ACTROB: An Active Robotic Knowledge Base

Ralph Meyfarth
Technische Universität München, Institut für Informatik 6
P.O.Box 20 24 20, 8000 Munich 2, FRG
E-Mail: meyfarth@lan.informatik.tu-muenchen.dbp.de

Abstract

In this paper the ideas of an active knowledge base for use in Computer Integrated Manufacturing (CIM) environments will be presented. The aim of the knowledge base was to support the integration process of all components in a manufacturing system with autonomous mobile robots. Therefore, it is necessary to store and manage all data adequately, which are shared between processes in the manufacturing task. This means first, to represent a world model of the factory, and second, to hold the state of the production tasks. Because the knowledge base should support the integration of all systems involved, it must be not only a common information pool. It is also necessary that the knowledge base actively distributes information to the processes in the manufacturing task. So processes need not permanently poll the common knowledge base. Instead, they will be automatically informed about changes of states through the knowledge base itself, as soon as possible. The main concepts of the knowledge base itself as well as its use in manufacturing tasks will be shown.

Key Words: Active knowledge base, CIM, demons, knowledge representation

1 Introduction

The key of success in Computer-Integrated Manufacturing (CIM) systems will be found in the term Integration. This means CIM systems must solve the problem of integration of single subsystems (for example CAD systems, planning tasks, sensor and control units of autonomous systems, etc.) involved in the manufacturing process working together. Suitable exchange of data between processes, using common communication routines as well as common data, are necessary.

Therefore, the heart of CIM is an adequate data management technology [17]. Standard database systems, based on the classical approaches (relational, network or hierarchical), are not well suited [6] for CIM applications because of their lack of supporting complex structured data used in real world. Furthermore, in these database systems dynamic changes of schemes, for adapting of changes in the real world, are not possible. Another important aspect, not supported by classical database systems, is the integration of procedural knowledge. Instead of standard database systems, more intelligent and sophisticated database systems are required.

In the last years big efforts were done in the field of knowledge representation. Different concepts of data modelling like rules, frames, logic, objects or many hybrid concepts [11], [2] were proposed. But most systems, based on these concepts, are single user systems suited only for specialized expert systems but not for complex CIM applications. Mostly, they are realized as main memory systems and do not support the permanent storage of data as in classical database systems. Therefore, the approach realized in ACTROB was to combine the possibilities of database systems with the possibilities of new knowledge representation concepts as proposed in [14].

ACTROB was developed at the Institut für Informatik of the Technische Universität München as a real-time knowledge base, managing all the information needed to run a factory in an autonomous way. It was part of the research project SFB 331 Information Processing Concepts in Autonomous Mobile Robot Systems [1], supported by the Deutsche Forschungsgemeinschaft (DFG).

ACTROB shall support all components in a CIM environment in an adequate way. This includes that data coming from various sensor systems, as well as data coming from production planning processes, must be stored suitably. Moreover, not all data should be stored explicitly. Instead to reduce redundancy and inconsistency, some data are computed only at time when needed. Furthermore, to get more power in the information process, the knowledge base should be integrated into a communication environment so that all processes in the system are able to access the data stored in the knowledge base. It is also desirable for the knowledge base to become active. That means, the knowledge base is no longer a passive shared memory. Instead, it is able to distribute knowledge automatically to interested processes in the system.

The modelling approach of ACTROB is based on the ideas of object-orientation [9] and frames [15]. The object-oriented approach has seen to be very useful to create a world model dealing with complex structured data in a natural way [6]. The ideas of self-describing data and also of demons (attached procedures), taken from the frame concept, were integrated. The possibility of integration of procedural knowledge specially combined with the self-describing data facility has been seen as a very helpful and powerful concept. Providing a connection to a communication system the knowledge base is accessible from many users in...
parallel.

This paper is organized as follows: First, we give a short overview of the basic concepts and the architecture of ACTROB. Then, we introduce the active component built on demons and the possibility of enrolling as interested process. In the last part, it is shown how the knowledge base can be used as an information tool for autonomous robots and machines in a production environment.

