OR and AI approaches to decision support systems

Citation for published version (APA):

Hee, van, K. M., & Lapinski, A. K. (1987). *OR and AI approaches to decision support systems*. (Designing decision support systems notes; Vol. 8702). Technische Universiteit Eindhoven.

Document status and date:

Published: 01/01/1987

Document Version:

Publisher's PDF, also known as Version of Record (includes final page, issue and volume numbers)

Please check the document version of this publication:

- A submitted manuscript is the version of the article upon submission and before peer-review. There can be important differences between the submitted version and the official published version of record. People interested in the research are advised to contact the author for the final version of the publication, or visit the DOI to the publisher's website.
- The final author version and the galley proof are versions of the publication after peer review.
- The final published version features the final layout of the paper including the volume, issue and page numbers.

[Link to publication](#)

General rights

Copyright and moral rights for the publications made accessible in the public portal are retained by the authors and/or other copyright owners and it is a condition of accessing publications that users recognise and abide by the legal requirements associated with these rights.

- Users may download and print one copy of any publication from the public portal for the purpose of private study or research.
- You may not further distribute the material or use it for any profit-making activity or commercial gain
- You may freely distribute the URL identifying the publication in the public portal.

If the publication is distributed under the terms of Article 25fa of the Dutch Copyright Act, indicated by the "Taverne" license above, please follow below link for the End User Agreement:

www.tue.nl/taverne

Take down policy

If you believe that this document breaches copyright please contact us at:

openaccess@tue.nl

providing details and we will investigate your claim.

**OR AND AI APPROACHES TO DECISION
SUPPORT SYSTEMS**

by

K.M. van Hee
and
A. Lapinski

NFI 11.87/02

EINDHOVEN UNIVERSITY OF TECHNOLOGY
Department of Mathematics and Computing Science
P.O. Box 513
5600 MB EINDHOVEN, The Netherlands

September 1987

OR and AI approaches to decision support systems

by

Kees M. van Hee and Antek Lapinski

**Department of Mathematics and Computing Science
Eindhoven University of Technology**

1. Abstract

First the concept of a decision support system (dss) is described. Then the OR-approach, in which the development of a network of models forms the kernel, is given. An example illustrates this approach. Then, architectures of a dss according to the OR-approach and the AI-approach are given. Afterwards an AI-approach is considered, in which an (abstract) dss machine is described that can be tuned to specific decision situations. Finally the applicability of this approach is illustrated with an example.

2. The concept of a decision support system.

Decision support systems (dss) may be considered as a class of **expert systems** (cf. [So84 p.280]).

Before we give our definition of a dss, we will consider some definitions of expert systems first. A compact definition is due to [Ja86]:

"An expert system is a computing system capable of representing and reasoning about some knowledge-rich domain with a view to solving problems and giving advise".

Systems that satisfy our definition of dss also fulfil Jackson's definition and therefore these systems may be called expert systems.

In the definition of Jackson an expert system is characterized by the **tasks** it may fulfil, hence by its functional behaviour. On the other hand expert systems are sometimes characterized by their architecture. According to that approach an expert system is made of a knowledge base, i.e. a set of rules and facts which is the same as a set of formulas in some logic, and an inference machine which performs deductions using a knowledge base and inference rules. A shell consists of a knowledge base management system, an inference machine and a user interface. To create an expert system one has to fill a shell with facts and rules. Sometimes a dss has this architecture, but many dss's have a different structure. Therefore some authors (cf.[Re83]) consider an expert system as a special kind of dss.

We consider a dss as a subsystem of an **information system** and therefore we first define information systems. An information system fulfils two tasks for some **target system**. Examples of target systems, also called object systems, are companies as a whole, departments of companies and small production units. The tasks an information system performs are : **monitoring** and **control** of **state transitions** of the target system.

An information system may consist of a human organisation and computer systems. Large parts of the monitoring task are nowadays performed by computer systems. The control task is often performed by persons, called decision makers. Besides the monitoring task computer systems assist decision makers by reporting and analysing the registered information to obtain knowledge of the mechanisms of the target system.

A dss is a computerized part of an information systems that **consults** decision makers with their control task by :

1. **Computing** the **effects** of actions that the decision maker proposes. We call this: evaluation of actions.
2. **Generating** of **actions** that optimize some criterion function, chosen by the decision maker.

Often the evaluation and generation of actions proceeds in an iterative way. For the evaluation of actions there are **evaluation functions**. These functions are defined by the decision maker in the operational phase of the system, or they are defined by the designer of the dss in the design phase.

We assume the ranges of these functions are some totally ordered sets (sometimes we assume that it is the set of real numbers). Often the evaluation functions are **conflicting**. Two evaluation functions E_1 and E_2 are said to be conflicting if there exist two actions a_1 and a_2 such that

$$E_1(a_1) < E_1(a_2) \text{ and } E_2(a_1) > E_2(a_2).$$

There are several ways to deal with such a problem. One way is to define some linear combinations of the evaluation functions and for each one some bound. Then one of the combinations has to be optimized under the constraint that the other linear combinations don't exceed their bound. A facility to help the decision maker in choosing these linear combinations and bounds is called a facility for **multicriteria analysis** and is often considered to be an essential facility of a dss. Another feature of a dss is a facility for **sensitivity analysis**. The evaluation of the effect of an action requires a mathematical model of a part of the target system.

