Centrum voor Wiskunde en Informatica

Centre for Mathematics and Computer Science

H.M.C.L. Kempenaar

A parallel expert system shell

Computer Science/Department of Software Technology Report CS-R8925 June

1989

Centrum voor Wiskunde en Informatica
Centre for Mathematics and Computer Science

H.M.C.L. Kempenaar

A parallel expert system shell

Computer Science/Department of Software Technology Report CS-R8925 June

The Centre for Mathematics and Computer Science is a research institute of the Stichting Mathematisch Centrum, which was founded on February 11, 1946, as a nonprofit institution aiming at the promotion of mathematics, computer science, and their applications. It is sponsored by the Dutch Government through the Netherlands Organization for the Advancement of Research (N.W.O.).

A Parallel Expert System Shell

Christine Kempenaar
Centre for Mathematics and Computer Science
P.O. Box 4079, 1009 AB Amsterdam, The Netherlands

A description is given of a parallel rule-based expert system shell, which is the result of a redesign of the existing sequential DELFI2 system. This effort was undertaken in order to investigate the potentials of parallel inference in such an expert system shell. The system has been developed in the language POOL-T (Parallel Object-Oriented Language-Target). In this report, the knowledge-representation and inference schemes used in the system are discussed. In particular the basic approach in exploiting explicit, coarse-grain, parallel inference and the consequences for the overall system architecture are treated.

1980 Mathematics Subject Classification: 68T99.
1982 CR Categories: C.2.4 [Distributed Systems]; I.2.1 [Artificial Intelligence]: Applications and Expert Systems.

Key Words & Phrases: expert systems, parallel knowledge processing, object-oriented programming.

1. Introduction

This report is the result of an experiment in which the application of parallel inference in a rule-based expert system was studied. The particular approach that has been investigated is called *coarse-grained parallel inference*, in which parallelism is explicitly specified in a knowledge-representation scheme. In coarse-grained parallelism in general, the amount of data relative to the number of instructions executed in one computational process is relatively large. As a vehicle for the experiment, a prototype expert system shell has been implemented in a parallel object-oriented programming language called POOL-T. The latter is a language that provides constructs for explicitly indicating parallel processing. The family of languages called POOL has been developed by Philips Research Laboratories, Eindhoven in the Esprit project 415 (DOOM, Decentralized Object-Oriented Machine) and the SPIN project PRISMA (Parallel Inference and Storage Machine). The aim of both projects is the development of a coarse-grained multi-processor computer, and the study of various applications for their suitability of exploiting parallelism.

The expert system shell described in this report is an initial effort towards the development of a system in which a parallel inference engine is incorporated. At the start of the project, after about two months library research, it became apparent that the subject matter was still an unexplored area, with very little material to build upon. The few articles that had been published on the subject of parallel inference techniques all described medium to fine-grain parallelism. This was contrary to the aim of this research project, that was specifically aimed at incorporating explicit coarse-grain parallelism.

Based on this conclusion and the lack of material concerning this form of parallelism the obvious conclusion was to concentrate on existing sequential expert system shells. From the shells that were considered, the DELFI2 shell seemed the most appropriate, because this system offers facilities to divide a knowledge domain into a number of subdomains. It seemed natural to apply explicit parallelism using a similar representation by defining a local inference engine for each predefined subdomain. This paper focuses on the structure of such an inference engine and the coupling of multiple inference engines, distributed according to the structure of the knowledge domain. Our working hypothesis is, in effect, that the structure of the knowledge domain can be used to guide the distribution of the inference task. In other words, the network of inference processes reflects the structure of the knowledge domain, in order to achieve effective (coarse-grained) parallelism. Considerable time

Report CS-R8925
Centre for Mathematics and Computer Science
P.O. Box 4079, 1009 AB Amsterdam, The Netherlands

has been spent in building a concrete system based on this concept.

The structure of this report is as follows. First, in section 2 the principles of simple sequential expert system shells are reviewed in general. This section provides the background for the remainder of this paper. In section 3 details of the parallel expert system shell are described and the important issues of process creation and synchronization are also treated in this section. In section 4 a possible experimental approach is described for testing the prototype with regard to parallelism and an informal discussion is presented of some pencil and paper experiments. Section 5 provides conclusions concerning the research project and some recommendations for further investment are proposed.

2. Principles of sequential expert systems

Expert system shells have evolved from several efforts that were undertaken in the seventies to generalize particular expert systems. To this end the specific domain knowledge for which these systems were originally designed was extracted in the hope that the system that remained would also be applicable to the representation of other problem domains. Expert system shells are also known as 'tools for building expert systems' and 'empty shells'. Examples of expert system shells, based on production rules are:

ROSIE, developed at the Rand Corporation;
OPS5, developed at the Carnegie Mellon University;
EMYCIN, developed at Stanford University;
KAS, developed at Stanford University;
DELFI2, developed at the Technical University of Delft.

EMYCIN is the domain independent version of MYCIN, a medical expert system for the diagnosis of certain infectious diseases and for recommending appropriate drug treatment. Using OPS5 a system known as R1 or XCON has been developed, which is used in configuring VAX-11/780 computer systems.

The principles described in this paper are mainly based on a stripped down version of the DELFI2 empty shell. More information about this system can be found in article [1].

Most expert system shells can be divided into the following major components:

(1) A *knowledge base*, containing specific knowledge in some particular domain;
(2) The *inference engine*, which does not itself contain any domain-dependent information, but instead uses the contents of the knowledge base to infer new facts from data and knowledge;
(3) A *user interface*, possibly including explanation facilities, for interacting with the user.

In the next two paragraphs the knowledge base and inference engine of the sequential expert system shell are discussed. It only provides, to refresh the memory, a very simplistic view on this matter. In reality the most expert system shells in operation are much more sophisticated then both the parallel and sequential shell described in this paper.

2.1. The knowledge base

All specific domain knowledge is stored in a knowledge base, which consists of two components: an *object tree*, and a set of *production rules*. The domain of discourse is described in the object tree; the production rules represent the problem solving knowledge. In the object tree the relational structure of these objects and their mutual dependency is represented. Furthermore, each object has several associated attributes.

A consultation of such a system is aimed at obtaining values for those *attributes*. When this aim is achieved successfully those three parts are combined and delivered as a fact having the following form <object, attribute, value>. The collection of facts is called a *factset*. A *trace class* label is defined for each attribute, describing in what way values should be determined. The determining of attribute values is called *tracing*. The possible trace classes are: goal, ask, askfirst, and database. Attributes belonging to the goal class will use production rules to trace a value, ask attributes will be determined from answers given by the user to questions. An askfirst attribute is asked to the user, and is only

inferred by using production rules if the user is not able to enter information for the attribute. The values for database attributes will be looked up in a predefined fact file.

The entity 'fact' in conjunction with a predicate is called a 'subgoal'. A similar construction can also be found in the antecedent and consequent part of the production rules. As will be discussed in more detail in the next paragraph, new facts found in the consequent part of the rule can be added to the factset, when a number of already known facts fulfil the antecedent part of this rule.

The directives for the inference engine are stored by means of production rules. A production rule is considered to have the following syntax:

```
<rule>       ::= if <antecedent> then <consequent> fi

<antecedent> ::= <clause> { and <clause> }

<clause>     ::= <condition> { or <condition> }

<condition>  ::= <predicate> <object> <attribute> <constant>

<predicate>  ::= same | notsame | lessthan | ...

<consequent> ::= <conclusion> { and <conclusion> }

<conclusion> ::= <action> <object> <attribute> <constant>

<action>     ::= conclude
```

The predicates within conditions relate the information represented in the set of facts and referred to by the first argument (the object-attribute pair) to the constant value specified in the second argument. The actions specified in the conclusions have an effect on the facts referred to by the first argument and assign to them the constant value mentioned in the second argument.
Consider the following production rule:

```
if same object2 attribute1 d and
  same object2 attribute2 e
then
  conclude object2 attribute3 a
fi
```

In applying this rule and having a factset which that contains at least the facts 'object2 attribute1 d' and 'object2 attribute1 e' will result in the creation of the new fact 'object2 attribute3' with the constant-value 'a' assigned to it. Then facts will be created and stored in the factset during a consultation of the system. The factset is a list structure containing <object, attribute, value> triples as entries.