2 Basic Knowledge Base Concepts

2.1 Object Structures

The basic structure in the knowledge base is the object which can be a prototype object (called class) or an individual object (called instance). A class describes the properties and the environment of its corresponding instances. Normally, similar real world objects are modelled by a class. The instances are incarnations of classes and contain the specific data of a real world object. Every property of a class is described in an attribute, where all the information about this property is stored. It is structured into facets as in frame-based systems. Here, you can find more specific knowledge about that attribute like the type, the default value, the allowed range of the attribute value, etc.

In figure 1 the facet list of attribute m-speed is shown. This attribute, for example, has the facet type which defines that an allowed value must be an integer in the range between 0 and 80. Also a default facet is defined which contains the value 50. This value is used for lack of an individual value of an object. The facet unit in this context defines the unit of measurement used for that attribute. Moreover, m-speed has the facets if-needed-before, if-needed-after. These facets contain names of demons (attached procedures; see 2.3) which are activated by read access to that attribute. The suffixes before/after define when the demons will be activated. The if-needed-before demon is started before the read access to the attribute while the if-needed-after demon is started afterwards.

Furthermore, there will be a distinction between system-defined standard facets and user-defined facets. The facet unit in figure 1 is a user-defined facet. That means, the knowledge base itself does not know the semantics of that facet but the user, who has defined it, knows it. Though only necessary facets will be created, there exists a set of standard facets, whose semantics are well known by the knowledge base. In detail we distinguish between the following standard facets:

value: contains the attribute value

type: describes the type of the attribute value (see 2.4)

default: contains a default value which is used in lack of an individual attribute value

if-needed-before, if-needed-after, if-added-before, if-added-after: contain demons which are executed by access to an attribute (see 2.3 for details)

receiver: contains a list of processes that want to be informed on changes to that attribute; used for active distribution of messages (see 4.1)

constraint: contains the constraints that must be checked on this attribute if the knowledge base should be in a consistent state (see 2.5)

The whole knowledge base, including the facet list, is dynamically changeable. It is possible to define new or to delete existing facets of an attribute.

Figure 1 shows also that attributes are distinguished in administration, class and instance attributes. Only class and instance attributes are assigned to a real object. Class attributes define attributes whose values are equal for all instances of a class. In contrast, instance attributes define properties whose values can be different for all instances of a class. Because of that, the value facet is stored in every instance (see fig. 1).

2.2 Inheritance

Special attributes are the administration attributes which are for internal use only. They define relations on classes and instances and are used for the inheritance mechanism. Classes can inherit properties from other classes. Inheritance allows to reuse definitions once created. ACTROB supports single inheritance as well as multiple inheritance. Therefore, objects can be created as a specialization or as an aggregation of other objects.

In figure 1 the class MOBILE.ROBOT inherits all attributes in its superclasses MANIPULATOR and VEHICLE. Therefore, the instance MB.3 of the class MOBILE.ROBOT has
values for all attributes, its own and those inherited.

2.3 Demons

A demon in ACTROB is a function, which satisfies the following conditions with respect to its declaration and calling: (a) A reference to the demon is put into a facet of an attribute in a class-object. Therefore, demons are attribute-specific. (b) Demons are always started within the use of the services put-attr-value and get-attr-value in the knowledge base interface [4]. They can be the first or the last action in this knowledge base functions only (see below). (c) Demons are allowed to access the whole knowledge base for manipulating data.

Read-demons (also called if-needed-demons) are connected with the knowledge base service get-attr-value only; write-demons (also called if-added-demons) are connected with the knowledge base service put-attr-value only. Before-demons are always executed as the first action in the knowledge base services put-attr-value and get-attr-value. After-demons are always executed as the last action of the knowledge base services put-attr-value and get-attr-value.

Demons are attribute specific, because they are defined in the facets list of an attribute. However, to support the general use of demons, it is necessary to give parameters to them. In particular, we suggest the same parameters as the knowledge base services get-attr-value and put-attr-value have. In ACTROB therefore, a demon has the parameters class, attribute, instance and the if-added-demon additionally the parameter value [5]. If a demon needs more information than it obtains through its parameters, it can fetch the necessary information from the knowledge base by itself.