Such a model contains parameters that are obtained from several resources, such as estimates based on historical data of the target system or hypotheses from a decision maker. To get confidence in the advises of a dss a decision maker wants to see the influences of variations of parameter values for parameters he is not sure of.

This is called sensitivity analysis.

Of course a dss has to have an adequate **user interface** which allows the decision maker to update parameters, to retrieve and compare already computed actions and their effects and to control the evaluation and generation processes.

Dss are used for operational planning and strategic planning

both. The first type of planning requires the optimal **assignment** of resources. Typical examples are jobshop planning and vehicle routing. The second type of planning requires the optimal determination of **capacities** of resources, such as the volume and locations of depots.

A dss for operational planning is used frequently while a dss for strategic planning is used incidentally. This difference reflects in different architecture of human interfaces.

3. Models in decision support systems

The use of operational research (or) techniques to assist decision makers is much older than the field of dss.

Traditionally OR-specialists analysed the decision situation and selected or designed a mathematical model to describe the set of feasible actions and their effects.

Then they designed algorithms to compute actions that are optimal with respect to some criterion, for instance a linear combination of effects. Finally they paid attention to the system-design. Hence in this phase they designed a database for parameters, actions and their effects, and a user interface. In the traditional approach it did not make much difference if the system was used by the OR-specialist as an intermediary between the decision maker and the model, or by the decision maker itself. In the last case the system should be more faultproof than in the first case. In fact the OR-specialist made a system to automatize his own work instead of a system to assist a decision maker. At the end of the seventies OR-specialists changed their views. The dss-concept as described in section 2 was born. A system that could assist a decision maker without interference of an OR-specialist became the target of their design efforts. Optimization was no longer a goal as such, however it became an approach to generate actions that could be considered as proposals to a decision maker. The dss has to propose actions that satisfy the needs of the decision maker and not in the first place some abstract criterion.

Nowadays **adaptability** of a dss for changes in the decision situation is one of the most important characteristics of a dss. Consider a dss in which some constraint on feasible actions is described by a linear inequality. Suppose that the structure of the decision situation changes such that this constraint has to be replaced by a quadratic inequality. Often a dss can't accept such a change without a serious modification of the model and the software. A lot of dss in practice had a short life according to this kind of problems.

There is a tradeoff between adaptability and **efficiency** of a dss. For the generation of actions usually algorithms are used that exploit the structure of the model of the decision situation, for instance if the model is a linear program. However, to obtain a high degree of adaptability these algorithms must use as less as possible of structural details of the model that are expected to

be charged in future.

The types of models that are used to build dss's are simulation models, queueing models, linear and nonlinear programming models, combinatorial optimization models and Markov decision models. The first two types are mainly used for evaluation of actions while the other models are used to generate actions. Time will almost always play a role in a decision situation. Sometimes however it is not necessary to represent time in a model. For instance if the decision maker has to take a decision for only one planning period and if the effect of this decision will not influence the decision situation after that period, time will play no role. Such decision situations can be called stationary. If a model is developed for a stationary decision situation it is usually difficult to adapt the model if it turns out that the decision situation is M-stationary.

It seldom occurs that a decision situation is adequately described by only one model. Mostly decision situations have several aspects that have to be described by different models, for instance a linear program and a queueing model. If these models describe **independent aspects** of the decision situation the dss will have the same architecture as with one model. Only the user-interface and the database serve more models instead of one. Models in a dss are called **independent** if they only need the **exogenous parameters** of decision situation to determine the effects of actions or the actions themselves. They are called dependent if at least one of the models needs a parameter that is computed by another model and that does not belong to the exogenous parameters. We call these parameters **endogenous parameters**. Dependencies between the endogenous parameters may be represented by a directed graph, where each node represents a model and each arc is labeled with the name of an endogenous parameter. If this graph is acyclic there is an ordering of model computations such that each model only needs exogenous parameters or already computed endogenous parameters. However if there is a cycle in the graph there is a more serious dependency between the models. An example of such a situation is described in the next section. Other examples occur in hierarchical planning situations where at the highest level a capacity is optimized using a resource assignment rule from a lower level. However this assignment rule is computed for a given capacity of the resource. In fig. 1 an example is given. The exogenous parameters as well

as the endogenous parameters that are not used in other models are not represented.

Figure 1.

A **consistency requirement** for such a **network of models** is that the parameters used in the network form a **fix-point** for the function formed by the network that maps the set of parameter vectors into itself. In the example the vector $\langle p_1, p_2, p_3, p_4 \rangle$ is a fix-point of the function if : $M_1(p_3) = \langle p_1, p_4 \rangle$, $M_2(p_1) = p_2$ and $M_3(p_2, p_4) = p_3$.

Often there are no analitical properties of this function, such bounds on derivatives know. One only may compute the function value for some parameter vector and each evaluation may be very costly. There are many techniques for the determination of fix-points; however many of them are infeisible for the situation we sketched. In practice the method of successive approximations (for $n=0,1,2,...:q_{n+1}=T(q_n)$ and q_0 is some start vector and T the function formed by the network) often converges. In case the parameter space forms a lattice it can be proved that if the function T is continuous from this parameter space to itself, the convergence is justified by Tarski's fixed point theorem (cf.[Ta55]).

4.Example : dss for a labourpool.

In this section we consider an example of a dss that reflects many of the aspects we mentioned in section 3. For a more detailed description of this dss we refer to [He86].

Consider a group of companies or departments having the same type of work and rather high variation of their daily workload. Our example originates in a harbour where stevedoring companies have highly varying workloads, with almost no correlation between each other. Such companies or departments may consider to establish a pool of worker. So they will cover their workload with own personel and poolworkers.

