar):

```
Eo_connect(struct ident)
START
     getLocation(struct ident) // this is added

     If (Connect(ident->port, ident->machine) == SUCCESS)
          Return(fd)
          Else
                    Return(Failure)
END
```

The bold line is the only addition to the existing code of MARIE, as far as using the location information goes. How the getLocation function works and where it gets its information from is explained in the following.

8.5 Retrieving Locations

When a module needs the location of another module it must use the interface function to obtain that location. This function is called getLocation().
The requester calls the getLocation() –function. With ‘Struct ident’ as argument. ‘Struct Ident’ is a structure that only contains an id-number of the desired module. On return it will contain the id, name, machine and port of the desired process.

The steps that the getLocation()-function takes to obtain locations are:

1. connect to feature datamanager
2. call the dm –interface function with appropriate query (dm_select)
3. translate reply into answer
4. disconnect

See figure 8-2. The block on the left depicts what is done by the getLocation() -function that is called from a module that registers its location. The block on the right depicts the datamanager, that contains and supplies the location information.
Refer to appendix B for commented sources of location2dm().

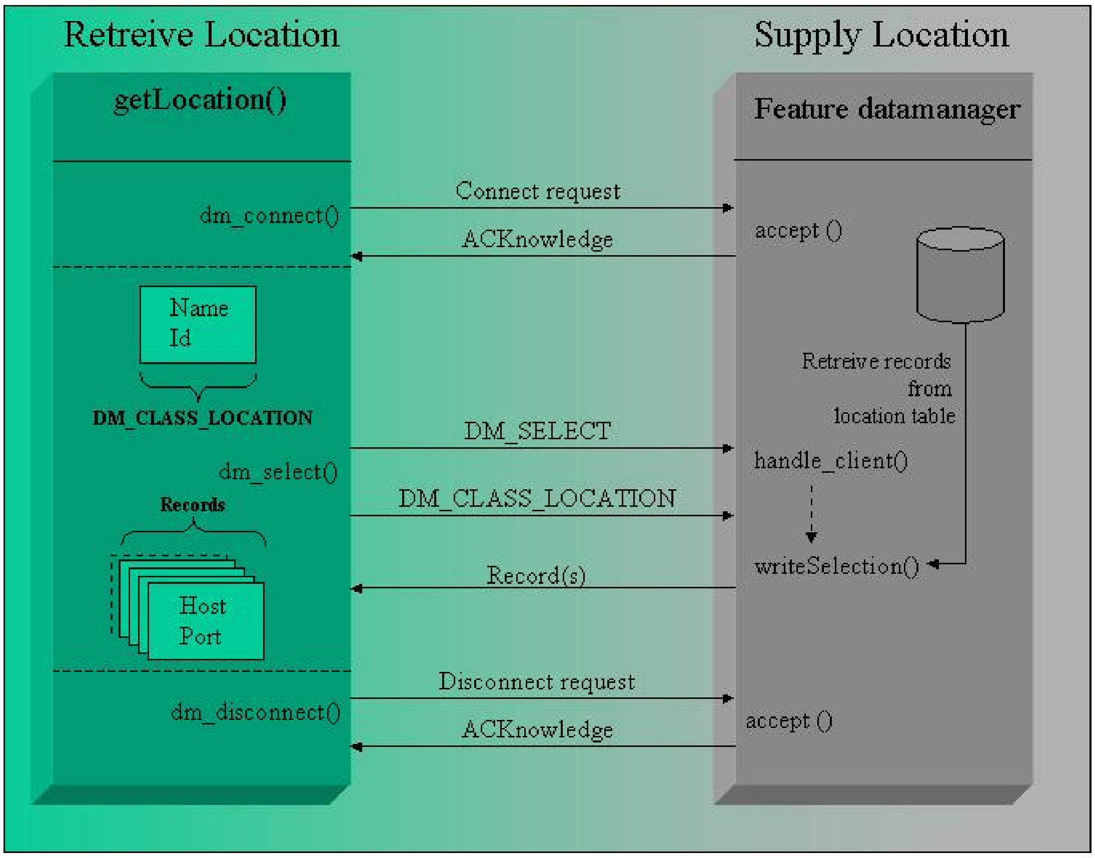


Figure 8-2: Retrieve Location

If there are multiple modules with the same id-number registered in the location table, then all of them will be returned. In the third step one particular record is selected from all the returned records, based on the order in which they were stored. This prevents selection of a particular record on some other basis, and that is not desirable, in the light of future applications, when, for example, based on specific control details or versions a selection is desired.
It would be better if getLocation() would return all location records that were found, and the caller of the getLocation()-function should then determine what records it wants. This is future work.

8.6 Removing Location Records

In the design of MARIE, the event of a module terminating or being shutdown, has not been taken into account. This means that modules do not have a destruction-routine. There is no mechanism to automatically terminate another module or have a module terminate itself. This means that one cannot deduce with the original methods of the system if a module is still active within the network.
This has implications for the location independence implementation. For, in theory, a module may have crashed, or could be stopped by the operator and the datamanager would incorrectly contain a location record for this absent module.
One approach to solving this issue is to implement some sort of heartbeat, that make the modules inform the datamanager, every now-and-then, that they still exist. This requires, from the datamanager, functionality to be able to understand these heartbeats. Such functionality is not present at the datamanager and therefore another solution will be sought.

Another approach would be to let the datamanager 'check' if modules are still present, but this also requires functionality on the datamanager side that is not there.
The way that this issue will be solved is to have modules, that discover *false* location information in the datamanager, report this to the datamanager, by removing the entry. The observation that an entry is false can be made when setting up a connection fails
Nothing has been implemented to inform the system of the existence or deletion of location records from the datamanager. Currently it is up to the operator to keep the datamanager information correct.

8.7 Encountered Problems and solutions

During the implementation phase of location independence various problems were encountered. Most of these had to do with the way that data is being shared between processes in memory. This section describes these problems and their solutions.

8.7.1 Machine Dependence

All modules were written so that they could run on the onboard computer of MARIE . To make distribution possible, modules had to be moved from that machine to other computers within the network. However, this was not possible for all modules; the low-level modules, such as the Phillips motion controller and the ultrasone sensor are hardware specific and therefore remain bound to MARIE's board computer.
The virtual cart software, does not depend on the hardware of MARIE, but can also not be moved, because, its functioning relies heavily on *direct* function calls to the mentioned hardware dependent modules.
Thus the following modules remain fixed to MARIE's onboard computer:

- Phillips Motion controller
- Ultrasone sensor
- Virtual Cart (the info-task and the controller-task)

8.7.2 Porting to UNIX and Linux

The issue about machine dependence is closely related to the porting issue. All modules were already written so that they could function on UNIX (or previous porting was done), except for the VxWorks modules mentioned in the previous section. Certain sources are explicitly written for the VxWorks operating system, such as the ultrasone sensors, the Phillips motion controller and the virtual cart. These need to be run on the on-board computer of MARIE, because they directly interact with the hardware.
Due to external reasons the software had to be able to run on Linux, so porting needed to be done. Little code needed to be changed; only deprecated and for backwards-compatibilities sake still supported, UNIX functions, that were never available for the Linux OS, because of their obsoleteness[9].

8.7.3 Shared memory

Originally, when MARIE was developed, there was no data being shared by processes of the system, but throughout the years, various changes in MARIE's software brought this about. The result is that the situations in which data is being shared, seems 'ad hoc' and only occurs at some places.

Data sharing between separate processes through memory was encountered in the following situations:

[9] This had to do with the UNIX thread library and the Linux pthread library.

- Connection descriptors

On startup the initialization file indicated what datamanagers and elementary operations were to be used during the experiment. With each of these modules a connection was set up. The descriptor of these connections where kept in two globally accessible tables: connection to data managers were stored in data_link-table and connection to elementary operations are stored in elem_obj-table.
Whenever communication took place between an elementary operation or datamanager and the action dispatcher, these connections were retrieved from the tables and then used for this purpose.
Since the new changes do not guarantee process to be run on one machine, it is not possible to use one connection per elementary operation or datamanager. In the location independent version of the system these tables will also be used, but must be filled differently.

- Linking object files

This is best explained with an example. Take the simple operations. For each simple operation a separate function is defined. In a functional sense the simple operations are completely different, and unrelated to one another, and so are the modules that use them. This means they use different variables, some of which are declared in external source files. Now suppose a module needs a simple operation that does not use those externally declared variables, it must include that external source file anyway to make sure the external variable exists.
So if one wants to link an elementary operation that uses a simple operation, then also all other simple operations are linked, including all the files these simple operations need *and* all the files that those files need, and so on. Meaning that to building that initial elementary operation requires compilation and linking of practically *the majority* of source files.
The solution consists of removing the dependence on external variables.
This has posed by far the most time-consuming obstacle to overcome and has not totally been solved.
An example of where it is still left intact are all the elementary operations that have communication shells at the action dispatcher level. Functions that are used by these elementary operations were declared in the communication shell-source files. Which means that these have to be linked with the elementary operations themselves. That's functionally not correct, but has no influence on the elementary operations behavior, and therefore has been left in place, for the time being.

- Collision Avoidance mode

This has to do with three modules: the collision avoidance module, the virtual cart and a third arbitrary module that can change the collision avoidance mode (for example the trajectory controller.)
Collision Avoidance can be set to 4 different modes, indicating how the robot is to respond to obstacles. This value is stored in a variable that was shared throughout memory. Each modules could access and change the variable (if it ran on the same machine as the collision avoidance module). This was mostly done by the Trajectory Controller. Since it is no longer a demand that the Trajectory Controller and the Collision Avoidance run on the same machine this information exchange must be changed.
It was worked around by adding the collision avoidance mode to the information structure that the Virtual Cart communicates. This structure contains all sorts of information about the state of the (virtual) cart. In addition, it now also contains the collision avoidance mode. Because the structure can be obtained and altered remotely, this solves the dependence between the two modules.
For example, the trajectory controller wants to change the collision avoidance mode. To do this it now invokes the collision avoidance interface function originally designed for this (i.e.

TcEnableAvoidance(desired_ca_mode)). This function (belonging to the collision avoidance module sources) will communicate and set the new value in the vc-struct. When the tc or vc or any other module needs to now the value of the ca-mode it can retrieve it with a similar function.
This is not a neat solution, though, since the ca-mode is a typical collision avoidance-property and not a virtual cart property. However, there is no functional difference and so this solution is acceptable, for the time being.

- Process Id's

Process Id's are used by the Data Managers and Elementary Operations to verify connections and when reporting errors during communication. A message send by a client contains the process id of the client. This makes it possible for the server to see if it is still communicating with the process that set up the connection.
The id's were stored in one globally accessible table, that links them to names in order to make error messages more readable. On startup, a client stored its process id and name in these tables. Because all processes ran on the same machine all the servers had to do was use the table to get a matching name with a process id.
Since modules no longer have to reside on the same machine this mechanism fails. It is solved by keeping a table on each machine that runs a dm or eo server. When a client connects to an eo- or dm- server the first message it now sends is a name-message, that contains its process id –number and name. The server reads these values and stores them in a table.

- debugging information

MARIE's software uses an intricate mechanism to specify the level of debug-information that is logged. This debug-level can be set per source file.
The mechanism relies on a collection of global variables. These were defined in the main file of the big executable that was loaded on the VxWorks machine. Because there no longer is one executable file, the definition of the debug information had to be moved to each individual executable.

8.7.4 Location dependence

All communication between processes on MARIE happens through socket based connections (besides from those processes that wrongly use memory sharing for communication). To establish a connection to another process, the location (machine and port number) where that processes reside, must be known. This information used to be either predefined or hardcoded:

- *Predefined Locations Information*
  Predefined location tables were used to store the expected location of other modules. If a process wanted to connect to another process, it would look up the expected location and try to establish a connection, regardless of the existence of the process. If establishing a connection to a process failed, this would be reported, but no other action would be taken. Failure of the experiment, due to lack of a vital process, was likely.
  A special case is the way the eo-shells get connected to their respective eo's. These shells had no means to set up a connection, but used the connection that the action dispatcher was using; on initialization the action dispatcher starts communication shells (i.e. the eo-shells) for all elementary operations. These shells use the same connection to elementary operations that the action dispatcher uses when sending commands to it.