Demons are activated as the first or last action in the two knowledge base services put-attr-value and get-attr-value. Conceptually, these functions are defined as follows:

- The knowledge base service put-attr-value (class, attr, inst, value) writes value into the knowledge base. It first checks, if there is an if-added-before demon in the attribute definition, specified by class and attr, and then reads the name of the demon. After that, the demon is executed. Yielding ok as a result of the demon, the value (possibly modified by the demon), is put into the knowledge base. As last action, the if-added-after demon is executed, if there is one.
- The knowledge base service get-attr-value (class, attr, inst) reads attribute values out of the knowledge base. Like put-attr-value, it first checks the existence of an if-needed-before demon and, if existent, executes the demon. After that, the attribute-value from the knowledge base, specified by class, attr and inst, is read and, if present, the if-needed-after demon is executed. The read value may be modified or even evaluated by the if-needed-after demon.

2.4 Types and Type Checking

In classical database systems no detailed type concept exists. All you have are basic types like integer, real and string. But in programming languages more type constructs are available. This leads to the so-called impedance mismatch problem when programs access data in the database. For coupling an application with a database system, an awful communication between the program language and the query language is necessary.

Therefore, it is desirable, (1) to have algorithms and data together; (2) to have a more dedicated type concept, when coupling the knowledge base with programs of the production process. The applications which access the knowledge base will be more safe, if all values are checked to be of correct type. To describe the type of a knowledge base value, a type-facet for each attribute was defined. This facet contains the information about the attribute type, which can be:

- basic types: bool, int, real, char, string, symbol
- generic types: range-types, object-types, enumeration-types, reference-types
- aggregate types: set-types, list-types, array-types

The whole type syntax is described in [5]. To check the correct type of a knowledge base value, the demon concept is very useful. ACTROB provides type-check demons for every basic type and all type constructors. It is the responsibility of the knowledge base to invoke the type checking demons in the right manner. This task can be done automatically using the type describing facet of each attribute.

2.5 Integrity Maintenance

Like the type checking mechanism, it is possible to use demons in general to check integrity rules in the knowledge base. Integrity rules in knowledge bases are often formulated as constraints in the form of a set of mathematical (in)equations. The main advantage of using constraints is that one can express integrity rules in a declarative way. It is not necessary to know explicitly the algorithm which solves the integrity problem. The approach realized in ACTROB is to generate demons for integrity checking and integrity maintenance from a declarative specification [10]. The goal pursued in this approach was to classify constraints under locality aspects.

In order to have a small computation overhead, the specification of constraints is divided into two parts. One part (called def-constraint statement; see below) specifies global conditions on knowledge base states that must be satisfied. Because it is provided to give parameters to a constraint, it is possible to use it in different objects. A special feature is the bind clause which allows to define any sets of knowledge base entities, and which can be used for integrity checking in the condition clause. Depending on the result of the condition an action can be evaluated.

```
def-constraint <constraint-name>
  on plist <param-list>
  { bind <typ> <var> { <term> | <set> } }*
  condition <cond>
  { action <act> }*
```

The second part (called bind-constraint statement; see below) is used to connect any attributes in the knowledge base to the conditions formulated in the def-constraint statement. In the case of an update to some attributes, specified in the on clause, this second part binds the defined conditions to these attributes that must be checked. Using the with clause the right parameters will be propagated to the constraint that must be checked.
This constraint concept allows to take advantage of the locality in integrity checking and integrity maintenance. Only if necessary, the perhaps general constraints are checked.

3 Architecture

3.1 Knowledge Base Process

ACTROB was written completely in C and runs on MicroVax or DECstations under the operating systems Ultrix and VMS. As shown in figure 2, ACTROB was realized as an independent process which is connected to the autonomous systems by a process communication facility, called Ports. Autonomous units can be mobile robots, flexible transport systems or numeric controlled (NC) machines. Also interactive access to ACTROB over a terminal device is possible.