The pool is a non-profit organisation and therefore it can't take the risk of idle time of workers.

The participating companies take all share of the pool personel for which they are guarantee. The pool is so divided into guaranteed parts. Each day a company may demand its guaranteed part, and it will get it. However if a company wants more workers and other companies don't need their guaranteed part the company can get more workers. If a company does not need its part completely, some workers may be used in other companies. If there are idle workers in some part then the guaranteeing company has to pay for these people.

On the other hand if a company can't cover its workload by pool workers then it has to hire people from outside the pool, which is supposed to be more expensive.

The pool is controlled by a board formed by the participating companies. Periodically the board has to consider the guaranteed parts and the companies may wish to switch personel from their own company to the pool and vice versa. The price of a pool worker per day is determined at the end of a period by dividing all the cost of the pool by the number of used labour days. Daily the companies put their demands to the pool and using a complex algorithm the pool management determines the daily assignments to the companies. The details of this algorithm are not important here, we only note that exaggerating the demands by the companies, in one direction or the other, does not influence the assignment.

For each company we describe the expected daily cost, using the following notations :

Model for one company

w = a random variable with known distribution expressing the daily workload
b = number of own workers, a decision variable
g = number of poolworkers in the guaranteed part, a decision variable
k = daily cost of an own worker
p = price of a poolworker per day
q = price of an external worker per day
t = assignment of poolworkers at some day, a random variable determined by all W's of the companies, B's and G's and an algorithm.

We assume $k < p < q$ because otherwise own workers or poolworkers would never be used.

Further we define :

$v = (w - b)^+$, the daily demand of the company for poolworkers; this is also a random variable, note $x^+ = \max(0, x)$

The exact value of the expected daily cost are

$$k*b + p*g + q* E(v-t)^+ - p*E(g-t)^+$$

where E is the expectation operator induced by the product probability of the daily workload distributions. This formula can be interpreted in the following way. The company has to pay its own workers, and its share in the pool ($k*b + p*g$). However if the demand v is larger than the assignment t it will hire external workers for q per day. On the other hand if the assignment is less than the guaranteed part g the company will get back the price of each poolworker for $g-t$ workers.

If each company would know the demand of other companies they could in theory compute this value. However they don't know these distributions for privacy reasons. Therefore they work with another cost function :

$$k*b + p*g + [p*B + q*(1-B)]*E(v-g)^+ - p*E(g-v)^+$$

where B is the probability of getting a worker from another company's part, if needed. It is clear that the second cost function is not equal to the first one. However it is a good approximation and it has a nice property : it can be optimized

for each company separately if B is known, because the expected values only depend on the distribution of the company itself.

The companies may estimate B from the past. In fact the companies determine their policy, i.e. their determination of the desired b and g using the second cost function. Therefore we use it in the dss as well. There is also another reason.

The exact cost function, given all the workload distributions is very time consuming, since for each vector of b and g values we can only compute the cost by running a simulation program. So each evaluation of a simultaneous decision (i.e. the vector of b and g values) is very time consuming itself. Hence a simultaneous optimization over all companies, for instance minimizing the sum of the expected daily cost is not feasible in an interactive system.

The poolboard will determine for a new period the b and g values of the companies, based on the estimated workload distributions in the following way. Note that these workload distributions are kept secret to the board members. The B values and the price p are computed by a simulation model, that given b and g values, simulates the pool for one or two years. The price p is computed as the salary cost of the workers plus a share in the overhead of the pool divided by the used labour days. The b and g values are determined by minimizing the second cost function per company, given B and p .

Hence we have here a simple model network.

Figure 2.

The model called "Optim" generates decisions while the model "Sim" computes the effects of decisions including the exact expected cost per company. The model consistency requirement provides here as a by-product, an interesting optimality criterion for this multi company game. The vector of b and g values is optimal if for each company the expected cost are minimized under the condition that the intercompany parameters B and p are constant. In practice the convergence of the iteration process to get the fix-point of the model network is rather quick.

The dss can also deal with constraints on b and g values, such as ranges for these values.

5. Architecture of decision support systems.

In this section we describe, using dataflow diagrams, the structure of a dss constructed following the usual OR-approach. Afterwards we charge this structure into a more expert systems architecture.

The term 'architecture' points to structure and to the way constructing an object. We will consider both aspects here. In both approaches one has to analyse the decision situation first. The designer has to determine the exogenous parameters, the domains of the decision variables, i.e. the sets from which the actions may be drawn and the endogenous parameters that the decision maker wants to see to judge a decision.

The endogenous parameters the decision maker wants to see are computed by evaluation functions however in the first stage of design ; only the names and types of these parameters are important. If there are several criteria to generate actions each criterion can often be described by linear combinations of the endogenous parameters and some bounds (cf. section 2).

If for instance these endogenous parameters are $e_1, e_2, \dots, e_n$ then a criterion has the form :

$$\text{maximize } \sum_{j=1}^n \gamma_j e_j \text{ under the condition that}$$

$$\forall i \in \{1, 2, \dots, m\} : \sum_{j=1}^n \alpha_{ij} e_j < \beta_i$$

Note that the endogenous variable may not be chosen freely but are of the form :

$$e_j = f_j(p, a)$$

where f_j is an evaluation function, p a data structure that represents exogenous parameters and a a data structure representing an action. Hence we are in general not dealing here with a linear programming problem. A set of coefficients α, β and γ is called a set of **criterion coefficients**.