- *Hardcoded Location Information*
  The low-level modules, such as the virtual cart and the ultrasone sensor connected with hardcoded information in the function calls, i.e. connect("local_host", 1234). The low-

level modules that connect like this are the system dependent modules, such as the Ultrasone, the Phillips motion controller and the Virtual Cart. Because of the machine dependence of these modules, this method of connecting is no limitation to the location dependence (i.e. localhost will always be correct). Nevertheless, the method was changed to be consequent with the rest of the system.

8.8 Conclusion

Implementation of location independence for MARIE's software modules involved two issues. Untangling the modules inter dependencies and adding a registration and using mechanism of location information. Untangling the system was done with respect to data sharing between processes, porting to Linux was done, and pre-defined location tables were removed.
The registration of location information is done at startup; modules register their location at the feature datamanager. A module that needs that module's location obtains it there, by specifying the identification number of the desired module. If the requested information is available, the location is returned and the requester then knows of that modules availability.

9 EXPERIMENTS

9.1 Introduction

This chapter describes the experiments that were performed and explains additional experiments for when the property interface is implemented.
Section 10.2 gives an overview of the experiments to be conducted. Section 10.3 describes the tests that were performed to ensure the consistency of MARIE's old functionality with location independence and describes other experiments that were performed.
Section 10.4 describes experiments to test the capabilities of the property interface with an example application.

9.2 Overview

This chapter first describes experiments that tested the proper functioning of the system with location independence. The experiments are designed to check if MARIE can still do the same things with location independence as it could without it. This is done by executing the Docking Spot mission, of which a complete set of results are documented [21].

The second series of experiments should test the property interface by investigating the robots ability to select between differently located instances of one elementary operations.
It is expected that, for the application of self-configuration, just location information is not enough to decide what elementary operation to use; there is a need for more information about the available properties. This will be tested in the next experiments, where the property interface comes in play; this third series of experiments test the robots ability to configure itself, based on predefined set of criteria, that require information into the elementary operations properties.

9.3 Consistency experiments

9.3.1 Experiment

The tests that were performed are designed to verify the continuity if the system, making sure that MARIE still functions with location independence, as it did before.
The mission that was used to test this was the Docking Spot –mission. This mission has MARIE follow a wall until a docking spot is found. Then the cart should park in that docking spot.
As said, the goal of the test is not to investigate MARIE's capabilities when it comes to maneuvering in the world, but a test to see if the changes made to its software, haven't affected the robots behavior.
The only changes that were made to MARIE's software, in a functional sense, concern setting up connections. This test should confirm that the setting up of connections is done correctly and that nothing else has inadvertently changed.

The reason for choosing the docking spot-mission is:

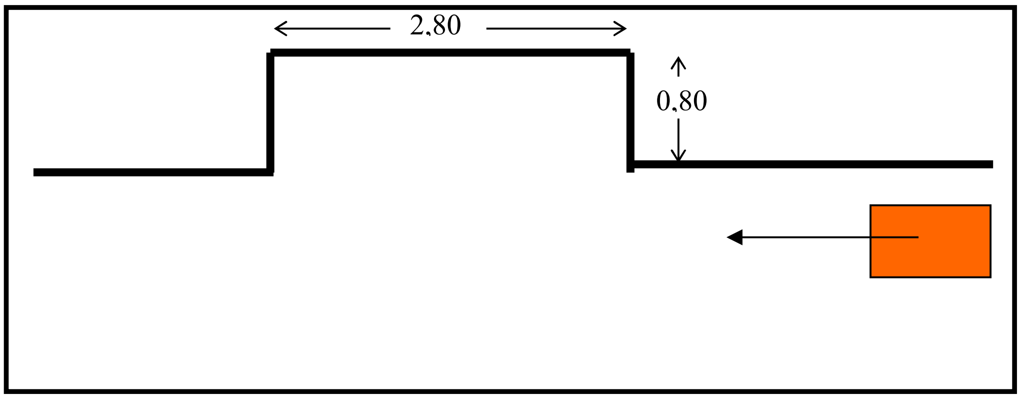
- there are previous test results available,
- that it requires the use of most elementary operations
- the mission can be executed with the "command interface", developed by George de Boer.

As was explained earlier, the command interface is an alternative way to execute missions. Instead of relying on the task planner to translate a mission description in suitable tasks and actions, direct calls to elementary operation functions are invoked, making the detailed mission description unnecessary[10] and tests more easily executed.

Test Setup
Software
The test was performed with the following modules:
- Action Dispatcher,
- Wall Follower, Docking Spot Detector, Path Planner, Trajectory Controller, Segment Extractor, Collision Avoidance, Virtual Cart, Ultrasone Sensors, Phillips Motion Controller
- Feature-, world- and parameter –datamanager.

The software ran distributed over 3 machines.
1. vw1.science.uva.nl (the cart): host for Virtual cart (and collision avoidance), the Ultrasone Sensors, the Phillips motion controller
2. rijn.science.uva.nl; host for feature datamanager
3. signor.science.uva.nl; host for action dispatcher
4. signor.science.uva.nl: host for all elementary operations and the parameter- and world datamanagers

Environment
The experiment was performed in the lab of Wetenschappelijk Centrum Watergraafsmeer. By placing wooden segments a wall and docking spot are created, as shown in the figure below. The docking spot is 80 cm by 280 cm. The carts starting position is at the far right of the wall.



The experiment
The test consists of 4 stages.
1. initialization. Of importance are the modules registering and retrieving locations for initial connectivity
2. Finding the wall
3. Following that wall until a docking is found
4. Parking in the docking spot

When any of these stages should fail (for example, no wall is found) full stop should be executed. Figure 9-1 fives graphical representation of this.

[10] This approach makes no use of the task planner. This has not been tested, but there are justifiable reasons not to do this; the task planner is not affected by location independence because this is the only module that already incorporated location independence.

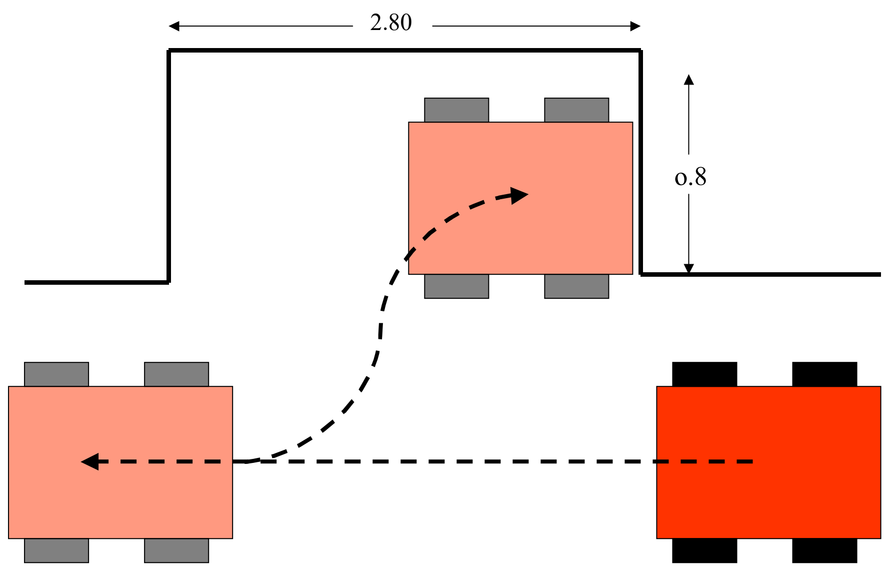


Figure 9-1: MARIE finding a docking spot and parking

9.3.2 Results

The test was performed a number of times and initially came back with wrong results, due to various reasons. Most notably the aspects mentioned earlier as the memory sharing deficits, in section 8.7 were the cause for these.
Another reason for the experiment failing was to do with the size of the docking spot that MARIE expected. It turned out that MARIE was very 'sensitive' to the this value; the docking spot was supposed to be rather large before MARIE would recognize it as such, otherwise it would just drive past it.

Connections that were made
By checking and combining the output of the various processes one can deduce what connections were made.
The action dispatcher requests all registered modules from the feature datamanager and connects to them. This is reported in a log file, like so:
The modules registered at the feature datamanager were:

```
ad_init: Found the following records:
EO found 0: ID: 1516 port: 1516 machine: 146.50.1.211 name: US - Ultra sonic system.
EO found 1: ID: 1511 port: 1511 machine: 146.50.1.106 name: TC - Trajectory
controller.
EO found 2: ID: 1513 port: 1513 machine: 146.50.1.106 name: SC - Sensor-based
controller.
EO found 3: ID: 1512 port: 1512 machine: 146.50.1.106 name: WS - Wall-sensor.
EO found 4: ID: 1515 port: 1515 machine: 146.50.1.106 name: PP - Path planner.
EO found 5: ID: 1518 port: 1518 machine: 146.50.1.106 name: DD - Docking spot
detector.
EO found 6: ID: 1517 port: 1517 machine: 146.50.1.106 name: SD - Line segment
extractor.
DM found 0: ID: 1201 port: 1201 machine: 114.105.106.110 name: DmFeature.
DM found 1: ID: 1202 port: 1202 machine: 146.50.1.106 name: DmParameter.
DM found 2: ID: 1203 port: 1203 machine: 146.50.1.106 name: DmWorld.
```

Figure 9-2 combines the above information in a graphical representation of the connections that were made.
Change in picture: one arrow to a datamanager from the virtual cart, dunno about the others.

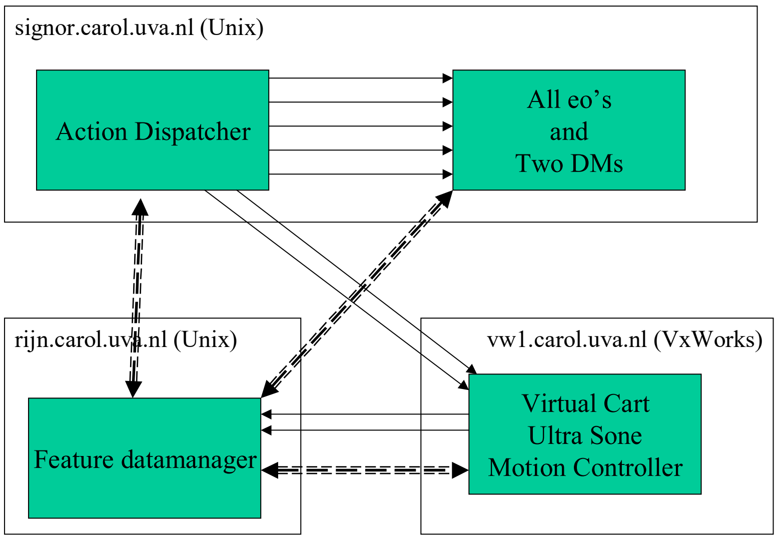


Figure 9-2: connection made, for the docking spot mission

The broad arrows indicate the registering and retrieval of location information.
The thinner arrows represent the connections that were made and for which location information was looked up.

9.3.3 Other experiments performed

A number of additional experiments were performed to investigate to what extent MARIE's processes are independent of their location and the operating system of the machine.

This served three goals:

- To show that processes do not have to run on one specific machine
- To show that processes are not dependent on a specific location of other processes
- To show that processes run both on the Linux and Unix –operating system

The following tests were done:

- Running the action dispatcher separate from the elementary operations
- Changing the hardcoded location information of the feature datamanager
- Running the action dispatcher, the elementary operations and the feature datamanager on a Linux System (arena.science.uva.nl)

Table 9-3 shows on what machines modules were tested (and ran correctly).

Operating System	Name	Module			
		ad	eo's	feature dm	vc/us/pm
VxWorks	vw1	x	x	x	x
Linux	arena	x	x	x	
Unix	eenlx	x	x	x	
Unix	carol	x	x	x	
Unix	signor	x	x	x	

Figure 9-3: Machines modules were tested on

All modules except for VxWorks dependent modules (the virtual cart, the ultra sonic sensor and the Phillips motion controller) function normally on both the Linux and the Unix operating system.