2.2. The inference engine

This paragraph consists of two parts. First, the inference engine is presented in an informal way, intermingled with some procedures in pseudo-code. The second part contains a small example.

The inference engine is used to find new facts by using a technique called *backward chaining* which is also known by its synonym: *top-down inference*. The term backward chaining refers to the way the production rules are examined. By using this technique, the system first looks at the conclusion part of a production rule and then tries to establish the accompanying conditions. In *forward chaining* or *bottom-up inference*, the conditions are examined first. If all these conditions succeed when evaluated, this leads to the conclusion of the rule being true and the specified action being executed.

The expert system will be considered as a collection of procedures, which represent the important

purpose of tracing new facts through application of information from the knowledge base and the factset. Two methods will be distinguished to trace these facts: the system will first attempt to infer facts from the knowledge base (by a procedure named `Infer`) and when this is not successful, a second alternative way is tried, which is to ask the user to enter relevant information (by a procedure named `Ask`). The aim of consulting the system is to trace values of those attributes, that are predefined and stored in a file called the goalfile. If an object tree is used, those attributes will be inferred that have been defined as being of the 'goal' trace-type. This operation, called tracing, can be formulated in pseudo-code as follows:

```
Procedure Trace(rulebase, factset, fact)
  Infer(rulebase, factset, fact);
  if fact not traced then
    Ask(factset, fact) fi
end Trace
```

The inference of a (sub)goal using production rules could be stated as:

```
Procedure Infer(rulebase, factset, fact)
  SelectRules(rulebase, fact, selected-rule);
  traced ← false;
  while (selected-rule ≠ nil) and not traced do
    if selected-rule not used then
      Apply(selected-rule, factset, traced) fi;
    if not traced then
      selected-rule ← next.selected-rule fi
  end-do
end Infer
```

To realize this inference, the system picks a goal attribute and starts examining the production rules. All production rules, containing the goal attribute within the conclusion part, are selected. In this way, a set of rules is chosen that is suitable for this specific goal. (See the procedure `SelectRules`.)

```
Procedure SelectRules(rulebase, fact, rules-selected)
  rules-selected ← nil;
  rule ← rulebase;
  while rule ≠ nil do
    if conclusion == fact then
      new.selected-rule;
      selected-rule.ruleref ← rule;
      selected-rule.link    ← rules-selected;
      rules-selected        ← selected-rule fi;
      rule ← next.rule
  end-do
end SelectRules
```

Subsequently, the system selects one rule out of this set and draws attention to its condition part. It tries to verify the condition, because if all conditions in a certain rule are valid, the conclusion of the rule is also valid and the associated goal can be put in the factset. The evaluation, or application of a production rule is described in the following procedure `Apply`.

```
Procedure Apply(selected-rule, factset, traced)
  traced ← false;
  EvalConditions(selected-rule, factset, failed);
  If not failed then
    EvalConclusions(selected-rule, factset);
    traced ← true fi
end Apply
```

Invocation of the procedure `EvalConditions` triggers the investigation of the factset in order to find resemblance between the conditions of this rule and certain facts in the factset. If this resemblance is established, the associated conclusions are added to the factset (by the procedure `EvalConclusions`). On the other hand if this resemblance is not found, a recursive call to procedure `Trace` will be made and this condition is supplied as the next subgoal argument. If all these attempts to trace a value are unsuccessful, the last possibility to obtain the value is to ask the user for this value. In the above pseudo-procedures, parameters are passed by 'call by reference'.

Example.
The knowledge base in this example has only one object called: 'object'. This object contains the following attributes: a, b, c, d, e, f, g, h and i. The only constant value used in this example is the value 'true'.
The production rules are:

```
1. if same object c true and
      same object d true and
      same object e true
   then
      conclude object a true
   fi

2. if same object f true and
      same object g true

   then
      conclude object b true
   fi

3. if same object h true
   then
      conclude object c true and
      conclude object d true and
      conclude object e true
   fi

4. if same object i true
   then
      conclude object f true and
      conclude object g true

   fi

Goals:
object a = ?
object b = ?

Factfile:
object i true
```

The system starts procedure `Trace`(rulefile {rulenumbers = 1,2,3,4}, factfile {fact = object i true}, fact {goals = b,a}) and continues with procedure `Infer`(rulefile, factfile, fact). This results in selecting rule 2. Next, procedure `Apply` is invoked with rule 2 as an argument, leading to a recursive call of the procedure `Trace` in order to find values for the attributes f and g. Because 'object i true' is already contained in the factset, the attributes f and g can be traced by using rulenumber 4. At this point for the previous subgoal 'object b' is established its constant value to be 'true'. Trying to trace goal 'object a ?' results in the same sequence of procedure calls, except the last step will be questioning the user to deliver the value for attribute h. If the answer happens to be 'true', the goal 'a' is also traced. Meanwhile the factset is expanded with the attributes f, g, b, c, d, e, h, a and their associated truth values.

3. Towards a parallel expert system shell

3.1. Overview of parallel inference

Several kinds of parallelism can be distinguished. In this section I will only briefly mention some of these. First, the exploiting of parallelism using a *dynamic approach* is described, thereafter its counterpart, the *static approach*, will be mentioned. The phrases, dynamic and static, used in this section refer to different mechanisms of process creation that are involved. In the dynamic approach it is not known in advance how many processes will be created during a program execution. For example, in logic programming the processes may be recursively created until some termination criterion is fulfilled. Thus, the number of processes to be created is not established in advance. In the static approach the control of process creation is laid down beforehand in a number of predefined processes that will operate concurrently during the program execution.

In the area of logic programming one also finds the following two basic approaches to parallelism:

- OR-parallelism: a process is assigned to solve the body of every clause that is active, that is every clause whose head unifies with a given goal.
- AND-parallelism: a process is assigned to solve each of the subgoals in the body of an active clause.

These forms of parallelism have a high level nature. This type of parallelism is applied in searching a resolution search space, where in all cases a complete body of a clause or even a procedure defined by a set of clauses is involved. Parallelism on a lower level is search parallelism: the program can be divided into disjoint sets of clauses so that several processes can search for clauses whose heads unify with a given goal in parallel, each working on a different set.

The first two mentioned above could be stated as the evaluation of an AND/OR tree and do have a dynamic nature [14]. The other provides a more static form of parallelism.

If we consider parallelism in an expert system based on Horn clauses then this can be considered as expanding these clauses. This means that all the 'and' and/or 'or' clauses, contained in a body, will be investigated concurrently by separate processors. For example, expanding the next two Horn clauses:

C1: if x_i and ... and x_n then y_j
C2: if v_k and ... and v_m then x_i

will result in rewriting the atom x_i in clause 1, by v_k and ... and v_m. This way clause 1′ becomes:

C1': if v_k and ... and v_m and x_{i+1} and ... x_n then y_j.

In the 'and' parallel execution, the system requires a consistency check when unbound variables are used in common in the body, because here could arise a variable binding conflict. The consequence of this rewriting mechanism is, that at the point where an atom in the body of this rule is dependent on other rules, these body atoms in turn will be expanded. This continues until a clause cannot be expanded any further. In this way a voluminous number of small processes arises. Because every clause is treated in the same manner, it is not possible to establish in advance the number of clauses that eventually will be created when a recursive evocation of rules is possible. This pictures the dynamic nature of this technique.

A more static approach is applied in the technique used in the OPS5 system. OPS5 [13] is a system, where knowledge is represented by production rules. The technique exploited here, is to translate all the production rules into a Rete graph. The Rete graph is a directed acyclic graph, where process nodes are created for all conditions and by these process nodes the conditions are tested; links represent conjunctions of conditions. If this test comes out with a positive result, the inference process continues to investigate the next node; if the test was unsuccessful the remaining nodes in this graph will be abandoned. The number of processing nodes will be known in advance.

The next part describes a rule-based system on a completely different machine architecture. This is the DADO architecture, which may be interpreted as the opposite of the PRISMA architecture. The

difference can be expressed as follows:
- DADO contains a large number of processors (in the article about DADO [10] the number in the order of a hundred thousand has mentioned), PRISMA on the other hand, has considerably less available (in the order of hundred).
- The local memory of each processor in the DADO architecture is approximately 16 Kbytes, while the local memory of PRISMA will be much larger (over 8 Mbytes).