A **scenario** is a datastructure consisting of values for the

exogenous parameters, an action, a set of criterion coefficients and values for the endogenous parameters such that the action optimizes the criterion and the endogenous parameter values are the effects of an action. Hence a scenario describes one **instance** of the **decision situation**.

The decision maker will only supply exogenous parameter values and an action or a set of criterion coefficients. The dss will supply the rest.

If the structure of scenarios is determined the designer may perform two tasks in parallel.

One of these tasks is to design a part of the dss we call the **manipulator** (cf. fig. 3). The manipulator consists of a database that logically may be divided into four subdatabases. One containing a set of exogenous parameter values. In each scenario only one set is used. However there may be many sets. Some of them may be derived by some filtering process from a database that monitors the target system. Others may be defined by the decision maker himself or by some external source. The second subdatabase contains actions. Note that actions, as exogenous parameters, may have complex database structures. Each action may be used in a scenario. The third database contains sets of criterion coefficient sets. Finally, the fourth subdatabase contains scenarios.

The manipulator further consists of a database management system to update the four subdatabases and to query the scenario database. The query processor must allow queries in which comparisons over several scenarios are possible.

Figure 3. The manipulator, a box represents a database, a bubble a (software) processor. (1) is coming from a monitoring database , (2) represents the decision maker.

The other task to perform is generation and evaluation of actions. In the OR-approach often models are developed that are specific for the decision situation. When there is more than one model we get the structure represented in fig. 4.

Here we have a number of models all covering some aspects of the decision situation. They can often be divided into two groups : generators and evaluators.

Further there is a processor that takes care of the model interfacing and the iteration of computations to approximate a fix-point for model consistency. This processor is exchanging endogenous parameters between the models and a database for these parameters.

figure 4. Architecture specific for one decision situation.
(cf. the explanations of fig. 3).

Often it is very expensive to construct a generator/evaluator part for one decision situation. What we wish is a system that can easily be adopted to a specific decision situation. For decision situations that can be modelled by linear programming models such systems exist. Such a system may be called a dss-generator. The only modelling activity is to create a matrix generator or to generate such a matrix generator (cf.[Om81]). For other decision situations we also would like to have an architecture in which only the domain specific knowledge has to be given to the system to behave as a dss for that situation. In expert systems constructed from a shell this is possible by only adding facts and rules to the shell. The domain specific knowledge consists of the data already considered by the description of the manipulator, a description of evaluation functions and search rules for a general purpose generator of actions. A sketch of such an architecture is given in fig. 5. There we see three processors. One generator and one evaluator. These processors are used in each specific decision situation. However they have parameters in the form of expressions to specify the evaluation functions and search rules to control the search process of the generator. Of course there are two databases to store these expressions and rules and there is a processor to update these databases. The decision maker will only use the manipulator. The designer of a dss will fill and update the expressions and search rules databases. It is the intention that the expressions and search rule definitions are formulated

in a very high level language.

figure 5. Architecture of a dss-generator.

In the next section we describe a machine that has in fact this architecture.

6. An abstract dss machine

We will present here a definition of a dss for a class of combinatorial problems. We will refer to it as an abstract dss machine.

Some of the applications we are interested in are job-shop scheduling, vehicle routing and ship loading.

Solving problems of this kind is equivalent to searching through a graph of plans, i.e. a graph of job-shop schedules, vehicle routes, ship loads, etc. for a plan satisfying a number of constraints and for which some cost function is optimized.

In practice achieving an optimal plan is seldom attainable, since the problems we are concerned with are known to be NP-complete. However by using good heuristics we can hope for finding a suboptimal plan.

The purpose of our definition of a dss is to facilitate specifying and solving such problems. In other words we aim at reducing the cost of constructing and modifying a dss. At the same time we hope to attain acceptable performance of our dss.

We will sketch here the components which make the specification of a problem and later we will discuss the behaviour of the dss machine during the problem solving.

6.1. Informal presentation of the abstract dss machine.

When specifying a problem like job-shop scheduling we need to define

(i) a target system (e.g. a job-shop).

We will define the target system in a first order language namely in first order predicate calculus with function symbols and equality (cf. [Me64]). We will do so in order to achieve a compact and a precise description of the target system.

We assume the existence of an associated proof system since it is essential to be able to reason about the target system. It is possible that a subset of first order predicate calculus like Horn Clauses with negation as failure and a proof

system based on resolution would be sufficient ([cf. [L184]]). In any case we require a high expressiveness and precision in describing relations holding among the elements of the target system as well as in describing the constraints on plans.

(ii) a graph of plans.

By the graph of plans we mean here a graph where nodes define admissible plans and edges define manipulations upon plans.

In order to apply an manipulation to a plan some constraints upon the plan must hold (they are preconditions to the manipulation) and some constraints must hold on the resulting plan (they are postconditions of the manipulation) (cf. [Ni82,, [Ko79]]).

(iii) a predicate defining a required plan.

We will call it a goal predicate.

(iv) some optimization criterion (like a cost function).

(v) a set of functions which administrate the search process and guide the search. We will call them selection functions.

By composing different selection functions we may change the search strategy in order to suit the search space and our requirements with respect to the quality of the solution, i.e a plan (cf. [Ni82], [Pe84]).

(vi) a recovery function which defines the behaviour of the dss machine in case no plan satisfying the goal predicate is found. One possible option is to deliver the best partial plan constructed.