9.3.4 Conclusion

The experiments have shown that old functionality is preserved with the new location independence implementation. Additionally it is shown that modules (besides from the low-level VxWorks software) can run on any machine within the network, without compromising functionality. More experiments must be performed concerning the flexibility of location independence, for example, the influence of big 'distances' between processes (i.e. hops). An additional observation must be made concerning the particular machine the location table is hosted at. As was explained the location table is implemented as a data class for the feature datamanager. It is currently not located on the VxWorks machine of MARIE. Since this machines hostname is not likely to change in the future, it is advisable to move the location table to it, because, in practice it is the only location whose name is fixed.

The implementation of location independence decreases the use of the system for both the developer and the operator. The developer can, much easier than before, test and debug elementary operations. The operator no longer needs to compile software to change configurations. A benefit for both is that log files have become separate from each other, although these can still be improved.
The downside of this test is that it doesn't demonstrate the functional benefit of location independence and the property interface, hence the following experiments, were devised.

9.4 Functionality experiments

The hypothesis is that the property interface increases autonomy and decreases user-interaction. How to test this hypothesis?
Because the interface only provides data, an application is required that uses the properties supplied by the interface, and that replaces some activity of the operator.
This section will describe such an application and experiments that should be performed with it. Since there is no implementation of the property interface the application is also not implemented and experiments are not actually performed. This remains future work.

9.4.1 The Application: self-configuration

The task of the self-configuration application is to set up a working configuration from the collection of available elementary operations. The situation is one in which there are two instances of a particular elementary operation; one is the 'old' version and one as the 'new' version. Based on their properties the system should decide which one of these two elementary operations to use.

Configuration must take place during initialization, before any action is executed or any elementary operation is activated. The place where the configuration must be determined is the action dispatcher.
Suppose there is an old and a new version of the wall follower active. The action dispatcher would do something like this:

1. locate all modules with the wall-follower ID
2. connect to all modules with that ID
3. obtain their relevant properties, using the property interface
4. decide which one to use, based on those obtained properties

Step 3 and 4 are the essential steps here.

9.4.2 The experiments

The success of a configuration can not be measured in a numerical sense. What can be done is show that a configuration works, and, even, that it works better than another configuration. Comparing the results of executed mission enables this. For example, one might look at the time it took to complete a mission. Or alternatively the number of exceptions that occurred can be taken as a measure. The aspect to base the performance of a mission on depends on the goal of the mission, and on what is being tested. Since in this case some configuration is subject to tests, comparison must be based on the performance of the configurable components. These components, obviously, are the elementary operations, and more specifically the two instances of the specific elementary operation that must be chosen from.

<u>Experiment 1</u>
Previously it was established that MARIE operates properly with location independence. This experiment should show that MARIE is *capable* of deciding what operation to use, when there are two instances available. The criteria to base this decision on do not matter, at this point, as long as it uses the property of the two instances for this.
For a given mission, three experiments should be conducted with the following elementary operation running:

- the wall follower and corridor follower [3]
- only the wall follower
- only the corridor follower

The action dispatcher should decide what elementary operation to use, based on a property it obtains via the interface. The distinguishing property of the wall follower and the corridor follower, is that the latter needs one parameter more than the former.
A test is considered successful when a configuration was established. A configuration is considered successful if the chosen elementary operation is part of it. More specifically, this means that the action dispatcher made a connection to it.

When the previous test was successful, an experiment can be conducted that checks if the autonomy of the system increases and the operators influence decreases.

<u>Experiment 2</u>
The experiment that is devised to check that the autonomy will increase and that the user interaction will decrease should extract the desired configuration from the information in the task tree, instead of the configuration being determined by the operator.

The task tree contains the parameters it will sent to the elementary operations, in addition to their id-numbers, this is information that can be used to reason about which elementary operations, will participate in the configuration, also, when there are multiple instances of a particular operation available.
Again, this experiment is conducted with the wall follower and corridor follower. These elementary operations can be distinguished on basis of the amount parameters they expect, the wall follower expects one parameters and the corridor follower expects two parameters.
In case the task tree contains, 6 parameter the action dispatcher should decide to include the wall follower in its configuration, and when it contains 7 parameters it should use the corridor follower.

9.4.3 Results

Note that the experiment is meant to test the interface and not the application. Although it is the application that (theoretically) decreases the user interaction with the system, it is the interface that allows those applications to exits in the first place.

The property interface has not been implemented, and so the devised experiments cannot be performed. However, if there were an implementation of the property interface, and an application using it, the above mentioned experiments can be done.

10 Discussion

The MARIE robot was improved by introducing location independence for its modules and a design for a property interface, that provides information about its controlling components. Location independence is clearly an improvement to the system, because the operator is greatly facilitated by the dynamic manner in which configurations can be set up, and also from a design point of view location independence makes things easier; testing and debugging new components will now be much less work.

The question whether or not the property interface is an improvement to MARIE and satisfies the characteristics of a source for meta-knowledge is a more delicate matter. Before discussing this issue the following must be noted:
The real usefulness of the property interface can only be determined when it is implemented and experiments are done with it. This is, for the moment, not of importance, since it is solvable by time and although it will result in the final answer to this paper' hypothesis, it is useless to discuss now. It just means that the discussion takes place in a theoretical context.

The following points of criticism about the property interface will be discussed:

- *The property interface is no source for meta-knowledge*
  The first point of criticism is concerned with the fact that although MARIE contains several modules, the interface only provides properties for a subset of these modules and thus may not be considered a meta-interface.
  However, the operations layer (of the operational design, see section 3.3.3) contains all the modules that are responsible for MARIE's behavior, because these modules do the actuation, sensing and controlling of MARIE. The remaining layers 'just' enforce this control. By means of the property interface additional information is available to facilitate the high-level decision making processes.

- *The property interface provides unstructured knowledge about its components*
  The second point of criticism states that it is no good, that the interface only provides properties and no information about the relationship between these properties. This is true, the property interface is not aware of the meaning of the information that it passes. This would require a semantic description of (the properties of) the elementary operations and additional reasoning capabilities to interpret this semantic information. This has not been designed. The interface relies on the fact that the requester is aware of the meaning of the information it requests.
  This means that instead of deciding what elementary operation is desired in a particular situation, or what operation can solve a particular problem or task (which is a possibility with capability descriptions), a broad applicability was realized, still facilitating task tree construction, self-configuration or self-healing. Besides the introduced interface does not make future development of semantics impossible, it might even augment it.

- *The requester must be aware of the information it requests*
  The requester of the properties, i.e. some module of MARIE, must be aware of the meaning of that property. This means that in practice the user of the property interface 'just' needs some additional property, in order to facilitate some reasoning process.This restricts autonomy as a result from the interface.
  One can argue against this point that, the developer of an application already is aware of the relationship and meaning between properties, before he or she decides to use them. Hence, enforcing structure would not aid the developer. It does not facilitate automatic

reasoning, as is done in agent-based system, but this was argued against, in chapter 5, section3: alternative solutions.

- *Proliferation of properties*

The absence of structure also allows *any* property to be introduced by the developer to be communicated by the interface. In itself this freedom can be used as an advantage, because a developer may introduce any elementary operations aspect at will. However, this might create a proliferation of handled properties, creating some sort of over-fitting. This is no valid criticism, it is a strong feature of the property interface, and just means that the developers must be moderate in introducing new properties to the interface.

11 CONCLUSION

11.1 Introduction

This chapter concludes this thesis. The first section gives an overview of the discussed issue. Section 11.3 discussed what has been achieved and section 11.4 lists some recommended future changes. Section 11.5 generally concludes the thesis.

11.2 Overview

The main focus on the control issue with MARIE was on the amount of control information, that the human operator has to supply, and the lack of control knowledge that the system determines on its own.
What is introduced is a way to obtain the control information per elementary operation. In theory the interface supplies knowledge about elementary operations that could be used to enhance the systems functionality, which is expected to increase in autonomy.
The elementary operations can be considered as building blocks for missions. Together with the datamanager interface and the elementary operations interface this is a powerful software package that allows the cart to perform various missions. The flexibility of controlling MARIE, that results from this, is augmented by the property interface. The combination of the three interfaces and the functionality of the elementary operations provides means to built autonomous software that enhances control of the MARIE robot.
Since the interface has only been designed and not implemented, there has been no possibility to tests this hypothesis. Nevertheless, it is very likely that the introduction of the interface will improve the robot’s behavior.

11.3 Achievements

The extent in which the propositions of this paper, formulated in chapter 1.2, have been realized is discussed next. The propositions were:

- Decrease of human interaction with MARIE, and
- Increase of the robots autonomy

The two changes to the system, that should realize these propositions, are the introduction of location independence and the property interface.

- Decrease of human interaction with MARIE

That the introduction of location independence decreases human interaction with the system can be seen when one realizes that the operator no longer has to compile the software of MARIE when the configuration is changed, and also for the developer the interaction with the system is decreased, because testing and debugging new modules is much simpler now.
The human interaction also decreases because of the property interface, because some decisions that the operator has to take, can be made automatically when the property information is available. This may concern aspects such as setting up a proper configuration, or building and verifying a task tree.

- Increase of the robots autonomy

For the increase of MARIE’s autonomy much the same reasoning applies; location independence makes the system more flexible; modules are able to locate each other.
It is the *use* of property interface that increases the robots autonomy. The property interface provides self-knowledge that can be used for reasoning on a high-level. The interface will makes it possible for, to-be-developed, applications to take decisions, based on properties,

that will allow it to have MARIE cope with new situations. For example when different versions of a module are available, the module's identity number is no longer unique within the network. Requesting the modules location would result in all modules with that identity number that are currently registered. The property interface can be used to distinguish between the different instances of the module, based on their properties (version, up-time, parameters, etc.).

When the property interface is realized applications can be developed that extend MARIE with the following collection of more specific autonomous characteristics:

- Parameter variation
- Parameter optimization
- Predicting results
- Understanding undesired behavior
- System analysis
- Monitoring of an elementary operation's behavior

From the above the following conclusion is drawn:

11.4 Future work

Some room for improvement was observed and future enhancements are recommended with respect to the following:

General

- Implementing the interface and performing the experiments.
- Redesign of action dispatcher

Functional:

- Provide elementary operations with termination function. This gives the elementary operations a life-cycle. The existence of elementary operations can be determined more accurately.
- Fusion with Erik de Ruiter's database [3], with regard to identifying module versions

Technical:

- Move the feature datamanager to the VxWorks machine
- The way that the eo-interface is implemented suggests that it is meant to deal with al communication that goes to or from an elementary operation. A generic listener function should be implemented, for both property- and eo-interface.

11.5 Conclusion

This master thesis concerns with controlling the MARIE mobile autonomous robot. It is researched if the autonomy of MARIE will increase when it has more knowledge about its reasoning, sensing and controlling elements.
A property interface was designed that provides property information about the elementary operations to all other modules within the system. Describing the elementary operations in terms of properties makes it possible to express practically anything that can be said about an elementary operation and thus it enables the merging of general concepts such as version, location and control information with technical, situation dependent information such as up-time, control values, etc. In addition to an interface for communicating data and an interface for communicating commands, the systems is extended with an interface for communicating about its sensing, reasoning and controlling components.

MARIE's modules were made location independent, which makes setting up the system more dynamic.
The property interface, in combination with the location independence, gives high-level modules access to information about all the controlling elements, which makes it possible for these high-level modules, that are (in-)directly in charge of controlling MARIE's behavior, to use the properties of elementary operations to make decisions, that were formerly taken by the human operator (or with the knowledge supplied by the human operator). It is possible to select, monitor or verify the (functionality of) elementary operations. This creates a variety of extra functionality; self-configuration, task tree verification, parameter varying, self-recovery, all become possible for MARIE.
With respect to answering the hypothesis of this thesis: tests have shown that location independence decreases the amount of work the human operator has to spend on MARIE, because setting up configurations can now be done automatically (and as a result MARIE's autonomy has increased).
In theory, the combination of the property interface and location independence of the robot's modules also increases autonomy. The introduction of the control details to higher-levels provides them with the information to enhance decision making processes, that are typically done by the operator, such as self-configuration or self-healing. Such functionality will increase MARIE's autonomy.