ACTROB KERNEL

ACTROB interface

Object Manager
- KB definition functions
- KB manipulation functions

MERKUR
- tuple layer
- segment manager
- OS calls

disc storage

Figure 2: Architecture of ACTROB

The process structure of the global knowledge base makes it very easy to establish a multi-user interface to the knowledge base. The autonomous units will have a radio link or ethernet connection to the knowledge base running on a central host. Therefore, ACTROB runs as a server process and receives requests from clients (i.e. the autonomous units). Using a remote knowledge base interface, the clients create messages that are routed to ACTROB. ACTROB will receive these messages and invoke the corresponding functions of the object manager. The object manager provides a basic interface for creating, manipulating and querying object-oriented knowledge base entities [4]. After processing such a knowledge base request, ACTROB creates a corresponding response message and sends it back to the client process.

3.2 Database Subsystem

The object manager itself is based on the tuple interface to the database system MERKUR [7]. The tuple layer provides an interface for creating prefix B*-trees and handling tuples in them. Object structures are transformed into tuples and, therefore, permanently stored in prefix B*-trees on disc.

When mapping complex object structures to tuples two variants are possible. First, one can only use a tuple as byte container and provide a type cast mechanism on tuples. The advantage here is that complex objects are stored together. Second, one can split the information of objects into various tuples. Here, sharing of information can easier and more efficient organized. In ACTROB the second way was chosen but tuples yielding information about one object are clustered together [14].

For supporting dynamic scheme evolution there is no distinction between data dictionary and user data. Instead, data and scheme information are stored together. The mapping of whole knowledge base structures to tuples is organized by three kinds of tuples, namely class tuples, attribute tuples and instance tuples. Every tuple has a unique internal identifier supporting object identification and object sharing. Additionally, there is a concept of tuple overflow integrated into the object manager to support long objects. Basically, the structure of tuples is as follows:

id: | name1 | name2 | next | content1 | content2 |
---|---|---|---|---|---
key attributes

Every tuple has an internal identifier, two name fields and an overflow field next used as key of the tuple. These fields are used for clustering object information. Besides these fields there can further fields which contain the information about the objects. You may have as many fields as you like. The values stored in the fields of tuples can be atomic or complex structured; here sets, lists or tuples are allowed.

For example, class tuples contain information about the class topology, the instance relation and the attribute relation. The key of class tuples is the class name and a selector depending on the kind of information stored in the tuple. They are structured as follows:

id: | class | selector | next | attribute ids
---|---|---|---|---

Attribute tuples contain the facet lists describing that at-
tribute. Facets are stored as pairs of facet name and facet value. The key of these tuples is the class name and the attribute name. The structure of attribute tuples is:

| id: | cname | aname | next | [fname1] val1 | ... |

Instance tuples contain the individual values of an instance object. The key of these tuples is the class name and the instance name. The structure of instance tuples is:

| id: | cname | aname | next | val1 | val2 | ... |

The integration of a database subsystem becomes absolutely necessary in the field of manufacturing automation because it brings the following advantages [14]:

1. Persistent storage of data
2. Reliable and safe data management
3. Managing big amounts of data
4. Controlled parallel data access

Why don't we use classical database systems like in some expert systems? First, they are mostly closed systems and the access to data is only possible by the query language interface. In the relational model, for example, structured data must be normalized and may be distributed in different relations. The data access, therefore, is only possible by joins over different relations in the query language. The data access is very slow and not adequate for this problem. Furthermore, classical database systems do not support dynamic changes in schemes once defined. But modelling a factory environment, a good model must be adaptable for all future changes in the factory.

The choice of the tuple layer of MERKUR as database subsystem is due to the background to get all advantages of a database system but to leave out all features which are not suitable. Using the tuple layer all requirements, listed above, are satisfied and we also have the flexibility of storing objects very compactly in tuples. The tuple layer also takes dynamic changes of schemes into account. Tuples and fields of tuples have no fixed length. They can be easily expanded or shrunk [14].

Besides the user-defined objects, which are stored on disc, also user-defined demons can be linked to the knowledge base operator.

3.3 User Interfaces to ACTROB

We will now focus the problem of how to work with ACTROB. Requests to ACTROB are transformed to messages by the remote service interface and afterwards transmitted to it. ACTROB processes these messages and sends a response for these requests back.