As it could be expected the dss machine has a memory structure for collecting constructed plans. The memory is divided into two disjoint substructures :

(i) the one which contains the plans that are going to be further transformed. These plans will be called active plans.

(ii) the one which contains the plans that will not be transformed any more. These plans are called non-active plans.

This is so because either all possible transformations for non-active plans have been considered or

- it is not worthwhile to transform these plans any further.

We will discuss now the behaviour of the machine during the problem solving. It is assumed that all the components discussed so far are given.

To start the search for a required plan an initial plan is input to the machine as the only active plan. The memory of non-active plans is empty.

An active plan is selected from the memory of the active plans by the selection function. In the beginning of the search the choice is limited to the initial plan.

The goal predicate is applied now to the selected plan. In case the predicate evaluates to true the plan is output and the machine stops.

Otherwise the machine attempts to transform the selected plan to a set of new plans by applying some manipulation to it as defined by the graph of plans. In case of a job-shop a manipulation could mean adding an operation to the schedule.

Suppose a set of new plans is found. The selected plan is transferred to the non-active plans and the subset of the new plans is added to the memory of the active plans.

Now the machine selects the plan for further transformations by applying a selection function to the active plans.

The search continues until either a required plan is found or the memory structure containing the active plans is empty.

In the latter case the recovery function is applied. One possible option for the recovery function is to select the best plan from the non-active plans as a partial solution to the problem. Since the recovery function is not a fixed element of the dss machine therefore another behaviour can be specified.

In case of a cyclic graph the dss machine can enter a loop, therefore some loop-detection mechanism has to be incorporated. However we do not discuss it here.

6.2 An abstract dss machine - a formal definition of the components.

An abstract dss machine is defined as a 12-tuple :

$\langle D, S, A, M, \text{Goal}, Tm, T, Q, Ra, Rr, Rs, Rm, \rangle$

where

D : a set of target systems.

A target system is defined by a set of formulas of a first order language.

S : a set of elements called plans.

A : a set of elements called manipulations.

M : $(S^*)^2$ - a memory structure.

$\text{Goal} : S * D \rightarrow \{\text{true}, \text{false}\}$

$\text{Goal}(p, d) = \text{"true if } p \text{ is a required plan otherwise false"}$.

$Tm : M * D \rightarrow S$; a constructor function.

$Tm(m, d) = \text{" a plan constructed in the context of the memory } m \text{ and the target system } d \text{"}$.

$Tm(n, d) :=$

if $Rs(m) = \text{nil}$ then $Rr(m)$ else

if $\text{Goal}(Rs(n), d)$ then $Rs(m)$ else

$Tm(Rm(m, \text{NewPlans}(m, d)), d)$

$\text{NewPlans} : S^* * D \rightarrow S^*$

$\text{NewPlans}(m, d) = \text{" a set of new plans constructed from the selected plan } Rs(m) \text{ and the set of selected manipulations } Ra(Q(Rs(m), d)) \text{"}$.

$\text{NewPlans}(m, d) := \{T(Rs(m), a) \mid a \in Ra(Q(Rs(m), d))\}$

$T : S * A \rightarrow S$

$T(\text{old_plan}, a) = \text{"a new plan created by applying a manipulation } a \text{ to the plan old_plan"}$.

T has to be defined with respect to a specific application (e.g. a job-shop scheduling or vehicle routing).

$Q : S * D \rightarrow P(A)$

$Q(s, d) = \text{"a set of manipulations for a given plan } s \text{ with respect to the target system } d"$.

Q has to be defined for a specific application.

$R_a : P(A) \rightarrow P(A)$; a manipulation selection function.

$P(A)$ denotes a power-set of A .

$\forall$ manipulations $P(A)$ $R_a(\text{manipulations})$ manipulations

R_a function selects a subset of manipulations that are going to be applied to a given plan.

$R_r : M \rightarrow S$; so called recovery function.

It defines the behaviour of the dss machine in case no plan satisfying the Goal predicate has been found.

R_r has to be defined for a specific application.

$R_s : M \rightarrow S$

$R_s(m) = \text{"a plan selected from the memory } m"$.

R_s is defined in the context of a specific search strategy as it is illustrated later by examples.

$R_m : M * S^* \rightarrow M$; a memory update function.

$R_m(m, \text{plans}) = \text{"a memory } m \text{ updated by a sequence of plans"}$.

R_m is defined with respect to specific search strategies.

Note:

We are using a higher order functional programming language to define the essential components of the dss machine.

Some conventions concerning the syntax and the semantics of the language are given below. This is not a formal definition of the language however.

For the introduction to functional programming see [cf. G184].

- (i) Function symbols start with uppercase and variables start with lowercase.
- (ii) "fname" ($p_1, \dots, p_2$) := "expression"
A function "fname" with parameters $p_1, \dots, p_2$ is defined by an "expression". The "expression" is made of a function application in a prefix form or it is made of "if then else" expression.
- (iii) A function application in a prefix form is denoted by "fname"($a_1, \dots, a_2$) where $a_1, \dots, a_n$ are arguments of the function.
As opposed to the function definition the function application is never followed by '=' symbol.
- (iv) if then else expression has a form:
If "condition" then "value A" else "value B".
Its meaning is:
if "condition" is true then the result is "value A" otherwise it is "value B".
- (v) Lists:
< > is a list (an empty list)
If x is a list and v some value then Cons(v,x) is a list.
By convention we write
Cons(x_1 , Cons ($x_2, \dots, \text{Cons}(x_n, < >)\dots$)) as $\langle x_1, x_2, \dots, x_n \rangle$.
Cons is called a list constructor and it is a primitive function.
- (vi) Some important functions :