REFERENCES

[1] G.A. den Boer: A control architecture for the MARIE autonomous mobile robot (1995)

[2] F. Terpstra: Een on-line planner voor MARIE (2001)

[3] E. de Ruiter: An experimentation platform for MARIE (2003)

[4] IBM corporation: Autonomic Computing, IBM's perspective on the state of information technology (IBM, 2001)

[5] B.C. Williams; Model-based autonomous systems in the new millennium. (1999)

[6] J.O. Kephart, D.M. Chess: The vision of autonomic computing, (IEEE magazine issue January 2003)

[7] N. Shadbolt: Beyond Brittleness, (IEEE intelligent systems, issue November/December 2002)

[8] M. Luck, P. McBurney, C.Preist: Agent Technology: Enabling next generation computing. A roadmap for agent-based computing. (AgentLink, January 2003)

[9] Sun Microsystems: Jini Architectural Overview (Sun Microsystems, 1999)

[10] K. Sycara, M. Klusch, S.Widoff, J. Lu: Dynamic Service Matchmaking Among agents in open information environments (2000)

[11] K. Sycara, J. Lu, M. Klusch, S. Widoff: Matchmaking among heterogeneous agents on the internet, (2000?)

[12] A. Cassandra, D. Chandrasekara, M. Nodine; Capability-based Agent Matchmaking (2000)

[13] P.C. Homburg, The architecture of a worldwide distributed system, (2001)

[14] O. Lassila, R. Swick: Resource Description Framework (RDF) Model and Syntax Specification. (W3 Consortium, 1998)

[15] N. Minar, M. Gray, O. Roup, R. Krikorian, P. Maes; Hive: Distributed Agents for Networking Things (MIT Media lab, 1999)

[16] Deitiel, Deitel; Java, how to program (Prentice Hall, third Edition 1999)

[17] RPC: rfc 1050 (http://www.faqs.org/rfcs/rfc1050.html)

[18] Object Management Group; The common object request broker: architecture and specification OMG, Needham, MA, 1999

[19] D. Obasanjo, S. Bhatia; An introduction to distributed object technology (www.25hoursaday.com)

[20] J. Geerts: Path planning for vehicle docking (199?)

[21] M. Mergel, N. Gouvianakis: Experiments with the MARIE vehicle; Autonomous docking in unknown environment, experiment, series D1-D4, (1992)

[22] F. Kilander, p. Werle, K. Hansson; JIMA: a Jini-based infrastructure for active documents and mobile agents (1999)

APPENDIX A: CONCEPTS FOR IMPLEMENTING THE PROPERTY INTERFACE

A.1 Introduction

This appendix will describes an onset towards the implementation of the property interface. The description focuses on two issues. One describes the properties of the elementary operations, in section A.4. In section A.5 the communication protocol is devised, and section A.6 gives a brief description on what server and client functions could look like. But first a preface, section A.2, to describe what context this chapter must be read in and second an overview is presented in section A.3.

A.2 Preface

Location independence has been implemented, this is done in such a way that it can be made part of the property interface. However, the communication of location information is different with the respect that communication proceeds actively, contrary to other properties that are only published on demand, and that obtaining the location information is done from the feature datamanager.
To tackle the communication of all the other properties some additional implementations are required that cannot directly be done in analogy with the way that was dealt with the location property. This requires some specification of the properties and the communication and publication mechanism.

A.3 Overview

In this chapter is looked ahead how the property interface for the other properties than the location property could be implemented. The aim is to propose how the further development of the property interface should proceed.

- More details are provided about the specific properties of the elementary operations. Most importantly details about the internal obtaining of properties is described.
- A protocol, to support communication, is proposed.
- Generic functions are presented

A.4 The Properties in Detail

This section provides additional details about the properties of tables 6-1 to 6-4.
A collection of details for each property must be identified, before they can be communicated. These aspects include high-level issues such as, how the properties are represented and technicalities, such as, how an elementary operation knows its own properties.

Internal Resources
An elementary operation supplies its own properties when requested to do so. But where can an operation obtain its own properties? An elementary operation receives a request for a certain property and then invokes some internal functionality to obtain that property. This can be done directly by the elementary operation, for example the up-time of an elementary operation can be obtained by using system-calls, others must be obtained differently. The table describes for each property where it is gotten from. This may be:

- from a file
- by system calls
- from the feature datamanager
- from eo-interface

- from the database of Eric de Ruiter

Internal Supplier of properties
Some properties of an elementary operation, such as the default value for a certain parameter, cannot be obtained automatically. Such properties must be thought up by either the developer or the operator, before the elementary operation can determine them.

Representation of properties
An additional issue that should be addressed is how the properties are represented. For some properties, such as its name or location, this is straightforward. But this is not the case for the parameters. These are not described by only providing the amount of parameters that control an elementary operation; properties such as their data type and order are also required.
This leads to a structural representation of the parameter properties.

A.4.1 The Properties Table

Note that an elementary operations contains a sheer unlimited amount of properties. Therefore a selection was made with the self-configuration example in mind.

Below the table with properties is given.

Property	*Description*	*Data Type*	*Supplier (operator/developer/system)*	*Local Resource*	*Example*
Name	Name of the elementary operation	Character string	D	File	"DS – Docking Spot Detector"
Location Information	Machine the elementary operation is running on	IP and port number	S	Feature datamanager	127.0.0.1:1201
Existence of operation	Existence of the elementary operation	Boolean	S	Feature datamanager	True or False
Version	Version Information	See notes below	D/O/S?	See notes below	See notes below
Amount of parameters	Amount of parameters	Integer	D	File	1,2,3,4,…
Parameter Names	The names of the parameters	Array of strings	D	File	"p1", "p2", "p3", …
Parameter type	The type of the parameters ('maps' on parameter names)	Sequence of predefined Characters	D	File	c,l,d,.. (c)har, (d)ouble, (l)ong
Default value	Default values for parameters	Sequence of alpha-numeric values	D	File	Sequence of any possible value
Minimum Value	Minimum value for a parameter	Alpha Numeric	D/O/system in the future?	File	A value
Maximum	Maximum value	Alpha	D/O/ system	File	A value

Property	*Description*	*Data Type*	*Supplier (operator/developer/system)*	*Local Resource*	*Example*
Value	for a parameter	Numeric	in the future?		
Previous Value	The previous value for a parameter	Alpha Numeric	S	Eo data	A value
Current Value	The current value for a parameter	Alpha Numeric	S	Eo data	A value
InUse	Indicates if a eo is already being *used* by MARIE	Boolean	S		

Table A-1: Implementation details of elementary operations properties

The fields have the following meaning:

1. Property Name
The name of the property

2. Short description
Description of the property

3. The data type of the property
The properties can have different data types, this field describes that data type. Simple types such as integers and characters are possible, but also more complex data types such as structures may typify a property.
When communicating these properties data types have to be taken into consideration, how this is done is discussed in section ??.

4 & 5. Supplier and local resource of the property information
There are two fields to represent where an elementary operation gets its property from. One specifies the actual source (file, system call, databse, etc) the other specifies who defined the data, if relevant.

6. Example
An example of the value for the property.

A.4.2 The parameter properties

To represent the parameter properties in one structure, it must be taken into account that the amount of parameters depends on the elementary operation and therefore the structure can not be of a fixed length. Since the interface (and thus the data representation) is demanded to be generic as well as non elementary operation specific, a dynamic data structure should be devised that can be used for representing all parameters of for each elementary operations.
Of each elementary operation, a parameter is uniquely defined by its name and data type, so each parameter can be represented by two fields. Ordering these fields for all parameters and adding a number that represents the amount of parameters of the elementary operation in one data structure gives a description of its set of parameters.
There is an issue not addressed here which is: embedded structures. When a parameter is not of a simple data type, such an integer or a character, but consists of a structure, then special measures have to be taken to deal with it. this is no big deal however, since that structure can be described in the same manner as the other data structure.

A similar construct can be devised for the (default) values of each parameter.

A.4.3 The version issue

Uniquely identifying an elementary operation is done by its id-number (or name). In theory multiple instances of the same elementary operation can be active within the network. In those cases the location information can be used to distinguish them. However, when these instances are of a different version, that version information is needed to uniquely identify the operation.
Problem is, that there is no version information available in the MARIE software. Erik de Ruiter has made a database that makes it possible to uniquely identify different versions, without the use of explicit version management [3]. This is realized by combining executables with there sources and compilation date.
This information can not be used by the property interface, however, because the sources and compilation date are unknown. So until this issue is not solved, Eriks database can not be used as source for versions.
I suggest, as temporary solution, that requires the developer to add a version number to each new (version) of elementary operation. This is, however, no proper version management and can therefore not be considered a long-term solution.

A.5 Communication

On the client-side functions will be developed that translate a request into a message (i.e. a request for some property). The message is sent to the supplier of the information on the other end of the line, i.e. the server. On that server side a function must be developed that receives and replies to the request. The next section describes the general steps the servers and clients must go through, to use the interface; it defines a protocol for the interface.

A.5.1 Messages

Information between processes is communicated via messages. Messages are identified by a message type, that is specified in the message, together with specific content. The message type facilitates following the rules of the protocol and defines consecutive actions. I.e. the message type tells the communicating parties what step to take next.
Each property has a message type associated with it. On receiving such a message the server will know that a request was send for a particular property. The server can then respond by examining the content of the message and sending the property (if found) or sending a property-not-found message.

A.5.2 Protocol

The exchange of information must obey certain rules. These rules must describe when to send what information, and must pertain to both client and server. Such rules are defined in a protocol.
The steps the client and server go through from question to answer are listed below. Note that it is required that there already exists a connection to the server.

Figure A-2 gives a graphical representation of the protocol.

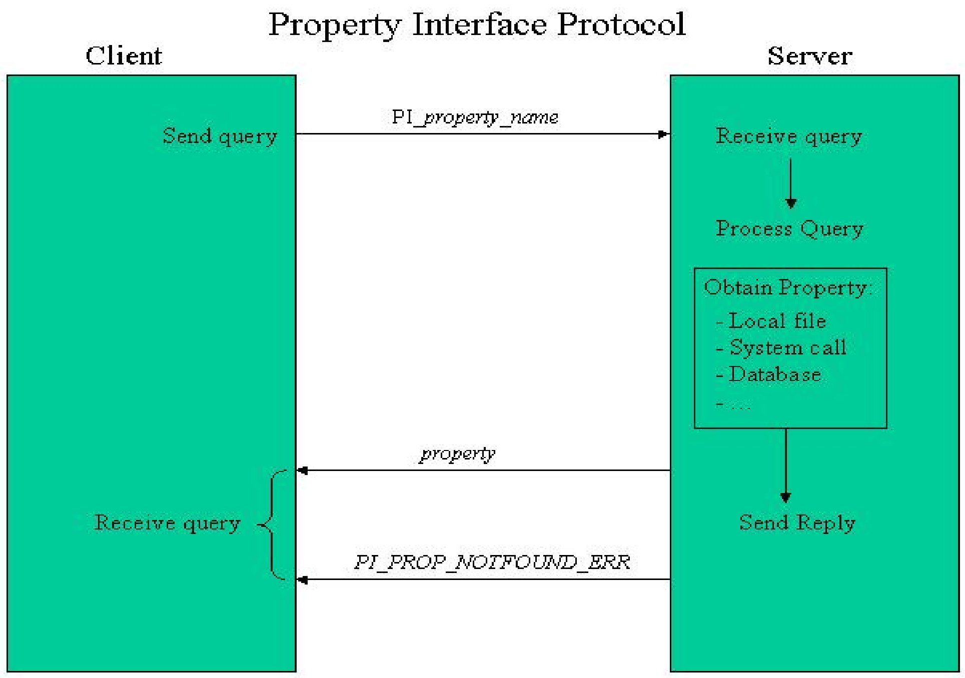


Figure A-2: Protocol of the Property Interface

A.6 Retrieving Properties

The interface must provide two functions for each property. One function for generating the appropriate message for the requesting module that needs property information of an elementary operation, and a function on elementary operation's side, that internally retrieves the property.