Thus DADO is a fine-grain machine, whereas PRISMA is a coarse-grained machine. This is why DADO is more suitable for using forward chaining techniques, since these lead to the same fine-grained structure. First some general information is given, followed by an algorithm, which uses forward chaining.

Consider Production Systems (PS) as a set of production rules, which form the Production Memory (PM), together with a database of assertions, called Working Memory (WM). Each production consist of a conjunction of pattern elements, called the Left-Hand-Side (LHS) of the rule, along with a set of Actions called the Right-Hand-Side (RHS). This RHS specifies information that is to be added to (asserted) or removed form WM when the LHS successfully matches against the contents of WM. The DADO machine consists of a set of processing elements (PE's). They are interconnected to form a complete binary tree. Each PE is capable of executing in either of two modes controlled by running software.
- SIMD mode (single instruction stream, multiple data stream). The PE executes instructions broadcast by some ancestor PE within the tree.
- MIMD mode (multiple instruction stream, multiple data stream). Each PE executes instructions stored in its own local RAM, independently of other PE's.

Each PE has a simple rule matching mechanism. A small number of distinct production rules are distributed to each of the PE's, as well as all WM elements relevant to the rules in question. Each PE executes the match phase for its own small PS. One such PS is allowed to 'fire' a rule, however, which is communicated to all other PE's.

Some important differences between this DADO approach and the system presented in this paper for the PRISMA machine are:
- DADO uses a forward chaining mechanism; The empty shell in this paper is using the backward chaining technique.
- Only a few production rules resides on every PE, whereas in our prototype the total number of production rules in one KB-object can be well over a hundred.
- DADO's database called WM contains assertions and those are stamped with a time tag. The assertions can be added and removed. Removing facts is at this moment in our system not allowed.
- The difference in grain size. DADO is fine grained, while PRISMA is coarse-grained.

It clearly depends on the problem involved, which technique must be preferred. Problems in the field of predictions, for example weather forecasts, are quite suitable for forward chaining techniques.

All earlier mentioned forms of parallelism give rise to exhaustive search spaces with a fine grain size. The opposite can be found in static parallelism, in which the control is explicitly defined beforehand. The explicit form allows for a much coarser grain of the structure. This coarse-grained structure affiliates in a natural way with the PRISMA machine. To arrive at an explicit form of parallelism, the expert system will be divided into concurrently operating subparts, according to the objects of its production rules and laid down in the object tree. All the production rules that contain the same object in its conclusion part are gathered together and put into the same knowledge-base object. In this way it is known in advance how many knowledge-base objects (KB-objects) will be cooperating in the inference engine. For each KB-object a local inference process is created. By this logical division, laid down in the KB-object tree a static approach comes into existence that has been exploited in the development of our parallel empty shell. This approach leads to an explicit control mechanism to enable parallelism and contrasts with the unrestrained implicit parallelism provided in some Prolog-like systems.

The parallel expert system shell presented here is considered to be suitable for parallelizing expert

systems in which a number of modular components can be recognized. In some technical applications such a modular set-up, where modules have a high degree of independence, is possible. Such technical applications can be found in a domain of fault diagnosis for which some expert systems have been developed in the past. For example RAFFLES, CRIB, DART are existing expert systems that diagnose computer hardware and software malfunctioning. B747/ATA-21, a system which supports trouble-shooting in the Boeing 747 airconditioning, is another example [11]. In those systems one is unable to point out beforehand the failure provoking part, so it could be advisable to investigate all the different components at the same time. With this approach in mind it becomes feasible to attach to the expert system an automatic control system with sensors collecting all the information needed from the several components.

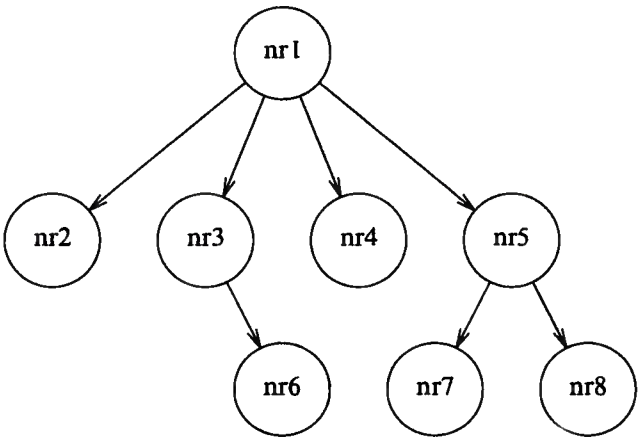


Figure 1: Object tree.

3.2. Object division and process creation

All information in the knowledge domain is represented by object, attribute, value tuples. By defining a structure on this knowledge domain it is possible to obtain an explicit form of parallelism. For example, if the knowledge domain is a block world, the following entities could have been defined: block length 12 inch; block colour red; block weight 100 kg. In this example block is the object; length, colour and weight are the attributes belonging to this object and their respective values are 12 inch, red and 100 kg. To obtain the structure we will divide the knowledge domain into a number of different objects. All tuples containing the same object are grouped together to form an entity called 'object'. The set of disjunctive 'objects' that originates in this way will be organized in a tree structure (figure 1). The root of this tree is formed by the most general knowledge-base object and the leaves contain the more specialized knowledge-base objects. Specific information needed to create the desired tree in a certain domain must be provided by the knowledge engineer. Furthermore, it is his responsibility to provide the information in such a way that optimum benefit is gained from this division. This object division will result in the creation of the different local inference engines, which in turn represent the parallelism of the system. For every object in the tree a process is allocated which starts up its local inference process. This object division reflects the control of the several parallel operating processes and it also represents the grain size (coarse-grained) of the parallelism. So the very important effect of parallelism resides on this level. The parallelism is achieved by the top object in the tree. This Root object, allocated on its own processor, starts up its own inference engine and meanwhile it will activate its children to start up their local inference mechanism. The children in turn will occupy separate processors. (See figure 2.)

Obviously, the object tree also reflects the communication pattern between the various processes. This is due to the fact that on a certain moment during the inference cycle a KB-object could need some information about an object-attribute pair that belongs to another KB-object (another local inference process), so the two inference processes need to communicate with each other about this. For this communication a protocol is provided that is described in detail in paragraph 3.4.

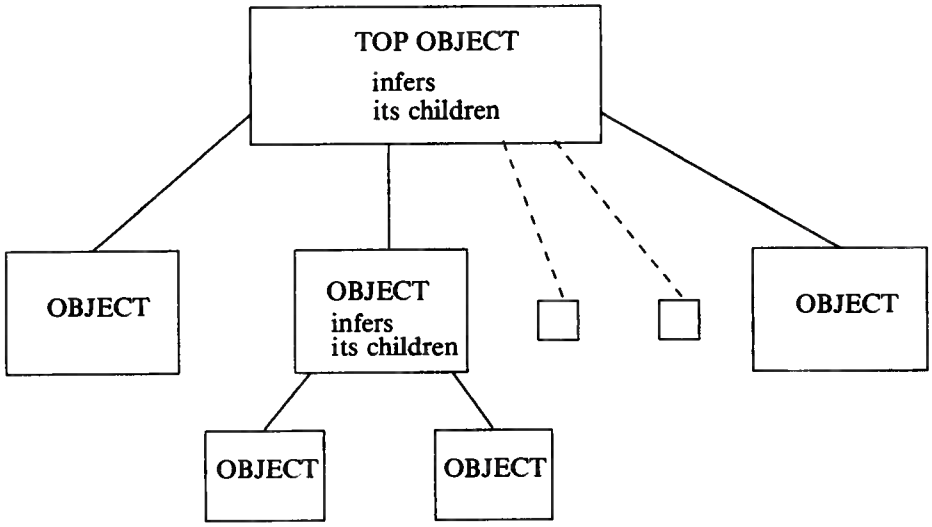


Figure 2: Inference processes.