Firstx : $S^* \setminus \{ \langle \rangle \} \rightarrow S$ such that
Firstx (Cons(x,y)) = x

Restx : $S^* \setminus \{ \langle \rangle \} \rightarrow S^*$ such that
Restx (Cons(x,y)) = y

First : $S^* \rightarrow S$

First(list) := if list = $\langle \rangle$ then nil else
Firstx(list)

Rest : $S^* \rightarrow S^*$

Rest(list) := if list = $\langle \rangle$ then $\langle \rangle$ else
Restx(list)

Append : $S^* * S^* \rightarrow S^*$

Append(listA, listB) :=
if listA = $\langle \rangle$ then listB else
Cons(Firstx(listA), Append(Restx(listA), listB))

Delete : $S^* * S \rightarrow S^*$

Delete(list, element) :=
if list = $\langle \rangle$ then $\langle \rangle$ else
if Firstx(list) = element then Delete(Restx(list), element)
else Cons(Firstx(list), Delete(Restx(list), element))

Eval(f, args) = "a result of function f applied to arguments
that are elements of the list args".
Eval is a primitive function.

Select : $S^* * (S \rightarrow \{\text{true}, \text{false}\}) \rightarrow S$

Select(list, predicate) :=
if list = $\langle \rangle$ then nil else
if Eval(predicate, $\langle \text{Firstx(list)} \rangle$) then Firstx(list) else
Select(Restx(list), predicate)

set abstraction function :

$\{x \mid x \in \text{"domain"} \wedge \text{Predicate}(x)\}$

means : a set of elements of the "domain" such that
Predicate(x) evaluates to true. A set is represented as
a sequence.

The set abstraction is a primitive function.

6.3. Specifying the abstract dss machine - search strategies.

We will show now how by giving the appropriate definitions of R_m and R_s functions we can change the search strategies performed by the function T_m . We deal with acyclic graphs only, therefore no loop-detection mechanism is considered here.

We assume that unless stated otherwise

$R_a(\text{manipulations}) := \text{manipulations}$ (the identity function)

$\text{Goal}(s,d) := \text{"a predicate defined over } S \times D\text{"}$

$R_r(m) := \text{nil}$ (the recovery function returns nil)

6.3.1. Depth-first search (dfs).

$R_s(\langle m_n, m_o \rangle) := \text{First}(m_n)$

$R_m(\langle m_n, m_o \rangle, \text{extensions}) :=$
 $\langle \text{Append}(\text{extensions}, \text{Rest}(m_n)),$
 $\text{Cons}(\text{First}(m_n), m_o) \rangle$

where:

m_n : a sequence of active plans,
 m_o : a sequence of non-active plans,
extensions : a sequence of new plans.

6.3.2. Hill-climbing (hc).

$R_s(\langle m_n, m_o \rangle) := \text{First}(m_n)$

$R_m(\langle m_n, m_o \rangle, \text{extensions}) :=$
 $\langle \text{Append}(\text{Order}(\text{extensions}), \text{Rest}(m_n)),$
 $\text{Cons}(\text{First}(m_n), m_o) \rangle$

where:

$\text{Order} : S^* \rightarrow S^*$

$\text{Order}(\text{plans}) = \text{"a permutation of plans such that for all}$
 subsequent elements p_i and p_{i+1} of the permutation,

$p_i > p_{i+1}$, where $>$ is an ordering relation on S .
 We call $\text{Order}(\text{plans})$ an ordered permutation of plans.
 The ordering on S can be constructed on basis of an
 evaluation function $E : S \rightarrow \mathbb{R}$ in particular.

6.3.3 Breadth-first search.

```
Rs(<mn, mo>) := First(mn)

Rm(<mn, mo>, extensions) :=
  <Append (Rest(mn), extensions),
    Cons(First(mn), mo)>
```

6.3.4 Best-first search (bestfs).

```
Rs (<mn, mo>) := First(mn)

Rm (<mn, mo>, extensions :=
  <Order(Append (extensions, Rest(mn))),
    Cons(First(mn), mo)>
```

6.3.5. A variant of heuristic search.

```
Rs(<mn, mo>):= Select(mn, Rs_select)

Ra (manipulations) = Select(manipulations, Ra_select)

Rm (<mn, mo, extensions) :=
  <Append (extensions, Delete (mn, Rs (mn))),
    Cons ( Rs (mn), mo)>
```

where:

```
Rselect : S --> {true, false}
Ra_select : A --> {true, false}
```

$Rselect(m_n, plan) = \text{"true if plan is an element of } m_n$
 and it satisfies some selection criterium otherwise false".

```
Ra_select(manipulations, manipulation) = "true if manipulation is  
    an element of manipulations and it satisfies some selection  
    criterion otherwise false".
```

7. A job-shop scheduler.

A job-shop is a production system consisting of a set of machines capable of performing different functions.

A job-shop processes jobs. A job is defined by a set of partially ordered tasks. A task is defined as an activity performed on a machine of some type for a specific period of time.

Since a number of tasks is present in the job-shop possibly competing for the machines we have to solve a problem of assigning the tasks to the machines. In other words we have to construct a job-shop schedule that is an assignment of tasks to the machines with respect to some constraints.

