Schedule management : an object oriented approach

Eindhoven University of Technology

Department of Mathematics and Computing Science

Schedule Management: an Object Oriented Approach

by

G. Wolf

Computing Science Note 91/22
Eindhoven, September 1991

COMPUTING SCIENCE NOTES

This is a series of notes of the Computing Science Section of the Department of Mathematics and Computing Science Eindhoven University of Technology.
Since many of these notes are preliminary versions or may be published elsewhere, they have a limited distribution only and are not for review.
Copies of these notes are available from the author.

Copies can be ordered from:
Mrs. F. van Neerven
Eindhoven University of Technology
Department of Mathematics and Computing Science
P.O. Box 513
5600 MB EINDHOVEN
The Netherlands
ISSN 0926-4515

editors: prof.dr.M.Rem
prof.dr.K.M.van Hee.

Schedule Management: an Object Oriented Approach

G. Wolf (1.9.91)

Department of Mathematics and Computer Science,
Eindhoven University of Technology,
P.O. Box 513, 5600 MB Eindhoven, The Netherlands

Abstract. In this paper we consider resource-constrained time-dependent scheduling systems, i.e. decision support systems with time as the important planning component. Instead of dealing with optimalization aspects of the planning problem, we concentrate on schedule management, i.e. stepwise planning with respect to primitive functions, like handling single decisions or constraints.

The design of these systems is based on a mathematical model, giving a formal characterization of a class of scheduling problems and allowing generic descriptions of scheduling objects like processors, operations, decisions and constraints. The model is applied to various scheduling problems, like resource constrained project scheduling, car routing, the construction of time tables both for schools and for nursery in hospitals.

An object oriented implementation of the model, based on the natural hierarchy of scheduling problems, turns out to lead to a clear separation between the generic and the domain specific components of the schedule manager, minimizing redundant code and resulting in software with a high degree of maintainability.

Keywords. Decision Support System, Scheduling, Object Oriented Design.

1. Introduction

Software for decision support systems is very sensitive to change requests. A minor change in the underlying model or in the limiting conditions can make it necessary to redesign the whole system. There are hardly reusable components. The claim for a more flexible DSS-design is uttered by many authors [2,6,10,20].

The reason for this inflexibility can be found among others in the fact, that conventional DSS-systems are dominated by the algorithmic aspect. Although it is stated, that the software components of a DSS should relate to both information management and optimalization [3,6,16], more attention is paid to the latter aspect, especially in the first design phase. Mostly mathematicians working in the field of operations research were involved in the software development. In spite of the mathematical relevance of algorithmic aspects, the software dealing with it plays an inferior role related to other software components like the management of decisions and the user interface.

In view of the fact, that software maintainability is our highest objective in this paper, the following requirements can be formulated:

- the software should contain reusable modules, that can be applied to a wider class of planning problems;
- the software architecture should allow a clear separation between the generic components and the domain specific components of the software;
- the software should be a platform with respect to the implementation for both generic algorithms (simulated annealing, genetic algorithms, etc.) [1] and powerful domain specific optimalization procedures;
- the software should be flexible with respect to changes in the (domain specific) model and it should be easy to add or to change constraints.

Although several attempts have been made to increase flexibility of optimalization components by using general search methods applicable in a wide range of planning situations [10], we concentrate in this paper on data manipulation aspects, particularly oriented towards scheduling problems. The basic system functionality is *schedule management.* Primitive functions should be provided to support *stepwise planning* [15]. The user should have among others the possibility to add (and delete) scheduling objects like "decisions" and "constraints"; the system should calculate the consequences of user actions and check feasibility. Optimalization is not relevant in this context.

Scheduling is the allocation of resources over time to perform certain tasks [5]. Scheduling problems can be formulated as sequencing problems with time as the important planning component. Moreover, we require that all feasibility constraints and schedule evaluation criteria can be expressed as measures of time. The class of these scheduling problems is rather extensive and does include job-shop planning, car routing, school time tables and time tables for nursery in hospitals. We will give a formal definition of this class. Although several scheduling models exist [5,9,13], which are suitable to formulate optimalization problems, they are not flexible enough for our purposes.

Because we need the flexibility of generic models as well as the power of problem specific models, modelling on more abstraction levels is necessary. This implies a software architecture consisting of *generic* and *domain specific* modules [20]. A specific

schedule manager is built up by linking the generic modules with the relevant group of domain specific modules.