Before applications can use a specific knowledge base, application-specific object structures must be created and filled with data. Normally, this is the task of a knowledge administrator and can be done easily by use of the knowledge base shell. It is an interactive tool for defining, manipulating or deleting objects in ACTROB. With this tool, it is also possible to control and monitor data inserted into the common knowledge base. Two shells are available, the graphic-oriented shell using a mouse-pad and a character-oriented shell using VT100-mode. Besides the interactive shells, developed as a human interface to the knowledge base, also a programming interface is available. All basic services for manipulating objects [4] are also accessible by programs. Often used services are those for changing and reading an attribute value of an object. They are defined as follows:

```plaintext
function write-attr-value
    ( var msg-id : integer;
      class,inst,attr : string;
      value : WBvalue;
      wait : boolean ) : integer;

function read-attr-value
    ( var msg-id : integer;
      class,inst,attr : string;
      var value : WBvalue;
      wait : boolean ) : integer;
```

The attribute of an object in the knowledge base is specified by the parameters class, inst and attr. The parameter value is a record structure containing a pointer to a data field and an integer value containing the length of the data field. So any user data computed in applications can be transmitted in or received from the knowledge base. The services can be used in a blocking or non-blocking mode specified by the wait parameter. In time critical situations an application process is able to send data to the knowledge base but it need not wait on a response. It is possible to receive asynchronously responses from the knowledge base later on. In this way the parameter msg-id makes sense because it is necessary in order to assign a response to the corresponding request. Besides using non-blocking mode, a process can wait on an immediate response using blocking mode.

Another, more efficient, way using the knowledge base is to run parts of applications directly in the knowledge base. This is possible using the demon interface (see fig. 3). Demons can operate on knowledge base entities directly by using the object manager. Besides the possibility of modelling application data, also application code can be ported as demons into the knowledge base. These demons can be activated by write or read access to some virtual attributes. For example, a robot can query the shortest way between two points in the factory using the knowledge base service get-attr-value. A suitable demon can then dynamically compute the path from the graph stored in the knowledge base which represents all possible connections in the factory floor.

4 The Active Knowledge Base

As mentioned before, the main reason for implementing ACTROB as an autonomous process, however, was to provide multi-user access to the knowledge base via process communication. Furthermore, the demon concept can now be used to send messages actively to connected processes. Hence, the knowledge base can distribute important data immediately after they have been inserted. The connected processes need no longer poll interesting data (e.g. status flags, sensor information) from the knowledge base.
base but get it automatically as soon as it is available. In this sense, the knowledge base works similar to an (active) blackboard [16], onto which all agents can put their messages but need not worry about how to address them to the right destination.

4.1 Enroling as Interested Process
A process, which is interested to be informed about changes in the knowledge base, can enrol itself as an interested process for these changes (e.g. some attribute values). As soon as a process is no longer interested, it cancels its enrolment as interested process. For these features two services are available. Moreover, a third service for waiting on knowledge base messages is applicable.

i) function enrol-process
   ( var msg-id : integer;
     class,attr,inst,proc : string;
     wait : boolean ) : integer;

ii) function cancel-process
   ( var msg-id : integer;
     class,attr,inst,proc : string;
     wait : boolean ) : integer;

iii) function wait-for-asyn-message
    ( var msg-id : integer;
      value : WBvalue ) :
    integer;

Using these functions an application process can wait on multiple events in parallel:

- The application process tells the knowledge base on which attributes it is interested by using i). For every attribute, the process-id is added to the process list in its receiver facet. Then, it waits for any message from the knowledge base. This can be done using an interrupt concept with asynchronous routines [8].

- The first change of one of these attributes causes that a message will be sent to a waiting process by means of a common applicable communication demon.

This communication demon is activated when an attribute is overwritten. Then, it looks for all interested processes in the receiver facet, and sends them an appropriate message.

- If a process is no longer interested, it cancels its enrolment as interested process by using ii).

The principles of how communication demons work are shown in figure 3. The communication facility is not restricted to one host only. Instead, all hosts in the local area network are reachable.

4.2 Integration by Means of Active Data Distribution
The knowledge base is the central tool for bringing together various components of a manufacturing system (see fig. 2). This integrating function mainly is made possible by the concept of actively sending demons [13]. The knowledge base, therefore, is not merely a (more or less) comfortable database or storage device. Moreover, it becomes an active component for controlling the flow of information taking place in a complex hard- and software system like a factory environment with autonomous robots and other intelligent agents.