Namely a task has to be assigned to a machine of a required type and for a required period of time (such assignment is called operation). The ordering of tasks as defined by the schedule has to preserve the partial ordering of tasks within a job. A machine cannot be used by two tasks at the same time. Usually there are some time limits (deadlines) imposed on the jobs.

Our goal is to define a job-shop scheduler as an abstract dss machine. We will do so by specifying elements of the 12-tuple discussed earlier.

We will use a method of forward scheduling that can be described as follows. The scheduling starts at some moment of time t . We construct operations with their begin time equal to t until we cannot continue any further. This may happen when there are no required machines free or all the tasks whose predecessors were finished before the time t have been scheduled.

Now we look for a future moment of time p when some of the scheduled tasks is finished. If p is found then we move the scheduling time forward to p and again we try to construct a new set of operations with their begin time equal to p .

We continue the scheduling process until all the tasks are scheduled or we reach the planning horizon.

The target system i.e. a job-shop will be defined by a set of formulas of first order predicate calculus with function symbols and equality. The symbols used in the definition of the target system should not be confused with the symbols used in the functional language already discussed. The conventions we follow

are :

- (i) $\wedge, \vee, \rightarrow, \leftrightarrow, \neg$ are logical connectives,
 $\forall, \exists$ are quantifiers,
 $=$ is the equality symbol,
 $<$ is the symbol of usual ordering relation on N ,
- (ii) predicate symbols start with uppercase,
- (iii) variables start with lowercase; all variables are quantified,
- (iv) function symbols start with lowercase or they are $'*$ or $'+$.

Other components of the job-shop scheduler are given below.

S : A set of schedules.

A schedule is defined by a set of operations (we will call it plan) and by a scheduling time greater or equal to the begin time of the most recent operation in the plan. An operation defines an assignement of a task to a machine from the begin time till the end time. Schedules are represented by terms: `schedule(time, plan)` and operations are represented by terms: `operation(task, job, machine, begin, end)`.

A : A set of manipulations.

There are two classes of manipulations :
manipulations represented by terms `move(timex)`
and manipulations represented by terms
`operation (task, job, machine, begin, end)`.
`move(timex)` defines a request to move forward the scheduling time to `timex`.
`operation(task, job, machine, begin, end)` defines a request to assign the task to the machine from the begin time till the end time.

Q : $S * D \rightarrow P(A)$

$Q(s, d) := \{a \mid l \cup d \vdash \text{Manipulation}(s, a)\}$
 $Q(s, d) =$ "a set of all manipulations such that $\text{Manipulation}(s, a)$ can be deduced from a target system d .
 Manipulation is a predicate symbol which appear in d .
 $\vdash$ is a derivability relation for a theory made of d and logical axioms l of the associated proof system.
We do not discuss here how such a system performs

deductions.

d - the set of axioms describing the target system.

Each axiom is defined informally first and then formally.

Manipulation (schedule(current_time, plan),
operation (task, job, machine, begin, end))

iff

"the task belongs to the job and the machine suits the task, and the machine is idle within a required period of time from the begin to the end, and all the predecessors of the task are finished before the current_time".

$\forall$ current_time, plan, task, job, machine, begin, end
(Manipulation (schedule(current_time, plan),
operation (task, job, machine, begin, end)))

<-->

Is_task(task, job, machine_type, duration) $\wedge$
Is_machine (machine, machine_type, speed) $\wedge$
Finished_predecessors (current_time, task, job, plan) $\wedge$
Idle_machine (machine, begin, end, plan) $\wedge$
begin = current_time $\wedge$
end = (current_time + speed * duration)) $\wedge$

Manipulation (schedule (timex, plan), move(timey))

iff

"there is no operation(task, job, machine, begin, end) which can be created in the context of the current schedule as defined by schedule(timex, plan) and timey is the next future moment of time when one of the tasks is finished".

$\forall$ timex, timey, plan

(Manipulation (schedule (timex, plan), move(timey))

<-->

($\neg \exists$ task, job, machine, begin, end
Manipulation (schedule (timex, plan),
operation (task, job, machine, begin, end)) $\wedge$
Nextevent (timex, timey, plan)))

```

∀ task, job, machine_type, duration
Is_task(task, job, machine_type, duration)
    iff
    "the task belongs to the job and requires a machine of the
    type machine_type for the nominal time: duration".
    It is defined by a set of ground atoms.

∀ machine, machine_type, machine_speed
Is_machine(machine, machine_type, speed)
    iff
    "the machine is of the type machine_type and has the speed
    machine_speed".
    It is defined by a set of ground atoms.

Nextevent (timex, timey, plan)
    iff
    "timey > timex and timey defines the first moment of time
    when some machine is released by a task".

∀ timex, timey, plan
(Nextevent (timex, timey, plan)
<-->
∃ task, machine, begin, job
(operation(task, job, machine, begin, timey) ∈ plan ∧
timex < timey ∧
¬∃ t, m, j, b, e
(operation(t, j, m, b, e) ∈ plan ∧
b > timex ∧ b < timey )))

Finished_predecessors (time, task, job, plan)
    iff
    "all the predecessors of the task within the job are finished
    before the time with respect to the plan".

∀ time, task, job, plan
(Finished-predecessors (time, task, job, plan)
<-->
∀ taskx
(Predessor (job, taskx, task) -->
∃ beginx, beginy, machine, endx

```

(operation (taskx, job, machine, beginx, endx) $\in$ plan $\wedge$
endx < time)))

$\forall$ taskx, tasky, job

Predecessor (job, taskx, tasky)

iff

"the taskx is a predecessor of the tasky within the job".