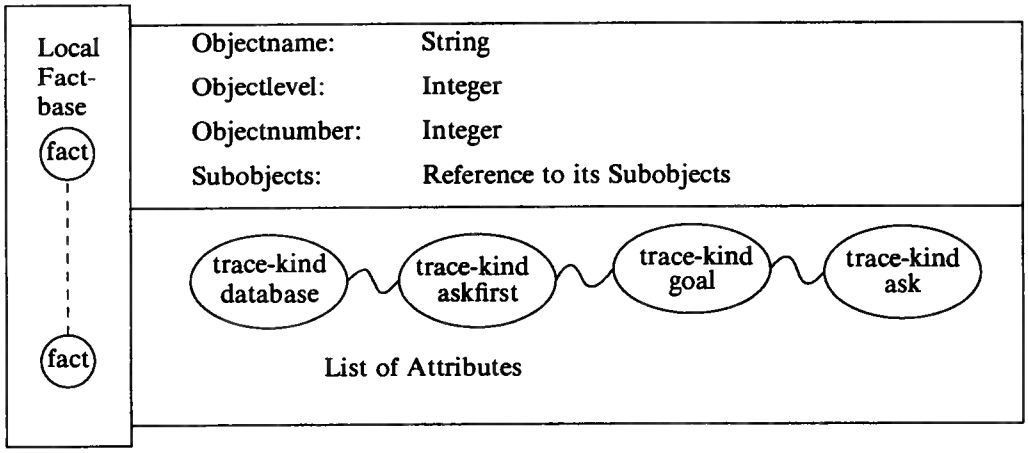


Figure 3: Inside an Object.

3.2.1. The structure of objects.
In each object (figure 3) the following data parts can be distinguished:
- Attributes,
- Facts.

Furthermore, the Knowledge-Base object itself is described by its name, a number, the related superobject and subobjects, and the level it is situated on. The root or top object is situated on level 1. The specifications of attributes is treated in the next paragraph. A factset should be considered as a local database for the object they belong to. In some other systems a factbase is maintained by a global 'blackboard' [8,9]. In order to achieve a really significant amount of parallelism in a distributed knowledge base, local storage of at least part of the factset is essential.

The factbase contains entities (the facts) in the form of a tuple: <object, attribute, value>. During execution the local database is used to store all traced facts. Not only the facts belonging to this object are stored in the factbase, but also certain facts that have been obtained from a subobject are stored in this local database. These last facts have come into existence when a subobject was invoked to trace a subgoal and this lead to a positive result. More details about the functioning of the different parts in a KB-object can be found in paragraph 3.4.

3.2.2. The structure of attributes.
As shown in the previous paragraph, each object has its own set of attributes. These attributes consist, among other parts, of a name (type String) and a value (type Value) and have a certain tracetype (database, askfirst, goal or ask). (See figure 4.)

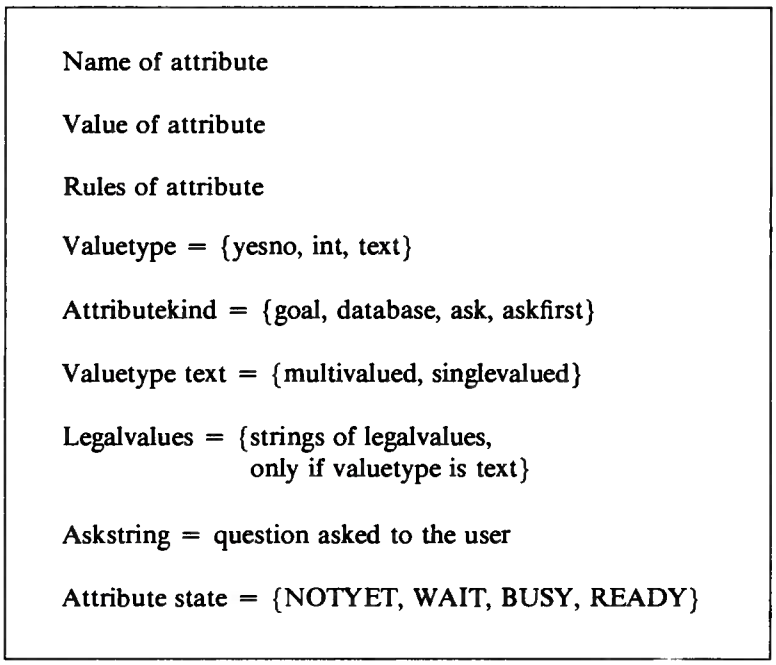


Figure 4: Inside an Attribute.

The Value is one of the following types:
- The int type, or
- The yesno type, which is functionally equal to a boolean type, or
- The text type, which stands for a String when an attribute is defined as singlevalued, or stands for

an array of Strings, whenever it is defined as being multivalued.
As will be discussed in detail in paragraph 3.4 the aim of the system is to find values for the attributes. When the program is started all attributes are initially assigned the empty value 'nil'. To achieve finding values for the attributes, each goal attribute has a number of rules associated with it. These rules can only be accessed by this particular attribute, no other attribute has the privilege of using a rule belonging to another attribute. During the inference process attributes can be in one of several states {NOTYET, READY, BUSY, WAIT}. These states represent the processing sequence of the attributes. Initially, all attributes are in state NOTYET. At a particular instant during the inference process, a goal attribute will be activated to release a subgoal (condition) for further investigation. At the same time its state will turn into WAIT. When the subgoal is traced, the state of the attribute changes to BUSY. This means that the conclusion part of the production rule is now being evaluated and the value contained in this conclusion part is assigned to this particular attribute. When this part is completed the attribute enters the READY state.

The parallel inference process is synchronized by these attribute states. Whenever the inference process (a KB-object) wants to trace an attribute value that belongs to another KB-object, the two processes involved should be synchronized until the particular attribute has entered its READY state. This is an obvious constraint because only when the attribute is in READY state it can deliver any useful information.

3.3. Knowledge base

The parallel shell may be combined with a knowledge base containing specific domain knowledge to form an expert system. This knowledge is partly described by means of production rules and partly in an object tree. The object tree contains the following domain information:

- Dependency information, that is the way the objects are related to each other by a specification of parent-child relationships;
- Information concerning the accompanying attributes of each object with allowed values;
- The trace-class type of each attribute (askfirst, database, goal or ask).

When dealing with a goal attribute, the associated rule numbers to this specific attribute should be provided also in the object tree.

The production rules represent the problem-solving knowledge in this domain. The production rules should conform to the restrictions laid down in the object tree. This means that rules situated in a particular object only may contain conditions that refer to the same object or to a successor object downward in the tree. This form of consistency is not automatically checked for at this moment.

3.4. Parallel inference engine

The description of the parallel inference engine in this and subsequent sections, takes a variant of the parallel object-oriented programming language POOL-T as a descriptive framework [2,3,4,5,6,7,12]. The programming language POOL-T provides the user with the ability to define entities, called objects. An object is described by its internal data and accompanying procedures manipulating this data. Procedures are either called *methods* or *routines* in POOL-T. Methods are the only way to manipulate the internal data, thus providing the hiding of information. An object may also be regarded as an active process by itself. One of the activities of an object is communication with other objects. The communication between objects is done by means of the rendez-vous mechanism (synchronized communication). The actual communication takes place by sending a message from the source object to a destination object. The message not only contains the name of the destination object, but also the name of a method that must be applied by the destination object. For example, the message 'localfactbase!putfact(subgoal)' is sent to the object which is the value of the variable 'localfactbase'. In the object, the method 'putfact' with argument 'subgoal' is executed upon answering the message. How and when a message is answered, is decided by the destination object itself after it has received the message. The source object is idle until the message has reached its destination. This leads to the synchronization of the processes. POOL-X, an extended successor of the

POOL-T language, also provides an asynchronous communication mechanism. The global behaviour of an object is defined in its BODY and consist, generally, of selectively answering a number of messages. Objects are instances of a class. A class specifies several objects of the same kind.

The inference mechanism is primarily based on backward chaining, as previously described in paragraph 2.2. However, within one object a form of *forward updating* also occurs, in the following way. Once a condition (predicate, object, attribute, value) has been traced it is forwarded, not only to the attribute that caused this subgoal to be investigated, but also to all goal attributes within this object that are not as yet in a READY state. (This is done by sending the message setconditiontraced(subgoal, trace-boolean).) Goal attributes receiving this subgoal react by putting attention to their accompanied rules, to find whether these rules contain conditions equal to this subgoal. All conditions meeting this equality requirement store the appropriate trace value in their variable 'traced' . This approach has been chosen to avoid the same condition/subgoal within one object being traced more than once.