Because of our high requirements for software maintainability, system implementation is based on an *object oriented* design [17,19]. The use of object oriented techniques for DSS-software has been up to now rather limited [8,12]. There are however many reasons, that plead for their application:

- the concept of *encapsulation*, to specify objects both by their attributes and by their methods, is a powerful mechanism to represent scheduling objects like processors, operations, decisions and constraints;
- the concept of *inheritance* (and *abstract datatyping*), to formalize IS-A relationships between classes and subclasses, is necessary to represent the natural hierarchy of scheduling problems and scheduling objects to get a clear separation between the generic and the domain specific components;
- *polymorphism*, i.e. the possibility to use object methods in different subclasses under the same name, but with specific functionalities, will enlarge the flexibility of the software.

2. Notations

We will follow some notational conventions:

$A \approx B$	: set A and set B are equivalent (of equal power);
$[A \rightarrow B]$	: the set of functions from A to B;
A^*	: the set of strings of elements from set A;
$\mathcal{P}(A)$	: the powerset of A (set of all subsets of A);
$\bigcup(A(i) \mid P(i))$	: the union of all sets $A(i)$, for which $P(i)$ holds;
$\sum(a(i) \mid P(i))$	: the sum of all $a(i)$, for which $P(i)$ holds.;

Let $Time$ be a finite interval on the real axis. With $B(Time)$ we denote the set algebra generated by subintervals of $Time$.

Let A be an entitytype with attributes $A_1, ..., A_n$, i.e. $A \approx A_1 \times ... \times A_n$, and let $t \in A$ be an entity of that type. With $A_i(t)$ we denote the projection of t to A_i.

3. A basic general model

We restrict ourselves to resource-constrained time-dependent planning problems with the property, that schedules can be represented as Gantt charts. This problem class is very wide and includes the general jobshop problem, car routing and the construction of time tables both for schools and for nursery in hospitals.

Existing models for scheduling problems are either too specific with regard to constraints [5,9,13] or too general to be powerful enough to characterize the time aspects of scheduling problems properly [10,14]. For this reason we present in this section a basic general model to characterize our problem class; extensions and problem specific interpretations follow later. Our object definitions (see section 7) will be based on this model.

3.1. Scheduling problems

A class of *scheduling problems* can be characterized by the following tuple

$$(P, O, D, pr, op, S, f, g, t_{max}) \quad \text{with}$$

- P, the set of all possible *processors* (or resources);
- O, the set of all possible *operations* (or tasks) to be assigned to processors;
- D, the set of all possible *decisions*; we have $D = D_+ \cup D_0$, with D_+ being the set of decisions, that assign an operation to a processor (*real* decisions) and D_0 being the set of decisions, that put a processor in the waiting state (*idle* decisions);
- the function $pr : D \rightarrow P$ assigning the processor, involved in a decision;
- the function $op : D_+ \rightarrow O$ assigning the operation, involved in a real decision;
- S, the set of all possible *states*, that a processor can have with the function $s_0 : P \rightarrow S$ assigning the initial states;
- the function $f : D \times S \rightarrow S$ describing a *state transition*, being the effect of a decision with respect to a processor;
- the function $g : D \times S \rightarrow R_+$ to determine the *duration*, that a certain decision will be active;
- t_{max}, a positive real number, defining the *scheduling interval* $Time = (0, t_{max}]$.

3.2. Schedules

Scheduling problems are distinguished from other planning problems by the aspect of time. All time aspects should be handled in the generic part of the system.

Let p_i be a processor from a finite subset of P and let $\{d_{ij}\}_{j=1,n_i}$ be a finite sequence of decisions with respect to p_i. The decision $d_{i,j}$ is effectuated at the time $t_{i,j}$ and is active till $t_{i,j+1}$, the time, that the next decision with respect to p_i will be effectuated. We call $t_{i,j}$ a *decision-point*. Let $s_{i,j}$ be the state of processor p_i after decision $d_{i,j}$ $(j > 0)$. Then we have the following equations

$$s_{i,0} = s_0(p_i)$$

$$s_{i,j} = f(s_{i,j-1}, d_{i,j}) \quad (j > 0)$$

$$t_{i,1} = 0$$

$$t_{i,j} = t_{i,j-1} + g(s_{i,j-2}, d_{i,j-1}) \quad (j > 1)\,.$$

We call $s_{\text{fin}}(p_i) = s_{i,n_i}$ the *final state* of processor p_i, which is the state after the last decision.

We can define a *schedule* as a tuple (P_S, O_S, F_S) with

- P_S a finite subset of P,
- O_S a finite subset of O,
- a function $F_S : P_S \rightarrow D^*$ assigning a finite sequence of decisions to every processor of P_S,

$$F_S : p_i \mapsto \{d_{i,j}\}_{j=1,n_i} \quad (p_i \in P_S;\ d_{i,j} \in D) \qquad \text{with}$$
$$pr(d_{i,j}) = p_i, \quad (op(d_{i,j}) \in O_S \text{ or } d_{i,j} \in D_0) \qquad \text{and}$$
$$\sum (g(d_{ij}, s_{ij}) \mid 0 < j \leq n_i) \leq t_{max} \ .$$

As a consequence of this definition only one operation at a time can be allocated to a processor, whereas more processors can be assigned to the same operation simultaneously.