By means of the enroling mechanism, the interface between processes can be defined in terms of object-oriented data description, rather than in those of communication protocols. This is very flexible and can be adapted easily to all kinds of applications.

Changing design or implementation decisions affecting these interfaces is possible without greater effort by only changing data representations. Algorithms, however, are rarely affected.

A further advantage of the concept of actively sending demons is that all processes can perform various actions all time and must not wait for a certain value. Also, no polling is necessary to get some values; changed values are distributed to interested processes as soon as possible. Communication takes place via Ethernet and through mailbox events that interrupt an interested process to indicate that new data has come in from the knowledge base [8].

The knowledge base, therefore, can be compared to an (active) blackboard [16]. One process writes its order or task onto the blackboard. Then, this order or task is automatically addressed to those processes which are responsible for processing it. Acknowledgements or receipts can be exchanged the same way.

A further advantage of our active knowledge base concept is the possibility to modularize large applications in a natural way. All modules can be developed and implemented independently by just using dummy values in the knowledge base. Also, testing and maintaining a single module rarely affects the others.

Figure 3: Message distribution using communication demons
5 ACTROB in CIM environments

Information processing in CIM environments requires flexible, problem-oriented models of shop-floor environments. It deals with concrete geometrical data of machines, tools, manipulators and vehicles, but also with abstract descriptions of tasks, plans, flowcharts and other knowledge on how to manage the whole. In addition to that, dynamically incoming sensor information, status reports and time-critical jobs must be processed as well as the static knowledge mentioned above. All these requirements can be solved by the proposed concepts of ACTROB.

![Figure 4: CIM environment in SFB 331](image)

The CIM environment proposed in SFB 331 is shown in figure 4. The global production planning control (PPC) of the flexible manufacturing system gets the customer orders. It roughly estimates the consuming of resources through the production process and afterwards distributes orders to the autonomous units (manipulators, vehicles, DNC machine tools). Then the local production control of every unit can partly optimize (e.g. split and/or re-order) the incoming jobs. Finally, it sends on a sequence of tasks to the task coordinator of the autonomous unit. The task coordinator decomposes the tasks in actions with possible parallel execution and passes them to the modules of the task execution layer. Furthermore, it synchronizes these modules, which have to perform and control the single actions by sending them to the machine layer.

For ACTROB was developed as a global knowledge base inside the joint research project SFB 331 the successfull use in a CIM environment could be demonstrated. Because no access to a real factory environment was possible, a complex demonstration application from the blocks-world was chosen. A wall of bricks with "windows" and "door-frames" was to be built on a rotating platform by a PUMA 560 manipulator. Within this demonstration the principles of flexible manufacturing automation could be presented [3].
The virtual state of the building process (in instance ACTUAL SHAPE). The type facet of the attribute structure shows the kind of representation of the wall, i.e. a two-dimensional matrix (10 x 7) of integers. An if-added-after demon is used to visualize the shapes on a monitor.

The class DEPOT is used for the administration of two depots of blocks. They are described by the two instances MAIN DEPOT and FITUP DEPOT which contain all information about rows and blocks. There are four types of blocks, namely B1, B2, B3 and B4. For every type of block a set of rows containing these blocks is defined. Additionally, the number of blocks and rows in the depot are defined by attributes of class DEPOT. The attribute ORIGIN holds the coordinate system of the depot, in the form of a homogeneous 3 x 4 transformation matrix. The virtual attributes of class DEPOT are used to hide algorithms behind read- and write-access functions. When a program calls the knowledge base, for example, with the two services

\[ \text{put-attr-value} \left( \text{DEPOT, MAIN DEPOT, block param, B3} \right) \]
\[ \text{get-attr-value} \left( \text{DEPOT, MAIN DEPOT, virtual in pos} \right) \]

the position of a free place in the MAIN DEPOT is computed into which a block of type B3 is put. The access to a virtual attribute is exactly the same as for reading an other attribute. But here, an if-needed-after demon, started at the end of the get-attr-value service, computes the desired coordinates. \text{Virtual_out_pos} works analogously when a block of some type will be used for the next step of construction. The virtual attributes \text{virtual_put_in} and \text{virtual_get_out} work similar to the virtual attributes above but they are used when a block is physically moved by the PUMA 560 manipulator.