The parallel nature of the system is contained in the fact that each object initiates a 'local' inference process for itself. This local inference process follows a sequence of stages. As described in paragraph 3.2.2 the attributes within one object are included in a list, the order being determined by their trace-type (database, askfirst, goal or ask-type). If present, database attributes are dealt with first. These attributes are looked up in a global database file (named: 'fact.file') that contains their value. This information should already be in existence, before the consultation is started. The attributes are then assigned this value. Simultaneously for each attribute that has been assigned a value this is stored as fact <object, attribute, value> in the local factbase that belongs to this object. Subsequently, the askfirst attributes will be dealt with. To assign them a value the user is asked a specific question that belongs to this attribute. Finally the collection of goal attributes is traced. With each goal attribute is associated a set of rules. The system selects the first rule for investigation and if this rule fails, then subsequent rules, if available, will be investigated. This process continues until a rule is found to be valid. As already mentioned in paragraph 3.2.2 (structure of attributes), some attributes are multivalued. When such a multivalued goal attribute does occur, then all the available rules are being used one by one.

Investigation of a selected rule consists of checking whether the conditions within that rule are true. Within a rule only 'and' operators are allowed between conditions and conclusions. If all conditions prove to be true then the rule is valid. If at least one condition is found to be false the rule fails. When a condition has been checked at a previous stage, this fact is stored in the local factbase. If a condition is being investigated that is not yet included in this local database two strategies are possible, depending on the object name that is included in this condition. The first strategy described, represents the local inference process inside its own KB-object (figure 5), the second strategy pictures the global inference mechanism (figure 6), in case another KB-object (inference process) is being involved.

In the first strategy, the inference engine which is active in the present object checks whether the object name in the given condition is equal to its own object name. When this equality is established the condition will be traced recursively within the present object, using only the attribute name. This leads to either another goal attribute becoming eligible for inference, like described above, or the querying of a so-called ask attribute to the user. This is the only way in which ask attributes can be activated, it is the only situation when they usefully contribute towards tracing a particular condition. For their value assignment, ask attributes depend on the answer provided by the user to the question posed by this attribute. In figure 5 an inference cycle is shown in which only one inference process is involved. The subgoal is locally inferred, thus no communication with other processes is needed.

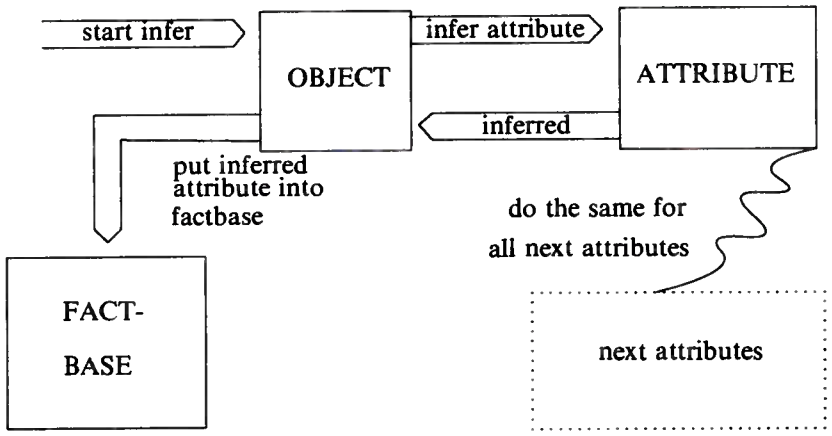


Figure 5: Structure chart of inference mechanism.

The second, global, strategy applies when the inference engine has established that a condition belongs to another object, causing that condition to be sent as a message to an object that may be eligible. (Sending the message 'subs!trace(subgoal)'.) As described in the paragraph concerning the tree-like architecture of objects, a parent-child relationship exists between them. This relationship reflect the communication pattern between objects. The objects taking part in the communication are the parent object asking a child object for certain information and the child object, which is prompted to provide an answer.

The information, needed by the parent is whether a certain condition holds. The parent will ask all its children, one by one, by sending it the condition, until it has found the one with the correct name.[1] The child object first extracts the objects name of the condition that is being asked for. If this does not equal its own name, then it will return the question to the parent with the remark not to be the child that is looked for. In case the name is right it will also always give an answer, either positive or negative, depending on the established truth of the condition that is being asked for. Describing the communication relation this way, as being asymmetric (a child is not allowed to ask questions to his parent), it is possible to prove the avoidance of deadlock (see 3.5). Furthermore, infinite waiting states will be avoided, because a child will ultimately come up with an answer. Figure 6 shows the inference steps when more objects are involved. The following sequence is distinguished:

(1) The top object has received the message startinfer, which causes all inference processes to be started up.
(2) At a certain moment a goal attribute will be activated to infer its value.
(3) The attribute will pass a subgoal to the object in order to infer its value.
(4) In turn, the object will examine its local factbase, to check whether this subgoal can be traced by using only local facts.
(5) If this does not succeed, the next step is to invoke another object that may contain information with regard to this subgoal.

1. This is eligible to optimization.

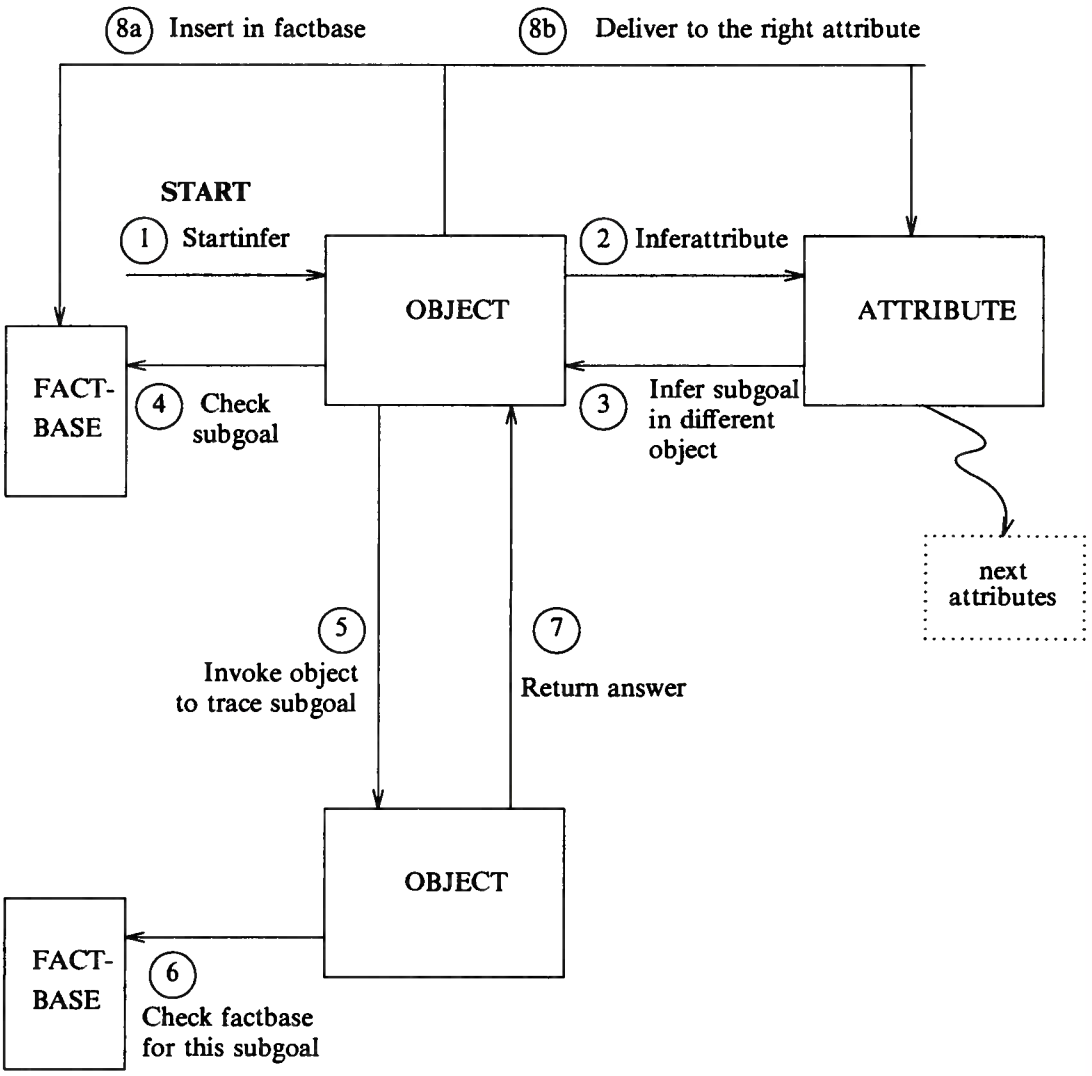


Figure 6: Structure chart with more objects involved.