3.3. Constraints

Constraints are used to define the *feasibility* of schedules. It is important to have the flexibility to add and to change constraints dynamically. To realize this we should represent planning knowledge and especially constraints declaratively in a domain independent way. Several ideas and concepts going in this direction, can be found in the literature [7,11,12,20].

Although we assume a certain homogeneity with respect to scheduling objects like resources, operations and decisions, this is not the case with constraints. On the domain specific level various subclasses (types) of constraints may be defined (deadline constraints, precedence constraints, etc.). It is therefore important to distinguish which properties are adherent to constraint classes and which to constraint instances.

We distinguish *hard* and *weak* constraints. The feasibility of a schedule with respect to hard constraints is guaranteed by the schedule manager, where as violations of weak constraints are only signalized to the user, who is ultimately responsible for feasibility. The hardness of a constraint is specified on instance level.

We require, that constraint violations can be expressed by time intervals, denoting the period, when the constraint is violated or when decisions should be effectuated to undo the constraint violation. This is just a generalization of the convention to specify constraints by Boolean functions. This is done for the following reasons:

- the time aspect of the constraint violation is more emphasized, which is rather important for scheduling problems;
- we have a measure of infeasibility with respect to constraints, which is important for weak constraints;

- a wide class of schedule quality measures can be formulated as simple expressions of critical regions, belonging to the same class of constraints (make span, earliness and tardiness for the job-shop problem, etc.).

Now we come to a formal definition of constraints. Let Sch be the set of all schedules with respect to a certain problem class. We define a *constraint-type* c by its *specific domain* and its *evaluator*, i.e. a pair $(Z_c, eval_c)$ with Z_c a set and $eval_c$ a (penalty) function with

$$eval_c \ : \ Sch \times Z_c \rightarrow B(Time) \ .$$

A *constraint instance* of type c can be defined as a pair (z_c, hd) with $z_c \in Z_c$ and hd, being a Boolean to specify the hardness of a constraint instance.

Evaluation of the penalty function $eval_c$ for a particular constraint instance and schedule will produce a subset of the scheduling interval $Time$ to indicate the time period, when the constraint is violated or when decisions should be effectuated to undo the constraint violation. This time period, which we call the *critical region*, can be represented as a finite set of disjunct intervals. The critical region can be empty in the case of feasibility or contain just the whole time interval as extreme cases.

In the Section 4 this definition will be illustrated by a variety of examples.

3.4. Quality measures

Beside constraints, that define the feasibility of schedules, there are quality measures for comparison of schedules. These criteria are often defined as penalty functions, mapping a schedule to a real value [13]. It makes sense to couple quality measures to constraints or sets of constraints by measuring the extent of violation with respect to these constraints. For the most common quality measures (make span, total or weighted tardiness, etc.) corresponding constraints can be defined.

The relation between constraints and quality measures will be formalized by the following definition. A *quality measure* can be defined by a pair $(Clist, agg)$ with $Clist$ being a finite list of constraints and agg being a real (aggregate) function defined on the set of strings of real numbers

$$agg \ : \ R^* \rightarrow R \ .$$

Examples of such aggregate functions are $sum(X)$, $max(X)$, $variance(X)$, etc. with X a sequence of reals.

The constraint list mentioned above will contain often all constraint instances of a certain type, but this is not necessary.

Given a particular schedule a *quality value* q with respect to a quality measure $(Clist, agg)$ can be calculated by applying the specified aggregate function to the list of the lengths of the critical time regions, defined by the constraint evaluation functions corresponding to the specified list of constraints (see fig. 3.1); the length of an element of $B(R)$ is defined by its Lebesque measure μ.

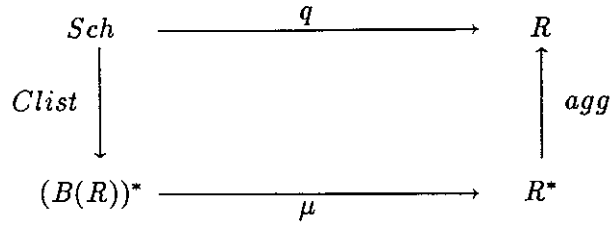

Fig. 3.1. Calculation of the quality value of a schedule.

For example we can define the make span of a schedule for the job-shop problem by (virtual) deadline constraints (with deadlines equal to zero) and the aggregate function giving the maximum of a string of real numbers.