It is defined by a set of ground atoms.

Idle_machine(machine, begin, end, plan)

iff

"the machine is idle (not used by any task) within the
period from the begin to the end in the context of the plan".

$\forall$ machine, begin, end, plan

(Idle_machine(machine, begin, end, plan)

$\leftrightarrow$

$\forall$ task, job, beginx, endx

(operation(task, job, machine, beginx, endx) $\in$ plan

$\rightarrow$ Overlap(begin, end, beginx, endx)))

Overlap (a, b, c, d)

iff

"two periods of time from a to b and from c to d overlap".

$\forall$ a, b, c, d

(Overlap (a, b, c, d) $\leftrightarrow$

((b $\geq$ c $\wedge$ a $\leq$ c) $\vee$ (a $\leq$ d $\wedge$ d $\leq$ b)) $\wedge$

a < b $\wedge$ d < c).

We will give now definitions of the other functions required by
the dss machine.

$R_s : M \rightarrow S$; a selector of the active plans.

$R_s (\langle n_n, m_0 \rangle) := \text{First}(n_n)$

$R_a : P(A) \rightarrow P(A)$; a selector of the manipulations applicable
to a plan.

$R_a(\text{manipulations}) := \text{manipulations}$ (we take identity function)

We will define now the function T by means of two equations.

$T: S \times A \rightarrow S$

"a plan is extended by an operation".

$T(\text{schedule}(\text{time}, \text{plan}),$
 $\text{operation}(\text{task}, \text{job}, \text{machine}, \text{begin}, \text{end})) :=$
 $\text{schedule}(\text{time}, \text{Cons}(\text{operation}(\text{task}, \text{job}, \text{machine}, \text{begin}, \text{end}),$
 $\text{plan})).$

"a scheduling time is move forward to a newtime".

$T(\text{schedule}(\text{time}, \text{plan}), \text{move}(\text{newtime})) :=$
 $\text{schedule}(\text{newtime}, \text{plan})$

$\text{Goal} : S \times D \rightarrow \{\text{true}, \text{false}\}$

$\text{Goal}(\text{schedule}(\text{time}, \text{plan}), d) :=$
 $\text{Timelimit}(\text{time}) \vee \text{All_tasks_scheduled}(\text{plan}, d)$

$\text{Timelimit} : \mathbb{N} \rightarrow \{\text{true}, \text{false}\}$

$\text{Timelimit}(\text{time}) := \text{"time is greater than a planning horizon"}$.

$\text{All_task_scheduled}(\text{plan}, d) := \text{"all tasks as defined by the}$
 $\text{Is_task predicate have been scheduled"}$.

We define now the memory update function R_m .

$R_m : M \times S^* \rightarrow M$

$R_m(\langle m_n, m_o \rangle, \text{extensions}) :=$
 $\langle \text{order}(\text{Append}(\text{extensions}, \text{Rest}(m_n))),$

Cons (First (m_n), m_o).

where

Order : $S^* \rightarrow S^*$

Order (schedules) := "an ordered permutation of schedules".

An ordering relation has to be defined on
schedules.

We can express here our preferences with
respect to schedules.

As a result of our definitions we get the best-first search
strategy for the job-shop scheduler.

Rr : $M \rightarrow S$; the recovery function

Rr($\langle m_n, m_o \rangle$) = "selects a plan from either m_n or m_o in case
no plan satisfying 'Goal' predicate was found.

Rr($\langle m_n, m_o \rangle$) := nil (we do not want any partial schedule)

Conclusions.

We have given a specification of the components of an abstract dss machine. We hope that the specification shows in a clear way the structure of the system with respect to the target system, the graph of plans and the search strategies.

It seems possible to begin the construction of the dss by neglecting the problems of selection of search strategies and choosing a standard one, and emphasizing the correct definition of the target system together with the graph of plans.

Later the efficiency of the dss can be tuned by supplying suitable selection functions and thus creating a more refined search strategy without the need to modify the whole system.

References

- [Be83] Bennet, J.L. "Building decision support-systems", Reading, Mass., Addison-Wesley, 1983.
- [Gl84] Glaser H., Hankin C., Till D. "Principles of Functional Programming", Prentice-Hall, 1984.
- [He86] van Hee K., Huitink B., Leegwater D.K. "Portplan, decision support system for port terminals", Eindhoven University of Technology Notes, 1986.
- [Ja86] Jackson P. "Introduction to expert systems", Addison-Wesley, 1986.
- [Ko79] Kowalski R. "Logic for Problem Solving", North-Holland, 1979.
- [Ll84] Lloyd J.W. "Foundations of Logic Programming", Springer-Verlag, 1984.
- [Me64] Mendelson E. "Introduction to Mathematical Logic", Van Nostrand, Princeton 1964.
- [Ni82] Nilsson N. "Principles of Artificial Intelligence", Springer-Verlag, 1982.
- [Om81] Omni Linear Programming System, User Reference Manual, Haverley Systems Inc., Nov. 1981.
- [Pe84] Pearl J. "Heuristics: intelligent search strategies for computer problem solving", Addison-Wesley, 1984.
- [So84] Sowa J.F "Conceptual structures: information processing in mind and machine", Addison-Wesley, 1984.
- [Ta55] Tarski A. "A lattice-theoretical fix-point theorem and its applications", Pacific Journal of Mathematics, pp.205-309, 1955.