(6) First, the invoked object examines its own local factbase, if this does not lead to the desired result, the next step is checking the status of the attribute mentioned in this subgoal. When this attribute is not yet in a READY state, the object will have to wait. After the attribute has entered its READY state, the object will check its local factbase again and returns the answer to the object that started the invocation. The returned answer is either positive or negative, depending on the success or failure result of the subgoal evaluation.

(7) The answer is returned to the object.

(8) The object passes the answer to the attribute that was involved in the inference, so the attribute can store the value that has been traced. If the subgoal was established it will also be stored in the local factbase, that belongs to this object.

We next describe the parallel inference engine in pseudo-code. The first method, `startinferobject`,

starts up all inference processes.

```
Method startinferobject() Object:
   DO for all its children
      subs ! startinferobject()
   OD
   RETURN SELF
   DO for all its attributes (sequential)
      subgoal ← attributes ! inferattribute(attribute)
      IF action of subgoal = conclude THEN
         localfactbase ! putfact(subgoal)
      ELSE
      trace(subgoal)
      FI
   OD
END startinferobject
```

The following method, `trace`, describes the mechanism for tracing a subgoal. First the object name is extracted from the subgoal. If the subgoal belongs to another inference object, the subobjects are invoked by the message `subs ! trace(subgoal)`; otherwise the subgoal will be inferred locally.

```
Method trace(subgoal) Boolean:
   IF ownobjectname ≠ objectname of subgoal THEN
      answer ← subs ! trace(subgoal)
   ELSE
      check ← localfactbase ! checksubgoal(subgoal)
      DO while not-check THEN
         status ← attribute ! givestatus(attributename of subgoal)
         IF status = READY THEN
            check ← localfactbase ! checksubgoal(subgoal)
            IF check THEN answer ← traced
            ELSE answer ← not-traced
            FI
         FI
      OD
   FI
   attribute ! setconditiontraced(subgoal, answer)
   RETURN answer
END trace
```

The method `inferattribute` is identical for all local inference processes. It depends on the trace class of the attribute involved, which action will be taken. It is the purpose of the procedure to find a subgoal that could be traced. In this case the goal attributes are the most important, because they have rules associated to it in order to trace their value. The subgoal is passed to the inference process, which will try to trace this subgoal.

```
Method inferattribute(attribute) Subgoal:
   traced ← FALSE
   IF ownkind = askfirst THEN
      subgoal ← askattribute()
      traced ← TRUE
   ELSIF ownkind = database THEN
      subgoal ← getoutdatabase()
      traced ← TRUE
   ELSIF ownkind = goal THEN
      ruletraced ← ownrules ! isruletraced()
      IF ruletraced THEN
         subgoal ← ownrules ! giveconclusion()
         status ← BUSY
      IF valuetype ≠ multivalued OR
         (no more rules AND no more conclusions available) THEN
         traced ← TRUE
      FI
      ELSE subgoal ← ownrules ! givecondition()
         status ← WAIT
         IF subgoal belongs to ownobject THEN
            newat ← subgoal ! attributename()
            subgoal ← inferattribute(newat)
         FI
      FI
   ELSIF ownkind = ask AND correct-name THEN
      subgoal ← askattribute()
      traced ← TRUE
   FI
   IF traced = TRUE THEN status ← READY FI
   RETURN subgoal
END inferattribute
```

3.5. Sketch of deadlock avoidance

In order to informally prove deadlock avoidance, a technique can be used in which a directed graph represents the communication pattern of processes. Using this technique, it is sufficient to show that a cycle will never occur in this graph. However, if unfortunately a dependency cycle should occur during the inference process, our approach provides a solution to this problem and so avoiding a real deadlock. In the case under discussion, there are two different locations in the inference process that are important in order to show this. First, the dependency cycle must not occur when the communication takes place between related attributes inside the same object. Secondly, a deadlock must not occur when two different KB-objects (processes) are involved in the communication. Consider the next two rules, that are situated in one KB-object.

```
Rule1:
IF same aircraft place in-the-air THEN
   conclude aircraft state flight FI

Rule2:
IF same aircraft state flight THEN
   conclude aircraft place in-the-air FI
```

Within the KB-object 'aircraft' we find two (sub)goal attributes, i.e. 'state' and 'place', in that order. Those two will result in a dependency cycle. During the inference process the attribute 'state' will be considered first. Subsequently, the value of the 'place' attribute is required, because this is a condition in the corresponding rule 1. In addition, this rule will be marked as 'used' so the same rule shall not be used again at a future occasion. During this action the 'state' attribute is assigned the WAIT status. At this moment the status of the 'place' attribute is still NOTYET. Subsequently, the system tries to derive the 'place' attribute. Rule 2, that belongs to 'place' provides its condition (same aircraft state flight). Meanwhile, it sets the 'place' attribute also in WAIT status. Next, the system

investigates the attribute 'state' in object 'aircraft' and concludes that it is in WAIT status and has not been assigned a value yet. This causes the inference within this object to leave those two attributes alone for the moment and to start searching for other goal attributes that could be derived. It could well be that during the derivation of subsequent attributes a subgoal is encountered, which could be of consequence to the WAIT attributes mentioned before. By means of the previously described mechanism of forward updating this subgoal is also presented to these two WAIT attributes, which might now be enabled to derive their rule using this subgoal. So, if this happened, they could come to a conclusion concerning their own value, thus solving the conceptual deadlock that occurred. If a deadlock is not resolved in this manner, and all attributes within an object have had their turn, then the system will conclude that two or more goal attributes cannot be derived. This results in a question being posed to the user whether she or he might be able to provide a value for that goal attribute. If that is the case, other attributes that contributed to this cycle may be derived using forward updating. If the user cannot provide an answer the next goal attribute in the dependency chain will be presented. This process continues until all goal attributes in the chain are treated. In the used example, when no solution has come up, the system will ask the user if he/she can give the value for the 'state' attribute. At the end of a consultation, when the system presents its results to the user, the system will inform the user which rule(s) it used to draw its conclusion. By a rule number that equals zero (0) the system indicates that a goal attribute has not been derived by the system itself but was obtained from the user.

When two or more KB-objects are involved, then this can be pictured by the following rules:

```
Rule1:
IF same child child-attribute oke THEN
   conclude parent parent-attribute oke FI

Rule2:
IF same parent parent-attribute oke THEN
   conclude child child-attribute oke FI
```

In this case, where the inference process involves two or more objects that could cause a cycle, it would have to be first determined at which level the two objects are situated in the tree. It is important here to distinguish between parent and child objects, as only the parent has access to the child's information, as described in paragraph 3.4, whereas the child can never obtain information from its parent. This prevents a deadlock cycle to occur. If during the reasoning process the parent asks the child for information, and the child needs an attribute value from the parent, this would simply cause the child to hand the question over to its own children. This results in the answer NOTFOUND, because this child has no children that could provide the required information. At the same instant the 'child-attribute' marks its rule 2 as not traced. The answer NOTFOUND is also returned to the parent, who will mark rule 1 as not traceable. Possibly, more rules are available for the 'parent-attribute' that could be tried next. If this is not the case it is impossible to derive the value of this 'parent-attribute'. The approach is illustrated for two directly subsequent objects, however, it can be generalised when more objects are involved. The chain of objects in between does not remove or add information crucial to the underlying reasoning process.

4. Some pencil and paper experiments

The implementation of a coarse-grained parallel program in which not only the conceptual modelling of parallelism is represented, but that also provides the means for investigating the achieved speed-up by exploiting the parallelism, requires more effort from the programmer than has been supposed until now. For the effective exploitation of explicit parallelism it turns out to be important to know how to map active POOL objects on the architecture of a particular multiprocessor machine, in this case the PRISMA machine. This requires some effort from the programmer to add allocation directives to the program through which a particular distribution of POOL objects on the machine is accomplished.