A.7 Conclusion

This chapter gives a preliminary description for the implementation of the property interface; it might be used as a guideline for the actual implementation, and merely presents some ideas. The structure of the properties of elementary operations and functions to obtain and communicate these properties are explained in some detail.
The requester of properties can be any module in the system. The process of requesting information is a matter of contacting the relevant elementary operation and starting communication, as defined in a protocol. It is required that the requesting process sets up and closes the connection. (This is not facilitated by the interface.)
The requester should be aware of the particular data type of the property it has requested. The elementary operation, whose property is requested, supplies the property itself. The way the elementary operations obtains these properties depends on that property and the elementary operation. The internal function used for this, can obtain the property from system resources, from a user-defined a text file, or it might be directly available. In the future a link to the database created by Eric de Ruiter may be developed.

APPENDIX B: SOURCE CODE

This appendix contains the source code of the functions that were added or altered for location independence. For clarities sake the following (already existing) structure definitions are added.
In the code these structures contain location records. Note that for they are not the same; the machine and name field are of different size, for the datamanagers and elementary operations.

```
typedef struct DM_IDENT
{
    long port;          /* the port of the dm service */
    long ident;         /* the unique id-number of the dm */
    char machine[64];   /* The name of the machine running
                         * the service (i.e. vw.science.uva.nl */
    char name[64];      /* The human readable name of dm
                         * (i.e. "world datamanager") */
} dm_ident;

typedef struct EO_IDENT
{
    long port;          /* the port of the eo service */
    long ident;         /* the unique id-number of the eo */
    char machine[32];   /* The name of the machine running
                         * the service (i.e. vw.science.uva.nl */
    char name[32];      /* The human readable name of dm
                         * (i.e. "Trajectory Controller") */
} eo_ident;
```

B.1 Location Independence functions for datamanager

B.1.1 Existing functions

Retrieving datamanager information and storing datamanager information is through slight adaptation of the existing datamanager functions, `dataman` and dm_connect.

```
int dataman(long ident, long port_no)
{

    dm_all D;
    unsigned long t_now;
    long      t_del;
    int          i;
    long    max_sleep;
    int         ret;       /*Locaction Indenpence */

    D.my_id = ident;
    D.fdmax = 127;
    D.n_cl = D.count = D.errval = 0;
    D.nr_close = -1;
    D.dm_list->p_seq = D.dm_list->n_seq = D.dm_list;
    D.dm_list->p_source = D.dm_list->n_source = D.dm_list;
    D.dm_list->p_class = D.dm_list->n_class = D.dm_list;
    D.cur_seq = 0;
    for (i = 0; i < 16; i++)
    {
```

```
    D.by_source[i].p_seq = D.by_source[i].n_seq = D.by_source + i;
    D.by_source[i].p_source =
                  D.by_source[i].n_source = D.by_source + i;
    D.by_source[i].p_class =
                  D.by_source[i].n_class = D.by_source + i;
    D.by_class[i].p_seq =
                  D.by_class[i].n_seq = D.by_class + i;
    D.by_class[i].p_source =
                  D.by_class[i].n_source = D.by_class + i;
    D.by_class[i].p_class = D.by_class[i].n_class = D.by_class + i;
  }
  for (i = 0; i < MAX_CLIENTS; i++)
  {
    D.clients[i].fd = 9999;
    D.clients[i].flags = 0;
    D.clients[i].proc_id = 0;
  }

  /* The Datamanager-space has been initialised now,
   * let's store it's existence
   * Store its location in the feature datamanager */
  if (ident != DM_FEAT_IDENT)
    {
    if((ret = dm_location2dm(ident, port_no)) < 0)
      {
        DPRINT(ERRS,
              ("dataman: Location not reported:
                           id: %d, port: %d, code: %d\n",
                (int) ident, (int) port_no, ret));
      }
    else
      {
        DPRINT(MESS, ("dataman: Location successfully reported:
                           id: %d, port: %d, code: %d\n",
                (int) ident, (int) port_no, ret));
      }
    }


  DPRINT(MESS, ("dm task: calling DtCreateSock\n\n"));

  D.in_fd = DtCreateSock(port_no, &(D.dm_sock), MAX_CLIENTS);

  DPRINT(MESS, ("dm task: returned from DtCreateSock\n\n"));

  D.fdmax = (D.in_fd > D.fdmax) ? D.in_fd : D.fdmax;

  D.nr_clients = 0;

  FD_ZERO(&D.readset);
  FD_ZERO(&D.writeset);
  FD_ZERO(&D.exceptset);

  FD_SET(D.in_fd, &D.readset);

  /* the feature datamanager should respond to keyboard input
   * this allows us to see what records are stored there */
  if (ident == 1201)  /* 1201 == feature datamanager */
```

```
{
  /*
   * stdin added to listen list, input on stdin should result in
   * a client list
   * Just for the Fe3ature Datamanager, though.
   */
  FD_SET(0, &D.readset);
}

D.repeat = 1;
D.nr_wait = 0;
D.first_wait = 0;

DPRINT(MESS, ("Now starting repeat loop\n"));

max_sleep = DM_MAX_SLEEP * sysClkRateGet();
DPRINT(MESS, ("max_sleep %ld\n", max_sleep));
while (D.repeat)
{
  D.readfrom = D.readset;
  D.writeto = D.writeset;
  D.excepton = D.exceptset;
  D.timeout.tv_sec = DM_MAX_SLEEP;
  D.timeout.tv_usec = 0;
  DPRINT(MESS, ("dataman: D.nr_wait %d\n", D.nr_wait));
  if (D.nr_wait > 0)
  {
      t_now = tickGet()+1;
      t_del = D.clients[D.first_wait].ret_time - t_now;
      DPRINT(MESS, ("dataman: t_del %ld\n", t_del));
      if (t_del < max_sleep)
      {
      double  d;

        if (t_del > 0) {
            d = (t_del * 1000000.0) / sysClkRateGet();
            D.timeout.tv_usec = irint(d);
        } else {
            D.timeout.tv_usec = 1;
        }
        D.timeout.tv_sec = D.timeout.tv_usec / 1000000;
        D.timeout.tv_usec = D.timeout.tv_usec % 1000000;
      }
  }
  DPRINT(MESS, ("dataman: (3) tv_sec %ld tv_usec %ld\n",
                                        D.timeout.tv_sec,
                                      D.timeout.tv_usec));
  D.count = select(D.fdmax + 1, &D.readfrom,
                                &D.writeto,
                                &D.excepton,
                                &D.timeout);
  if (D.count == 0)
  {
      /* We have run into a time-out.*/
      if (D.nr_close >= 0)
      {
        close(D.clients[D.nr_close].fd);
        D.clients[D.nr_close].fd = 9999;
```

```
          D.nr_close = -1;
        }
    } else if (D.count < 0)
    {
        D.errval = errnoGet();
        perror("dataman: Error in select ");
    } else
    {
        /* count > 0, i.e. regular I/O */

         if (D.nr_close >= 0)
         {
          close(D.clients[D.nr_close].fd);
          D.clients[D.nr_close].fd = 9999;
          D.nr_close = -1;
         }

         /*
          * Every input on stdin will result in a list of clients.
          */
         if (FD_ISSET(0, &D.readfrom) && (ident == 1201) )
           {
           char x[10];
           bzero(x, 10);
           read(0, &x, 10);
           DPRINT(ERRS, ("Current Client Information:\n"));
           for (i=0; i < D.nr_clients; i++)
             {
               printf("Client %d on fd %d with process id: ",
                    i, D.clients[i].fd);
               if (D.clients[i].proc_id==0)
                 printf("<no messages yet>\n");
               else
                 printf("%d == %x\n", D.clients[i].proc_id,
                                        D.clients[i].proc_id);
             }
           DPRINT(ERRS, ("Listed IDs\n"));
           list_id();
           }

        /* Now check for new connections */
        if (FD_ISSET(D.in_fd, &D.readfrom))
        {
          set_up_connection(&D, D.in_fd);
          D.count--;
        }
        /* Now, at last, read messages from clients */
        for (D.n_cl = 0; (D.n_cl < D.nr_clients) && (D.count > 0);
          D.n_cl++)
        {
          if ((D.clients[D.n_cl].fd <= D.fdmax) &&
              (FD_ISSET(D.clients[D.n_cl].fd, &D.readfrom)))
          {
              handle_client(&D, D.n_cl);
              D.count--;
          }
        }       /* End of loop over all clients */
        }           /* End of all I/O actions following select */
        /* Now test pending waits */
        if (D.nr_wait > 0)
        {
            t_now = tickGet() + 1;
            D.reply.sender = D.my_id;
            D.reply.type   = DM_TIMEOUT;
            D.reply.info   = 0;
            D.reply.t_send = tickGet();
            while (D.nr_wait &&
                         (D.clients[D.first_wait].ret_time <= t_now))
            {
              write(D.clients[D.first_wait].fd, (char *) &(D.reply),
                                               sizeof(D.reply));
              D.nr_wait--;
              D.first_wait = D.clients[D.first_wait].next_wait;
            }
        }
    }            /* End of infinite loop */

    return (0);
}

int
dm_connect(dm_ident *ident, dm_work **dm)
{
    dm_name  nm;
    int       s;
    dm_ident *bak;

    /* Location Indepence: to get backward compatability this is
     * The place where locations are verified and retreived
     */
    if (ident->ident != DM_FEAT_IDENT)
        bak = dm_getLocation(ident);
    else
        bak = ident;

    if (NULL == (*dm = (dm_work *) malloc(sizeof(dm_work))))
    {
      fprintf(stderr,
           "Task %d: cannot allocate working storage \n%s\n",
           taskIdSelf(), "for data manager interface routines");
      return(DM_NOSTORE);
    }
    do {
      /* ident->machine can either be an IP number or a
       * machine name (i.e. a string). Distinctions between those
       * two are hard to find. (IP:114.105.106.110 could
       * also be machine 'rijn')
       */

            s = DtConnect2Sock(ident->port,
                         (struct sockaddr_in *) &((*dm)->sockname),
                                                     ident->machine);
              if (s < 0) {
                /* Location Independence:
                 * found location failed, try old settings'*/
                s = DtConnect2Sock(bak->port,
                         (struct sockaddr_in *) &((*dm)->sockname),
                               bak->machine);
                /*-- Location Independence] */
                if (s < 0) {
                    if (s == ERROR)
                          return (ERROR);
                    if (s == -2)
                          return (DM_NOHOST);
                    if (s == -3)
                          taskDelay(4 * sysClkRateGet());
                } else
                  {
                    DPRINT(ERRS,
                  ("Connection established with old settings: %s:%d\n",
                                       bak->machine, (int) bak->port));
                    printf(
                        "Connection established with old settings.\n");
                    bcopy(bak, ident, sizeof(dm_ident));
                  }
              }
    } while (s <= 0);
    (*dm)->fd = s;
    (*dm)->dm_id = ident->ident;
    (*dm)->my_id = taskIdSelf();

/*
 * NEW MESSAGE TYPE FOR HEADER
 * A connection  has just been established; now send this process'
 * proc-id.
 */
    nm.reply.sender = (*dm)->my_id;       /* id-number */
    nm.reply.type   = DM_HEADER;          /* message type */
    nm.reply.t_send = (int) tickGet();    /* ticks */
    nm.reply.info   = 0;
    strncpy(nm.name, taskName(taskIdSelf()), 32);    /* taskName */
    nm.name[31] = '\0';
    write((*dm)->fd, (char *) &nm, sizeof(dm_name));

    return (DM_OK);
}
```

B.1.2 New functions

The datamanager function library has been extended with four functions to facilitate location independence. These functions are `dm_location2dm`, `dm_getLocation`, `dm_nr2id`, `dm_get_all_modules` and `dm_list_all_modules`.