Figure 6 shows the attributes of the instance of class POSITION which contain all information necessary to transport a block from a certain place to another one. Further classes in our demonstration are TASK, ROBOT, ROW and BLOCK. The class TASK provides an interface between the local production control and the task coordinator. It works similar like the class ORDER.

All data in the knowledge base can be accessed by all manufacturing tasks running in this CIM environment (see fig. 4). As the knowledge base works as an autonomous process, an interprocess communication, based on DECnet and shared memory, provides the interface to distributed processes on various computers.

Note in figure 4 that no direct communication is performed between the application, production control and task coordination levels. They are parallel processes, independent from each other, and they are connected only to the global knowledge base. This architecture becomes absolutely necessary in the SFB 331 environment, where the task coordinators, for example, are in tended to work "on-board" autonomous vehicles. All vehicles will have a radio link to a central host (many to one), because this is easier to establish than a direct communication between each of them (many to many). The central host will contain a production planning control (PPC) system, supervisor tools and all global knowledge to provide the necessary information about paths in the factory, geometrical models of manipulators and machines, sensor data, plans and tasks.

One of the most important features when realizing complex systems by larger teams is the modularizing effect obtained by the knowledge base. Independent modules can be reused and easily combined for new applications. For example, it was possible to use the task coordinator completely unchanged in a factory environment where an autonomous mobile robot had to supply a machining center with workpieces [8].

Defining of interfaces between the modules in CIM applications is now very easy and naturally in terms of common objects and communication demons in the global knowledge base. Therefore, interfaces are now part of the model of the domain. Because the modelling process in a manufacturing environment proceeds step by step, ACTROB takes into account that objects can change dynamically.

Details of communication protocols are hidden completely behind the remote service routines [13]. Only one process link, namely, that to ACTROB, must be provided. This becomes important if one thinks of autonomously moving vehicles which cannot ensure many-to-many communication on the factory floor.

Independent developing and testing of modules in CIM applications is quite easy now. Every programmer must only define the particular couple of objects necessary for later communication. By just inserting some test values into these objects with the interactive shell it is possible to simulate the partners of the
intended communication and to test the own part independently. Integrating all components in our example was not much more than using one common knowledge base process on the same computer, and then running all modules together.

6 Conclusion

ACTROB is a very useful tool for building knowledge bases for a CIM environment. The object-oriented modelling approach provides a natural description of real worlds with complex object structures which can be found in factory environments with mobile robots and working cells. Also the type concept for attributes, integrated in ACTROB, together with the class concept and inheritance offers a dedicated modelling facility. A knowledge base constructed by means of ACTROB can be changed dynamically by adding or deleting attributes of classes, for example. Relational databases, in comparison, allow their users, perhaps, to add columns to a table, but never to delete one. The dynamic properties, however, are very interesting when changes occur in the factory environment.

The integration of a database subsystem provides many advantages. The persistent storage of objects without loss of any data is necessary because normally the life time of objects in a CIM environment is longer than process run-time. Furthermore, the mapping from objects (instances) to tuples of a database leads to a favourable kind of locality. Every object/tuple can normally be fetched by one disc access. Also the controlled parallel data access is possible by integration of a database.

A feature, used very extensively in ACTROB, is the demon concept. It is suitable for consistency checks or even an automatic, consistent update of values according to previously defined integrity constraints [5], [10]. Also the type-check facility was realized by demons. In order to make the knowledge base active the demon concept together with a process communication interface was absolutely necessary. The active role of the knowledge base plays the central part for the integration postulated in the last section.

The process interface of ACTROB allows its very flexible integration in heterogeneous hardware environments. Therefore, ACTROB can run independently on a separate computer (perhaps on special-purpose hardware) to provide very fast data access.

Future work will be focused on defining a retrieval language for objects. We will also work on the problem of distributed knowledge bases. For performance reasons, that is not quite good up to now, an object cache together with demons running in parallel will be integrated into the current implementation of ACTROB.

References