We have not investigated this approach due to time limitations. However, some pencil and paper experiments have been carried out using the approach of parallel inference discussed in the previous pages and these will be discussed in the present section.

The basic idea is to divide a knowledge base in such a way that a suitable configuration for parallel inference is achieved. In order to establish the difference in processing speed almost identical knowledge bases are used. This means they contain exactly the same objects, attributes and rules.

In example I all the inference is being done inside one knowledge-base object. This represents the case in which all inference is done sequentially, by one process. (See figure 7.) In the other examples, three parallel inference engines do their job on the different child objects. (See figure 8.) The configuration of these three objects will be the same in all three parallel test cases. The fourth object, which is the top object, will be changed in the three distinct examples, in which parallel inference is investigated. The knowledge base contents is shown in figures 7 and 8. The database attributes are a1, a2, a3 and a4 and they have a 'yes' value assigned to it. The other attributes in the example will use rules to obtain their values; the rule number they respectively use is also shown in the picture.

Next, we will describe a simple mechanism for counting the processing times of the different operations. The allowed operations and their measurements are:

(1) Assigning a value to a database attribute will cost: m_D;
(2) The processing used for obtaining values for the attributes during local inference will be charged with: m_I;
(3) The communication with other inference processes leads to an additional measurement m_C for the communication overhead. Thus, the total operation cost for those attributes is: $m_I + m_C$.

After each inference step some administration will be needed; for example the obtained fact should be stored in a local factbase. However, this administration costs will not be taken into account in the overall picture, because they are considered to be approximately of the same magnitude in all different inference steps. The initialisation will not be considered for either of the test-cases, in order to be able to obtain a real picture of the respective behaviour.

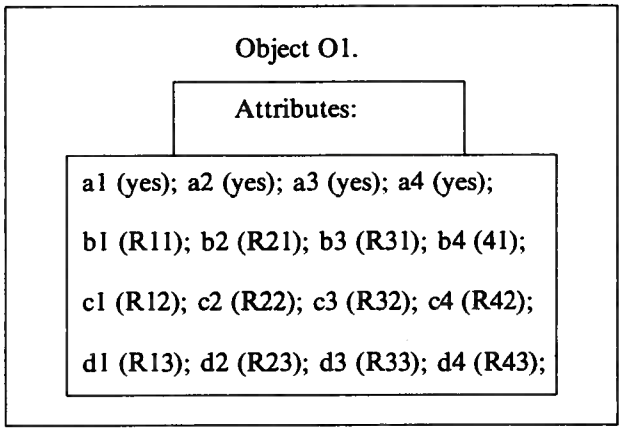
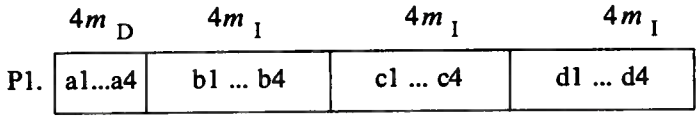


Figure 7: Sequential test structure.

The rules that have been used in example I are:

```
R11:IF same o1 a1 yes THEN conclude o1 b1 yes FI
R12:IF same o1 b2 yes THEN conclude o1 c1 yes FI
R13:IF same o1 c3 yes THEN conclude o1 d1 yes FI
R21:IF same o1 a2 yes THEN conclude o1 b2 yes FI
R31:IF same o1 a3 yes THEN conclude o1 b3 yes FI
R41:IF same o1 a4 yes THEN conclude o1 b4 yes FI
R22:IF same o1 b2 yes THEN conclude o1 c2 yes FI
R32:IF same o1 b3 yes THEN conclude o1 c3 yes FI
R42:IF same o1 b4 yes THEN conclude o1 c4 yes FI
R23:IF same o1 c2 yes THEN conclude o1 d2 yes FI
R33:IF same o1 c3 yes THEN conclude o1 d3 yes FI
R43:IF same o1 c4 yes THEN conclude o1 d4 yes FI
```

With the above measurements information, example I, sequential test, needs $4m_D + 12m_I$ time units to finish its inference.



Example I.

We next, discuss the result of the three parallel test-cases. In figure 9, the object tree is shown which represents the process relation for the examples II, III and IV.

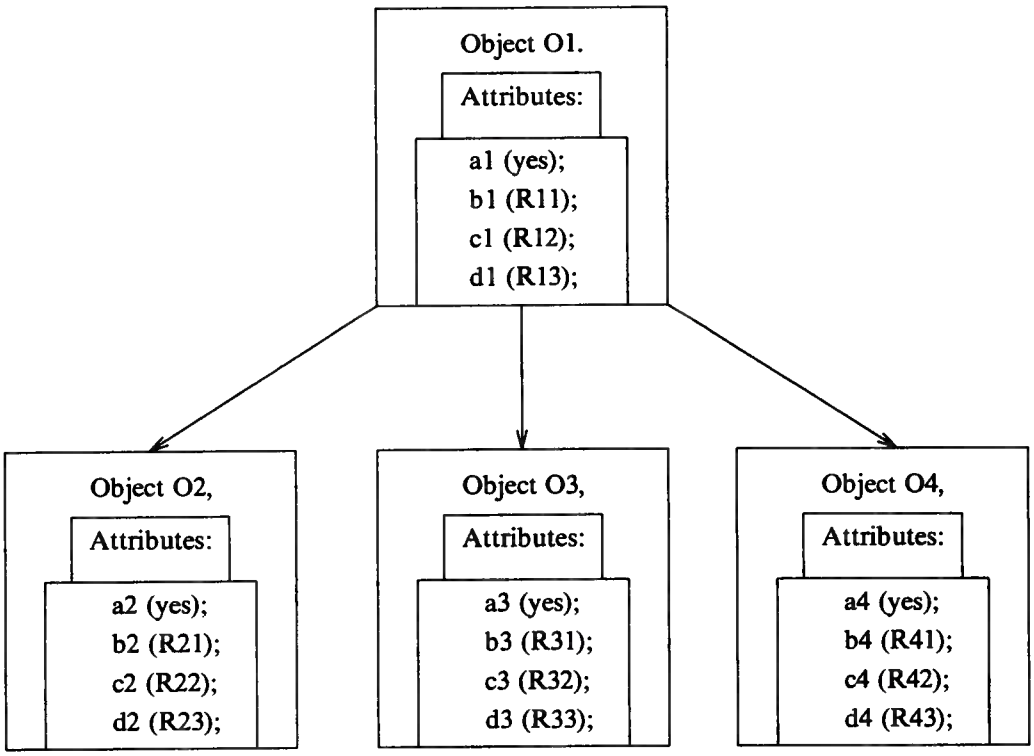


Figure 8: Parallel test structure.

The rules in example II are:

```
R11:IF same o1 a1 yes THEN conclude o1 b1 yes FI
R12:IF same o2 b2 yes THEN conclude o1 c1 yes FI
R13:IF same o3 c3 yes THEN conclude o1 d1 yes FI
R21:IF same o2 a2 yes THEN conclude o2 b2 yes FI
R31:IF same o3 a3 yes THEN conclude o3 b3 yes FI
R41:IF same o4 a4 yes THEN conclude o4 b4 yes FI
R22:IF same o2 b2 yes THEN conclude o2 c2 yes FI
R32:IF same o3 b3 yes THEN conclude o3 c3 yes FI
R42:IF same o4 b4 yes THEN conclude o4 c4 yes FI
R23:IF same o2 c2 yes THEN conclude o2 d2 yes FI
R33:IF same o3 c3 yes THEN conclude o3 d3 yes FI
R43:IF same o4 c4 yes THEN conclude o4 d4 yes FI
```

The above rules show the information needed to draw a certain conclusion. For example, Rule 12, describes that Object O1 needs to trace the value for its attribute c1. In order to do this, Object O1 will invoke Object O2 to check the condition 'same o2 b2 yes'. This operation will cost process 1 (Object O1): $m_I + m_C$ time units. Attribute d1 in Object O1 is traced in the same way by rule 13. The accumulated costs of all operations are shown in picture Example II. In example II process 1, represented by Object O1, needs $m_D + m_I + 2(m_I + m_C)$ or $m_D + 3m_I + 2m_C$ time units. The processes 2, 3 and 4 (Objects O2, O3, and O4) finish their job in $m_D + 3m_I$ time, all three in the same amount of time. These times also apply on the examples III and IV, only the processing time of process 1 varies.

	m_D	m_I	$m_I + m_C$	$m_I + m_C$
P1.	a1	b1	c1	d1
P2.	a2	b2	c2	d2
P3.	a3	b3	c3	d3
P4.	a4	b4	c4	d4

Example II.