The first function, `dm_location2dm`, is used to store a datamanger's location information at the feature datamanager.

```
/*
 * Stores location information in the Feature Datamanager
 *
 * A location is defined as the process' 'port_no',
 * its IP ('machine'), a string identifing it ('name') and its
 * ID number ('ident').
 * These are stored as dm_ident-structs, as a DM_CLASS_LOCATION
 * All records. Even the eo's are stored as dm_ident (and not
 * eo_ident), when an eo-record is retreived one should take this
 * into account!
 */
int dm_location2dm(long ident, long port_no)
{
  char         host_name[64];  /* Locaction Indenpence change      */
  dm_work      *p_dm_work;     /* idenitfies dm-connection         */
  dm_record    *dm_rec;        /* dm record containg info to store */
  dm_ident     *tl;            /* Actual information to add to dm  */
  char         *tmp;
  int          my_ip, i;
  int          ret=-1;

  if (ident != DM_FEAT_IDENT)
    {
      /* Construct dm-record's message header */
      dm_rec = (dm_record *) malloc(
                               sizeof(dm_record)+sizeof(dm_ident));

      dm_rec->header.tick_obs = tickGet();
      dm_rec->header.source = port_no;
      dm_rec->header.class  = DM_CLASS_LOCATION;
      dm_rec->header.size   = sizeof(dm_ident);

      /* Initialize the body of the message */
      tl = (dm_ident*) malloc( sizeof(dm_ident) );
      memset(tl, 0, sizeof(dm_ident));
      tl->port = port_no;            /* modules port number */
      tl->ident = ident;             /* modules id-number   */

       /* Get 'this' machines IP number */
#ifdef VXWORKS
      gethostname(host_name, 64);
      /* get localhost-name */
      DPRINT(MESS, ("gethostname: host_name is: %s ident: %d\n",
                                            host_name, ident));
      /* conver name to IP */
      my_ip = hostGetByName(host_name);  /* i.e.
      tmp = (char*) &my_ip;
      DPRINT(MESS, ("TEMP: hostGetByName-> my_ip: %d.%d.%d.%d\n",
           (unsigned int) tmp[0], (unsigned int) tmp[1],
           (unsigned int) tmp[2], (unsigned int) tmp[3]));

#else
      gethostname( host_name , 64);
      host2ip(&my_ip, host_name);
#endif
      memcpy(tl->machine, &my_ip, 4);           /* store IP address */

      /*
       * Fill in the name of the module (from its ID-number)
       * This name comes from a predefined list, defined in eo_inits.
       */
      i=0;
      while (module_names[i].ident > 0)
```

```
            if (module_names[i].ident == ident)
                  break;
            else
                  i++;

      if (module_names[i].ident)
        strcpy(tl->name, module_names[i].name);
      else
        strcpy(tl->name, "<unknown>");

      memcpy( dm_rec->data, tl, sizeof(dm_ident));

      /* connect top feature datamanager */
      ret = (int) dm_connect(&feat_ident, &p_dm_work);

      if (ret > -1)
      {
        /* using the existing dm_add-function, add the record
         * to the datamanager
         */
        ret = (int) dm_add( dm_rec, p_dm_work);

        if (ret < 0)
          {
            DPRINT(ERRS,
             ("dm_location2dm: Adding record FAILED with code: %d\n",
                                                                ret));
          }
        else
          {
            DPRINT(MESS,
              ("dm_location2dm: Adding record returned with: %d\n",
                                                                ret));
          }

        if (dm_disconnect(&p_dm_work)<0)
            printf("dm_location2dm: dm_disconnect error.\n");
      }
      else
      {
        DPRINT(ERRS,
        ("dm_location2dm: can't connect to FEATURE DATAMANAGER!\n"));
      }
    }

  return(ret);
}
```

The `dm_getLocation`, is used to retrieve a datamanger's location information. The actual retrieval is done by a function called `dm_nr2id`, which is explained later.

```
/*
 * Retreives location from Feature datamanager.
 * The requested module's location is identified by the
 * 'ident->ident' field.
 * On return the rest of this dm_ident structure is filled in.
 * The function also returns a pointer to the original
 * 'ident' struct, for recovery reasons.
 */
dm_ident *dm_getLocation(dm_ident* ident)
{
  long        target;  /* Identity number for DM to find */
  int         ret;
  int         ip1;
  char        ip2[4];
  dm_ident    *bak;     /* Backup of old record */
  ret = 0;

  /* Save the original record */
  bak = (dm_ident*) malloc(sizeof(dm_ident));
  bcopy(ident, bak, sizeof(dm_ident));

  /* Get the record */
  target = ident->ident;
  if (DM_OK  == dm_nr2id(target, ident))
    /* Record was found.
     * (ident-> machine could be an IP address and not a
     * machine name). Now check if the record has changed. It might
     * be possible that, because of old code, the dm_ident struct
     * contains incorrect location information. If so, the user
     * should be told and the record corrected.
     */
    {
#ifdef VXWORKS
      if ((hostGetByName(bak->machine) != (int) ident->machine) ||
         (ident->port != bak->port) )
#else
      host2ip( (int) ip2, bak->machine);

      if ( (ip2[0] != ident->machine[0]) ||
           (ip2[1] != ident->machine[1]) ||
           (ip2[2] != ident->machine[2]) ||
           (ip2[3] != ident->machine[3]) ||
           (ident->port != bak->port) )
#endif
      {
        DPRINT(WARN, ("dm_getLocation:
                                  Destination has been corrected!\n"));
        DPRINT(WARN, ("Old: %d.%d.%d.%d:%d\nNew: %d.%d.%d.%d:%d\n",
                                   bak->machine[0],  bak->machine[1],
                                   bak->machine[2],  bak->machine[3],
                                   bak->port,
                                   ident->machine[0], ident->machine[1],
                                   ident->machine[2], ident->machine[3],
                                   ident->port));
      }
    }
  else
    {
      DPRINT(ERRS,
        "dm_getLocation: Could not retreive location with id: %d.\n",
                                                              target));
    }

  return (bak);
```

```
}
```

The `dm_nr2id` –function is looks for a specific location record in the feature datamanager, based in a id-number. On success the function returns a completely filled ident-structure.

```
int dm_nr2id(long id, dm_ident *ident)
{
    int i, found, MAX_RECS=15;
    dm_work *p_dm;
    dm_loc_rec *records[MAX_RECS];
    i = 0;

    if ( dm_connect(&feat_ident, &p_dm) < 0 )
      {
      DPRINT(ERRS, ("dm_nr2id:
            can't connect to feature datamanager.\n"));
      }
    else
      {
      /* retrieve all records from feature dm */
      found = dm_select(0, 0.0, 0.0, 0, 0,
                        DM_CLASS_LOCATION, 10.0,
                        MAX_RECS,
                        (dm_record **) records, p_dm);
      if (found < 1)
        {
          DPRINT(ERRS,
             ("dm_nr2id: can't find id in feature DM: ret = %d.\n",
                                                     found));
          if (dm_disconnect(&p_dm) < 0)
            printf("dm_nr2ida: dm_disconnect error.\n");
          return(DM_WRONG_ID);
        }
      else if (found > 1)
        {
          DPRINT(MESS, ("dm_nr2id: found %d records.\n", found));
        }

      for (i = 0; i < found; i++)
        {
          if( records[i]->loc.ident == id )
            {
                  /* The id-number is found in the retrieved records
                   * Save that record, disconnect from feature
                   * datamanager, and return success
                   */
                  DPRINT(ROUT, ("dm_nr2id: found it.\n"));
                  /* copy records[i] to ident structure */
                  bcopy(&(records[i]->loc), ident, sizeof(dm_ident));

                  if (dm_disconnect(&p_dm) < 0)
                         printf("dm_nr2idb: dm_disconnect error.\n");

                  return(DM_OK);
            }
        }
      if (dm_disconnect(&p_dm)<0)
        printf("dm_nr2idc: dm_disconnect error.\n");
      }
    /* When executioon is here, no valid records was found:
     * Return failure
     */
    return(DM_WRONG_ID);
}
```

The `dm_get_all_modules` –function is used to retrieve all location records stored at the feature datamanager.

```
/*
 * Retrieves, at maximum MAX_RECS registred module locations
 */
int dm_get_all_modules(dm_loc_rec *records[], int MAX_RECS)
{
  int found;
  dm_work *p_dm;         /* connection to feature dm */

  if ( dm_connect(&feat_ident, &p_dm) < 0 )
    {
      DPRINT(ERRS, ("dm_get_all_modules:
                           can't connect to feature datamanager.\n"));
    }
  else
    {
      /* Get all records, using dm-interface function */
      found = dm_select(0, 0.0, 0.0, 0, 0,
                           DM_CLASS_LOCATION, 10.0,
                           MAX_RECS,
                           (dm_record **) records, p_dm);
      if (found < 1)
        {
          DPRINT(ERRS, ("dm_get_all_modules:
                  can't find any records in feature DM: ret = %d.\n",
                                                              found));
          if(dm_disconnect(&p_dm) < 0)
            printf("dm_get_all_modules: dm_disconnect error.\n");
          return (found);
        }
      else if (found > 1)
        {
          DPRINT(ROUT, ("dm_get_all_modules: found %d record(s).\n",
                                                              found));
          if(dm_disconnect(&p_dm) < 0)
            printf("dm_get_all_modules: dm_disconnect error.\n");
          return (found);
        }
      else
        {
          DPRINT(WARN, ("dm_get_all_modules:
              There are no modules registerd at Feature
                                                         Datamanager.\n"));
          if(dm_disconnect(&p_dm) < 0)
            printf("dm_get_all_modules: dm_disconnect error.\n");
          return (found);
        }
    }
  return(0);
}
```

The `dm_get_list_all_modules` –function is used to retrieve all location records stored at the feature datamanager and output hem to screen.

```
/*
 * Retrieves, at maximum MAX_RECS registred module locations
 */
void dm_list_all_modules()
{
    int i, found, MAX_RECS=100;
    dm_work *p_dm;
    dm_loc_rec *records[MAX_RECS];
    i = 0;

    if ( dm_connect(&feat_ident, &p_dm) < 0 )
      {
      DPRINT(ERRS,
         ("dm_list_all_modules: can't connect to feature
                                               datamanager.\n"));
      }
    else
      {
      found = dm_select(0, 0.0, 0.0, 0, 0,
                        DM_CLASS_LOCATION, 10.0,
                        MAX_RECS,
                        (dm_record **) records, p_dm);
      if (found < 1)
        {
        DPRINT(ERRS, ("dm_list_all_modules:
                can't find any records in feature DM: ret = %d.\n",
                                                         found));
        if(dm_disconnect(&p_dm)<0)
          printf("dm_list_all_modules: dm_disconnect error.\n");
        return;
        }
      else if (found > 1)
        {
        DPRINT(ERRS, ("dm_list_all_modules:
                           found %d records.\n", found));
        }
      for (i=0; i< found; i++)
        {
        printf("Record %d: ID: %d port: %d machine: ", i,
                                    records[i]->loc.ident,
                                    records[i]->loc.port);
        printf("%d.%d.%d.%d name: %s.\n",
                                    records[i]->loc.machine[0],
                                    records[i]->loc.machine[1],
                                    records[i]->loc.machine[2],
                                    records[i]->loc.machine[3],
                                    records[i]->loc.name);
        }
      if (dm_disconnect(&p_dm)<0)
        printf("dm_list_all_modules: dm_disconnect error.\n");
      }
}
```

B.2 Location Independence functions for elementary operations

B.2.1 Existing functions

Retrieving elementary operation information and storing elementary operation information is done through slight adaptation of the existing eo-functions , `eo_connect` and `eo_init` . Connecting to elementary operations is done using a function called `eo_connect`. One of its arguments is a structure with the id-number, machine and port of the service. This information is likely not correct, since it always was retrieved from the hardcoded location table. For location independence `eo_connect` has been adapted. The adaptation is such that the callee does not know it passed the wring location information. `eo_connect` it is called with the exact same arguments as before, but inside the function the correct location is retrieved. Below colored in red are the lines that were added.

```
int eo_connect(eo_ident *ident, eo_client_struct *eo)
{
/* The machine to on which the eo task runs is specified in machine.
 * This routine sets up the connection with the eo-server task.
 * It will also create a structure, pointed to by eo, containing all
 * the local information needed by the elementary operation interface
 * routines. The routine returns zero on success, and a negative
 * value on failure
 */

    int     s, count = 0;
    eo_con_rec *eo_rec = NULL;
    eo_name nm; /* [may'02] */

    /* backup for old information */
    eo_ident *bak;

    DEBUGON(WARN);
    *eo = NULL;
    if (NULL == (eo_rec = (eo_con_rec *) malloc(sizeof(eo_con_rec))))
    {
      logMsg ("eo_connect: cannot allocate working storage \n%s\n",
           "for eop interface routines");
      return(EO_NOSTORE);
    }

    /* save old record in 'bak' and store correct record in 'ident'
     */
    bak = eo_getLocation(ident);

    do {
      s = DtConnect2Sock(ident->port,
                              (SOCKADDR *) &(eo_rec->sockname),
                                                ident->machine);

      if (s < 0) {
        s = DtConnect2Sock(bak->port,
                              (SOCKADDR *) &(eo_rec->sockname),
                                              bak->machine);

        if (s < 0) {
          if (s == ERROR)
            return (ERROR);
          if (s == -2)
            return (EO_NOHOST);
          if (s == -3)
            {
            DPRINT (WARN,
                  ("eo_connect: Waiting for connection to '%s'\n",
                   ident->name));
            taskDelay(4 * sysClkRateGet());
            }
          else
            {
                logMsg("eo_connect:
                      Error no. %d in attempt to connect to '%s'\n",
                     s, ident->name);
                taskDelay(2 * sysClkRateGet());
            }
          count++;
          if (count > 3)
            {
            DPRINT (ERRS,
                  ("eo_connect: cannot connect to eo '%s'
                   (port %d@%s)\n",
                   ident->name, ident->port, ident->machine));
            return(EO_WEIRD_ERR);
            }
        }
        else
          /* Newly found location was not connectable */
          {
            DPRINT(ERRS,
               ("Connection established with old settings: %s:%d\n",
                      bak->machine, (int) bak->port));
            bcopy(bak, ident, sizeof(eo_ident));
            /* change dm record ? */
          }

      }
    } while (s <= 0);

    DPRINT(WARN, ("Eo_connect made a connection on fd = %d\n", s));
    eo_rec->fd = s;
    eo_rec->eo_id = 0;  /* WHY IS THIS ZERO AND NOT IDENT */
    eo_rec->my_id = taskIdSelf();
    *eo = eo_rec;

/*
 * NEW MESSAGE TYPE FOR HEADER
 * There has just been established a connection,
 * now send our proc-id.
 */

    nm.reply.sender = eo_rec->my_id;
    nm.reply.type   = EO_HEADER;
    nm.reply.t_send = (int) tickGet();
    nm.reply.size   = 0;
    strncpy(nm.name, taskName(taskIdSelf()), 32);
    nm.name[31] = '\0';
    write(eo_rec->fd, (char *) &nm, sizeof(eo_name));
    return (EO_OK);
}
```

Storing locations is initiated during initialization. Below is the code that an elementary operation calls when initializing.

```
int
eo_init(long ident, long port_no, eo_serve_struct *Dvoid)
{
    int             i, ret;
    eo_data *D;

    *Dvoid = D = (eo_data *) malloc(sizeof(eo_data));

    D->my_id =      ident;
    D->fdmax =      127;
    D->n_cl =       D->count = D->errval = 0;
    D->nr_close =         -1;
    D->rate =       sysClkRateGet();
    D->usec_tick =        1000000 / D->rate;
    D->state =      EO_JUST_INIT;
    D->params =     NULL;
    D->parsize =    0;
    D->boss_id =       -1;
    D->sentNotification = 0;

    /* start -Location Independence- */
    /* Store Location at feature dm */

    if ((ret = eo_location2dm(ident, port_no)) < 0)
      {
      DPRINT(WARN, ("eo_init: Location not reported: id: %d,
                                        port: %d, code: %d\n",
                             (int) ident, (int) port_no, ret));
      }
    else
      {
      DPRINT(MESS, ("eo_init: Location successfully reported: id:
                                          %d, port: %d, code: %d\n",
                                  (int) ident, (int) port_no, ret));
      }
    /* end   -Location Independence- */

    DPRINT(MESS, ("eo_init: calling DtCreateSock for port no. %d\n",
       port_no));

    D->in_fd = DtCreateSock(port_no, &(D->dm_sock), MAX_CLIENTS);
    if (D->in_fd < 0)
    {
      fprintf(stderr,
            "eo_init for task %x failed horribly\n", taskIdSelf());
      eo_show_id(D);
      return(D->in_fd);
    }

    DPRINT(MESS, ("eo_init: returned from creat_sock\n\n"));
    D->fdmax = (D->in_fd > D->fdmax) ? D->in_fd : D->fdmax;

    D->nr_clients = 0;
    for (i = 0; i < MAX_CLIENTS; i++)
    {
      CL(i).fd = 9999;
      CL(i).proc_id = 0;
      CL(i).flags   = 0;
    }

    FD_ZERO(&D->readset);
    FD_ZERO(&D->writeset);
    FD_ZERO(&D->exceptset);

    FD_SET(D->in_fd, &D->readset);


    D->repeat = 1;
    return(0);
}
```

B.2.2 New functions

The functions to store and retrieve location records for elementary operations are very similar to the functions for storing and retrieving datamanager location records. However, of old, there is an annoying difference. The records used, throughout the original code, for storing eo-records and dm-records are different. The field that stores the name (description) and machine of the module are, in case of the `eo_ident` struct 32 bytes, and in case of the `dm_ident` struct 64-bytes.
This is no problem when storing records, because the `eo_ident` struct is easily ‘upgraded’ to a `dm_ident` struct. But the reverse, i.e. obtaining a record and downgrading it to a `eo_ident` struct is harder, and therefore there have to be separate functions for the `dm_getLocation` and `eo_getLocation`.

```
/*
 * Just call dm_location2dm
 */
int eo_location2dm(long ident, long port_no)
{
  return (dm_location2dm(ident, port_no));
}
```

This function is (as said) similar to dm_getlocation, but merely differs when memory space is allocated for the retrieved record.

```
/* eo_getLocation returns the first record with ident->ident
 * from feature datamanager the old ident-struct is also returned
 */
eo_ident *eo_getLocation(eo_ident* ident)
{
  long   target;
  int    ret;
  char   ip2[4];
  eo_ident  *bak;
  ret = 0;
  /* Save old ident-struct */
  bak = (eo_ident*) malloc(sizeof(eo_ident));
  bcopy(ident, bak, sizeof(eo_ident));

  /* Set target for search */
  target = ident->ident;
  if (DM_OK  == eo_nr2id(target, ident))
    {
      /* Found a record, see if it is changed */
#ifdef VXWORKS
      if ( (hostGetByName(bak->machine) != (int) ident->machine) ||
#else
      host2ip( (int) ip2, bak->machine);

      if ( (ip2[0] != ident->machine[0]) ||
           (ip2[1] != ident->machine[1]) ||
           (ip2[2] != ident->machine[2]) ||
           (ip2[3] != ident->machine[3]) ||
#endif
           (ident->port != bak->port) )
        {
          DPRINT(WARN, ("eo_getLocation:
                         Destination has been corrected!\n
                         Requested: %s:%d\n
                         Corrected: %s:%d\n",
                         bak->machine, (int) bak->port,
                         ident->machine, (int) ident->port));
        }
    }
  else
    {
      DPRINT(ERRS, ("eo_getLocation:
                          Could not retreive location.\n"));
    }
  return (bak);
}

/* Given an id-number, eo_nr2id looks for the location struct
 * stored in the feature DM with that number
 */
int eo_nr2id(long id, eo_ident *ident)
{
    int i, found, MAX_RECS=15;
    dm_work *p_dm;
    /* eo's are stored in dm-fashion (64 byte fields): dm_loc_rec */
    dm_loc_rec *records[MAX_RECS];
    i = 0;

    if ( dm_connect(&feat_ident, &p_dm) < 0 )
      {
        DPRINT(ERRS, ("eo_nr2id:
                        can't connect to feature datamanager.\n"));
      }
    else
      {
      for (i=0; i < MAX_RECS; i++)
```

```
          records[i] = (dm_loc_rec *) malloc( sizeof(dm_loc_rec));

        /* only one record should be stored */
        found = dm_select(0, 0.0, 0.0, 0, 0,
                         DM_CLASS_LOCATION, 10.0,
                         MAX_RECS,
                         (dm_record **) records, p_dm);
        if (found < 1)
          {
            DPRINT(ERRS,
               ("eo_nr2id: can't find id in feature DM: ret = %d.\n",
                                                          found));
            if(dm_disconnect(&p_dm) < 0)
                   printf("eo_nr2id: disconnecting error!\n");
            return(DM_WRONG_ID);
          }
        else if (found > 1)
          {
            DPRINT(MESS, ("eo_nr2id: found %d records", found));
          }

        for (i = 0; i < found; i++)
          {
            if( records[i]->loc.ident == id )
              {
                DPRINT(ROUT,
                   ("eo_nr2id: found the record with id %d.\n", id));

                /* retreived records contain 64 byte fields,
                 * hence the 32 of eo_ident
                 */
                ident->port  = records[i]->loc.port;
                ident->ident = records[i]->loc.ident;
                strncpy(ident->name, records[i]->loc.name, 32);
                strncpy(ident->machine, records[i]->loc.machine, 32);
                if(dm_disconnect(&p_dm)<0)
                         printf("eo_nr2id: disconnecting error!\n");
                 for(i=0; i < MAX_RECS; i++)
                         free(records[i]);
                return(DM_OK);
              }
          }

      if(dm_disconnect(&p_dm)<0)
            printf("eo_nr2id: disconnecting error!\n");

      for(i=0; i < MAX_RECS; i++)
        free(records[i]);

      }
    return(DM_WRONG_ID);
}
```