In example III, rule 11 is changed into:

```
R11:IF same o2 b2 yes THEN conclude o1 b1 yes FI
```

this results in the measure for process 1 to be:
m_D + idle-m_I + $3(m_I + m_C)$ or $m_D + 4m_I + 3m_C$ time units. Due to the changing of Rule 11, process 1 (Object O1) needs to wait until Object O2 has finished the inferring of the value for its attribute b2. Thus process 1 is idle during this time.

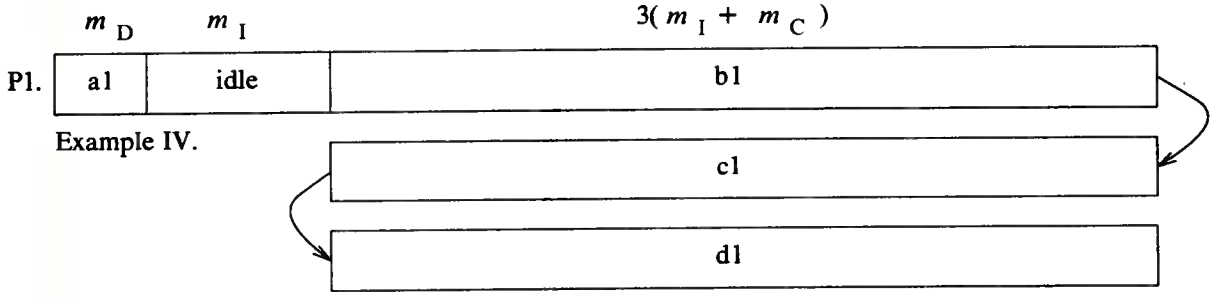
	m_D	m_I	$m_I + m_C$	$m_I + m_C$	$m_I + m_C$
P1.	a1	idle	b1	c1	d1

Example III.

In example IV, the following rules are changed:

```
R11:IF same o2 b2 yes AND same o3 b3 yes AND
       same o4 b4 yes THEN conclude o1 b1 yes FI
R12:IF same o2 c2 yes AND same o3 c3 yes AND
       same o4 c4 yes THEN conclude o1 c1 yes FI
R13:IF same o2 d2 yes AND o3 d3 yes AND
       same o4 d4 yes THEN conclude o1 d1 yes FI
```

In this last example three different Objects will be invoked to trace the value for the o1 b1 object-attribute pair. This swallows a tremendous amount of processing time and the consequence is shown in picture Example IV. In this example process 1 is engaged in its inference job for:
m_D + idle-m_I + 3(m_I + m_C) + 3(m_I + m_C) + 3(m_I + m_C) or m_D + 10m_I + 9m_C time units.



Example IV.

The examples I and II, show that a time profit is obtained of $3m_D$ + $8m_I$ - $2m_C$ time units. In this case it is assumed that $2m_C$ is considerably less than $3m_D$ + $8m_I$; thus real time benefit is gained in example II. However, when we compare example I with IV, this results in a 'profit' of $3m_D$ + $2m_I$ - $9m_C$. It is obvious that the communication overhead of $9m_C$, largely exceeds the quantity of $3m_D$ + $2m_I$. Naturally, practical experiments, in contrast to these pencil and paper experiments, must be carried out before real figures can be produced of the communication overhead. These examples give only some idea about the different processing times that are possible and that even a small change upon a single rule does have its influence. It shows also that when the right division of a knowledge base is found, real benefits can be obtained from this coarse-grained parallel approach.

5. Conclusions and recommendations about prototype

The current state of the parallel rule-based expert system shell has been described in this report. In this section some issues will be treated that would merit further research.

For the development of our system we chose a coarse-grained form of parallelism. This approach is contrary to existing systems based on Parlog or the DADO machine, where the approach has led to a fine-grained structure of parallelism. The PRISMA machine, for which the empty expert system shell described in this report was developed, is only suited to accommodate a coarse-grained parallel process structure. This structure was achieved in our system by means of initially defining the control over the parallel processes, based on a partitioning of the knowledge domain. The knowledge domain is divided in a number of subparts (KB-objects). Their mutual relationships are defined in a tree structure (KB-object tree). The nodes in this tree represent the parallel processes. The pencil and paper experiments which we used as test cases showed that this approach can lead to satisfactory results. These results, however, are totally dependent on the partition into parallel processes (represented by the KB-objects). The communication between these processes is a delaying factor. Therefor, this communication should be limited to the minimum in order to obtain an optimal speed-up factor. At the development phase of an expert system it is generally not possible to establish the most effective division into parallel functioning components. Hence, much research effort will be needed to find among these different configurations the most beneficial one.

Although the empty expert system shell presented in this paper is relatively self contained, the following parts could be added or optimized. All inference processes in the system are started implicitly. One could consider to develop a control mechanism that would allow for a more explicit control in

order to achieve that only those inference engines will be started that contribute effectively during a specific consultation. This would render the remaining inference engines redundant at that moment. At the moment a parent process needs information from a child process, it will have to solicit this information from the appropriate child, and not bother all its children, as is the case at present. This condition is caused by the fact that the parent now addresses its children by means of a number, which should be changed in an access of the children by their names. At present the process communication within the system is severely restricted. Processes on a equal level in the process tree cannot exchange information among each other. Neither can a child obtain information from its parent process. Obviously, these restrictions could be lifted by adding a global fact set (blackboard) to the system, with as disadvantage a significant communication overhead.

ACKNOWLEDGEMENTS

I would like to thank the CWI that offered me the opportunity to acquire valuable practical experience and Peter Lucas, who brought the research subject to my attention and scrutinized several sections of this report. Especially I owe much to Anton Eliëns, who supported me during the first period in developing a concrete approach from the research subject and furthermore for his valuable comments and suggestions on draft versions of this paper.

REFERENCES

[1] H. de Swaan Arons, E.P. Jansen, P.J.F. Lucas, H. Stienen (1985). DELFI2 handleiding, Onderafdeling der Wiskunde en Informatica, TH Delft.

[2] P. America (1985). Definition of the programming language POOL-T. *Esprit Project 415A Document 91.* Philips Research Laboratories, Eindhoven.

[3] L. Augusteyn (1985). POOL-T User Manual. *Esprit Project 415A, Document 104.* Philips Research Laboratories, Eindhoven.

[4] L. Augusteyn (1985). POOL-T Standard Environment. *Esprit project 415A, Document 138.* Philips Research Laboratories, Eindhoven.

[5] J.P. Katoen (1987). User manual of Parpl, a POOL-T Simulator. *Esprit project 415A, Document 333.* Philips Research Laboratories, Eindhoven.

[6] J.P. Katoen (1987). Simulation of DOOM, a loosely coupled multi-processor system. *Esprit Project 415A, Document 299.* Philips Research Laboratories, Eindhoven.

[7] H. Muller, M. Beemster (1987). Manual pages of pl, a POOL-T interpreter. *PRISMA Project, Document P38.* Philips Research Laboratories, Eindhoven.

[8] L.D. Erman, F. Hayes-Roth, V.R. Lesser, and D.R. Reddy (1980). The HEARSAY-II speech understanding system. *Computer Surveys, Volume 12.*

[9] L.D. Erman, P.E. London, and S.F. Fickas (1981). The design and an example use of HEARSAY-III. *Proc. of IJCAI'81.*

[10] S.J. Stolfo (1984). Five parallel algorithms for production system execution on the DADO machine. *Proc. of AAAI'84.*

[11] Boeing 747 Airconditioning. Master Key: 747 environmental control system, Hamilton Standard.

[12] P. den Haan (1986). SODOM user's manual. *Esprit project 415A, Document 170.* Philips Research Laboratories, Eindhoven.

[13] B.K. Hillyer, D.E. Shaw (1986). Execution of OPS5 production systems on a massively parallel machine. *Journal of Parallel and Distributed Computing, Volume 3.*

[14] M.V. Hermenegildo (1986). An abstract machine based execution model for computer architecture design and efficient implementation of logic programs in parallel. *Department of Computer Sciences,* The University of Texas at Austin.