B.3 Action Dispatcher

The Action Dispatchers `ad_init` –function looks up all elementary operations and datamanagers in the feature datamanager and connects to them Originally, no looking up was

done, but a predefined list (maintained by the developer) was used. Below is the changed code of the `ad_init` –function.
The retrieved records are filtered in two groups: eo’s and dm’s. The eo’s are stored in `eo_id` and the dm’s in `dm_table`.

```
/*
 * Action dispatcher initialize routine
 * EO's and DM's locations are dynamic, not predefined.
 * This results in various changes.
 */
int ad_init(ad_client, server)
    int     *ad_client;
    SOCK_INET *server;
{
    int    ret        = OK, rv = ERROR;
    int    eo_ret     = EO_OK;
    int    MAX_RECS   = MAX_NUMBER_EOS;
    dm_loc_rec         *records[MAX_RECS];
    int    i,j,k,l,m, eosFnd;

    DPRINT(ROUT,
         ("%s: Init action_dispatcher started \n", "ad_init"))

/********** Start of addition for Location independence **********/
    /*
     *
     * Fill table with Elem Operations
     * Previously eo_id-array was predefined.
     * Now it is filled with EO's registered at the Feature Dm
     */
    i=0; j=0; k=0; m=0;
    /* feature dm is predefined, and should always be present */
    dm_table[0] = feat_ident;
    l=1;                         /* 'l' is at least 1 */
    if (eosFnd = dm_get_all_modules(records, MAX_RECS))
      {
      /* For each found records check
       * if it is dm or eo
       * and then check if it is already present
       */
      for (i=0; i < eosFnd; i++)
        {
          if ( (records[i]->loc.ident > dm_port_start) &&
               (records[i]->loc.ident < (dm_port_start+100)))
            {
            /* The record is a location of a DM
             * Check if it is unique
             */
            for(m=0; m < l; m++)
              if (dm_table[m].ident == records[i]->loc.ident)
                {
                   bcopy(&(records[i]->loc),
                            &(dm_table[m]), sizeof(dm_ident));
                   break;
                }
            if (m==l)       // I.e. it's unique, so append to end
              bcopy( &(records[i]->loc), &(dm_table[l++]),
                                                sizeof(dm_ident));
            else
              DPRINT(WARN,
                ("ad_init: dm \'%s\', ident: %d, was seen twice.\n",
                       dm_table[m].name, (int) dm_table[m].ident));

            }
          else if ( (records[i]->loc.ident > eo_port_start) &&
                    (records[i]->loc.ident < (eo_port_start+100)) )
            {
            /* The record is a location of an eo
             * eo-locationrecords might be registered twice,
             * The latest version is best. Hence:
             */
            for (k=0; k < j; k++)
              if (eo_id[k].ident == records[i]->loc.ident)
                {
                  eo_id[j].port = records[i]->loc.port;
                  eo_id[j].ident= records[i]->loc.ident;
                  strncpy(eo_id[j].machine,
                                      records[i]->loc.machine, 32);
                  strncpy(eo_id[j].name, records[i]->loc.name, 32);
                  break;
                }
            if (k==j)      // I.e. it's unique, so append to end
              {
                eo_id[j].port = records[i]->loc.port;
                eo_id[j].ident= records[i]->loc.ident;
                strncpy(eo_id[j].machine,
                                    records[i]->loc.machine, 32);
                strncpy(eo_id[j].name, records[i]->loc.name, 32);
                j++;
              }
            else
              DPRINT(WARN, ("ad_init: eo \'%s\', ident: %d,
                  was seen twice.\n",
                  eo_id[k].name, (int) eo_id[k].ident));
            }
          else
            {
            DPRINT(WARN,
               ("Found a record which is not a DM or EO.\n"));
            }
        }
      /* Terminate arrays */
      dm_table[l].ident = -1;
      dm_table[l].port  = -1;
      strcpy(dm_table[l].machine, "localhost");
      strcpy(dm_table[l].name, "End-of-list");
      NEW_NUMBER_DMS = l;
      eo_id[j].ident = -1;
      eo_id[j].port  = -1;
      strcpy(eo_id[j].machine, "localhost");
      strcpy(eo_id[j].name, "End-of-list");
      NEW_NUMBER_EOS = j;

#ifdef MARIE_DEBUG
        /* print results */
        printf("ad_init: Found the following records:\n");
        for (i=0; i < j; i++)
        {
          printf("EO found %d: ID: %d port: %d machine: ",
                i, (int) eo_id[i].ident, (int) eo_id[i].port);
          printf("%d.%d.%d.%d name: %s.\n",
                         eo_id[i].machine[0], eo_id[i].machine[1],
                         eo_id[i].machine[2], eo_id[i].machine[3],
                         eo_id[i].name);
        }
        for (i=0; i < l; i++)
        {
          printf("DM found %d: ID: %d port: %d machine: ",
              i, (int) dm_table[i].ident, (int) dm_table[i].port);
          printf("%d.%d.%d.%d name: %s.\n",
              (char) dm_table[i].machine[0], dm_table[i].machine[1],
                     dm_table[i].machine[2], dm_table[i].machine[3],
                     dm_table[i].name);
        }
#endif  // MARIE_DEBUG

      }
    else
      {
         DPRINT(ERRS, ("ad_init: Fatal error,
              no location records found in Feature Datamanager."));
         return(ERROR);
      }


    for (i = 0; i <= NEW_NUMBER_EOS; i++)
    {

      bzero((char *) &(Elem_obj[i].eo_rec),
                                         sizeof(eo_client_struct));
      Elem_obj[i].id = eo_id[i];
      DPRINT(MESS, ("%s: Elem_obj[%d].id.ident %ld\n",
                    "ad_init", i, Elem_obj[i].id.ident));

      if (eo_id[i].ident < 0) break;
    }

/*********** End of addition for Location independence ********/

    /*
     * Connect to data-managers
     */
    i = 0;
    while (dm_table[i].port > 0)
    {
      data_links[i].id = dm_table[i];
      rv = dm_connect(&(data_links[i].id), &(data_links[i].work));

      if (rv != DM_OK)
      {
        ret = ERROR;
        printf("ad_init: Failed to connect to %s\n",
                                                dm_table[i].name);
        fflush(stdout);
        taskDelay(10);
      }
      else
      {
        printf("ad_init: connected to %s\n",dm_table[i].name);
      }
      i++;
    }

    /* initialise file-descriptor client */
    *ad_client = -1;

    /* create server_side of action-dispatcher socket */
    DPRINT(MESS, ("%s: Creating server-side socket\n", "ad_init"))

    if ((ad_sock =
        DtCreateSock(AD_PORT_NMB, server, MAX_CONN_PEND)) < 0)
      {
         printf("ad_init: Cannot create socket!! Exiting ad_init\n");
         ret = ERROR;
      }

    if (ret >= 0)
    {
      fd_last = ad_sock + 1;

      /* Connect to all EO's */
      /* 'defined NUMBER_EOS' number eos
       * changed to 'int NEW_NUMBER_EOS'
       */
      for (i = 0; i < NEW_NUMBER_EOS; i++)
      {
          if (eo_id[i].port < 0)
          {
            break;
          }
          printf("ad_init: Attempting to connect to '%s'\n",
                  Elem_obj[i].id.name);
          fflush(stdout);
          taskDelay(45);

          eo_ret = eo_connect(&eo_id[i], &(Elem_obj[i].eo_rec));

          if (eo_ret != EO_OK)
          {
            DPRINT(ERRS, ("%s: Error %d connecting to port %d\n",
                          "ad_init", (int) eo_ret,
                          (int) Elem_obj[i].id.ident))
                  ret = ERROR;
          } else
          {
            DPRINT(MESS, ("\t%s: connected on port %u\n", "ad_init",
                         (unsigned int) Elem_obj[i].id.port));
          }

      }
    }
    return (ret);
}
```

If eo’s (or some sort of proxies of each eo) could reason about the service they need, and would have sources to obtain these services from, this would steal control from the ad. This is much desired, because the drawbacks of centralized control, which are (…).
Original idea was probably:
Theory: agent based approach with out matchmakers. Meaning; the reasoning (matching request and reply, client and service) must be done y the requester itself.
Practice: groups of eo’s are there to accomplish certain tasks. These tasks are predefined. These are the reasoning, controlling and sening elements. Each of these tasks has an reopresentative. (sort of proxy) They determine what they want, and formulate this into a request. This request is resolve by themselves, using some simple semantics, (probably rule based like). By contacing all other tasks they negotiate which task to use next. All this within an action.

This has considerable consequences, since it means that the functionality of the four autonomic elements can not be taken over in an other module (say, the action dispatcher) because it would require internal information of each specific elementary operation.

Many distributed systems are being built today to address problems of embedded network applications. Sun's Jini system [2] is a leading architecture in this domain. A comparison between Hive and Jini is useful for highlighting the details of Hive's ecology of distributed agents. In some cases we are trying to make Hive more like Jini. In other cases, we believe Hive has advantages.

Both Hive and Jini are distributed application infrastructures, both are based on Java, and both rely on RMI distributed objects and mobile code. Both systems have discovery and lookup implementations. Hive and Jini both make extensive use of events for communication; indeed, Hive uses the Jini distributed event specification. And both systems represent devices and capabilities on the network, proxying if necessary.

Jini services are roughly analogous to the combination of Hive's shadows and agents. But Jini does not have anything like the conceptual split between the two. The distinction between shadows and agents gives Hive a useful abstraction between local, trusted code and networked, untrusted code. The autonomy of Hive agents gives a clear place to place computational activity in the system.

Another important difference is Hive's location-dependent model. In Hive, an agent's cell is an important fact; it tells you where the agent is on the network, (potentially) where it is physically, what resources it has access to, etc. Jini focuses mostly on services; the actual place a service is hosted on is not a major part of the Jini model. We believe that the location dependence of Hive contributes to scalability, both technically and conceptually.

Both Jini and Hive rely on mobile code for flexibility, and the argument for the usefulness of mobility is the same. A difference is that Jini only has single hop mobility: a service can upload a smart proxy to the user as an interface, but that proxy does not then migrate around the network. Currently, we do not make much use of Hive's multi-hop mobile agents, but we believe that it will become more important as the system grows.

For description, both Jini and Hive use the Java type system for a syntactic ontology. But where Hive uses RDF for semantic descriptions of agent capabilities, Jini uses more Java types. The Jini Lookup Attribute system is more closely like our use of RDF, but we believe the RDF model is more flexible due to the ability to perform deeply structured queries. Jini's Lookup Attribute system does not support queries on subattributes of attributes.

Hive does not currently support Jini's leasing or transactions, but probably should. Transactions will be useful to allow agents to enter into multi-message communications with a guarantee of consistency. Leasing will be a useful hook for allowing Hive agents to explicitly negotiate their relationships; an agent can express the decision to work with another agent for a limited period of time as a lease.

As a practical matter, Hive and Jini could be integrated by encapsulating a Jini service as a Hive shadow or making a Hive agent present itself as a Jini service.

Finally, Hive has so far stayed inside the Media Lab network. But in every decision we have designed Hive to expand beyond that, to work across the Internet. The abstractions inherit in the ecology of distributed agents gives us a conceptual model for organizing a worldwide network of interacting processes.

More specific goals are:

- Providing information more
- behavior is expected to be more autonomously and better enforced
- self-knowledge increases autonomy

[
What to do with these thoughts about different approaches:
Semantics could be used in order to facilitate brokering of some sort. (downside is that one needs a good, generic description of what one wants)
Control could be 'stolen' from the action dispatcher by having a task (or action?) take decision on its own. The task or action would then have to know what must be done (in and out status?) and on its own determine how to accomplish this. Can also be extended with brokering or matchmaking.

Does the discussion contain the assumption that reason (about properties) must be done by the requester, and not by a third party.
]

"De meerwaarde van semantische beschrijving is beperkt, omdat er door het geringe en redelijk constante aantal eo's genoeg overzicht is over wat een eo kan; redeneren in deze context (bepalen wie en hoe welke eo's een taak oplossen) kan makkelijk gedaan worden door middel van de gebruiker, itt tot automatisch, complex reasoning en infernce rules (en daarbij komende semantishe representatives.)"

TITLE: Improving control of the MARIE robot
Extending … with self-knowledge
- Voorkant
- appendix

- link to appendix of implementation

(done) - figure xx-x
(done) - page 63, discussion reference…
(done) - Experiment: plaatje
(done) - literature (evt. chaims: ww-db.stanford.edu/CHAIMS/)
(done) - Experiments log files
(done) - Discussion
(done) - Conclusion
(done) - Eo-specific or property-specific or not specific…(in design)
(done) - introductions: section and chapter titles…
(done) - bullet an numbering consequent

Denken:
literature and references to other sections and chapters.
What are HORN clauses again
Java type

Rol van architecture in oplossing en probleem en literatuur
Data verkeer aangaande elementaire operaties, de extra waarde van de interface

De alternative solution: no middleagent
Abstract…
More autonomy for MAIRE is realized by making the system modules location independent and extending all modules with the ability to obtain the properties (among which the specific control information) of the elementary operations.

Realizing more autonomy for MARIE by extending it with self-knowledge, is hindered by the location dependence of its modules and the way that is dealt with control information:

Preface…
Eindelijk is mijn scriptie af. Ik ben dank verschuldigd aan mijn vaders voor de steun, tijd, geduld, kritieken en correcties die hij gaf en leverde tijdens het schrijven van deze scriptie. Dank aan Arnoud Visser, voor de begeleiding, en voor het klaar staan als ik weer een vraag had. Ook dank ik Erik de Ruiter voor de gesprekken en discussies over MARIE in de rookpauzes.
Verder ben ik dank verschuldigd aan Frank Terpstra en Bram Heerebout voor het lezen en vinden van onjuistheden en andere fouten in deze tekst.