3.5. Extended schedules

Let $(P, O, D, pr, op, S, f, g, t_{max})$ be a class of scheduling problems and let C be the set of all possible constraints for this class. We define an extended schedule by a tuple $(P_s, O_s, F_s, C_s, Q_s)$ with (P_s, O_s, F_s) being a schedule, C_S a subset of C and Q_s a set of quality measures with $c \in C_S$ for every constraint instance c being involved in Q_s (i.e. c an element of some $Clist$ with $(Clist, agg) \in Q_s$).

3.6. Expressions

In this section we assume a fixed given schedule (P_s, O_s, F_s). The set D_S of decisions involved is determined by

$$D_s = \left\{ d_{i,j} \mid d_{i,j} = (F_S(p_i))_j,\ p_i \in P_S,\ 0 < j \leq n_i \right\} .$$

We can now introduce some expressions, that are needed in the following paragraphs:

- the start time of decision $d_{i,j}$:

$$start(d_{i,j}) = \sum (g(s_{i,k-1},\ d_{i,k}) \mid 0 < k < j) \quad (j > 1)$$
$$start(d_{i,1}) = 0$$

- the end time of decision $d_{i,j}$:

$$end(d_{i,j}) = \begin{cases} start(d_{i,j}) + g(s_{i,j-1}, d_{i,j}) \\ start(d_{i,j+1}) \quad (j < n_i) \end{cases}$$

- the activity interval of decision $d_{i,j}$:

$$interval(d_{i,j}) = (start(d_{i,j}), end(d_{i,j})]$$

- the time period, when processor p executes operation o:

$$pr\text{-}op\text{-}set(p, o) = \bigcup (interval(d_{i,j}) \mid pr(d_{i,j}) = p,\ op(d_{i,j}) = o,\ d_{i,j} \in D_S)$$

– the executing period of operation o:

$$op\text{-}exec\text{-}set(o) = \bigcup(pr\text{-}op\text{-}set(p_i, o) \mid p_i \in P_S)$$

– the start time of operation o:

$$start\text{-}op(o) = \begin{cases} min(start(d_{i,j}) \mid op(d_{i,j}) = o,\ d_{i,j} \in D_S) & \text{if } op\text{-}exec\text{-}set(o) \neq \emptyset \\ t_{max} & \text{otherwise} \end{cases}$$

– the end time of operation o:

$$end\text{-}op(o) = \begin{cases} max(end(d_{i,j}) \mid op(d_{i,j}) = o,\ d_{i,j} \in D_S) & \text{if } op\text{-}exec\text{-}set(o) \neq \emptyset \\ t_{max} & \text{otherwise} \end{cases}$$

– the finishing time of procesor p:

$$end\text{-}pr(p) = \sum(g(d_{i,j}, s_{i,j-1}) \mid pr(d_{i,j}) = p,\ d_{i,j} \in D_S)$$

– the allocation of processor p to operation o at time t:

$$busy(p, o, t) = \begin{cases} \top & \text{if } t \in pr\text{-}op\text{-}set(p, o) \\ \bot & \text{otherwise} \end{cases}$$

– the cummulative activity time of decision $d_{i,j}$ at time t:

$$dur(d_{i,j}, t) = \begin{cases} end(d_{i,j}) - start(d_{i,j}) & \text{if } t \geq end(d_{i,j}) \\ t - start(d_{i,j}) & \text{if } start(d_{i,j}) < t < end(d_{i,j}) \\ 0 & \text{otherwise} \end{cases}$$

– the cummulative allocation time of processor p to operation o at time t:

$$pr\text{-}op\text{-}dur(p, o, t) = \sum(dur(d_{i,j}, t) \mid pr(d_{i,j}) = p,\ op(d_{i,j}) = o,\ d_{i,j} \in D_S)$$

4. Domain specific interpretations of the generic model

In this section we give domain specific interpretations of the model. The basic model may be extended with domain specific parameters. We specify attributes of the relevant entities and give specifications of the relevant functions. Constraints are defined by their specific domain and their evaluators (see section 3.3).

4.1. Job-shop

Problem description:

We consider general job-shop scheduling with job splitting and preemption. Operations (tasks) may be allocated several times to processors (resources). An operation has a size, i.e. the total resource capacity necessary to complete the task. Processors can execute operations with different speeds. To execute operations certain abilities of processors are required. The sequence of task execution is restricted by precedence constraints. Deadline and release time constraints are related to the finishing and starting times of operations. Processors are only at certain time windows available.

Domain specific model parameters:

A – a set of abilities.

Processors:

id : identifier
abilset : set of abilities
speed : executing velocity

$$P \approx N \times \mathcal{P}(A) \times R_+$$

Operations:

id : identifier
req-abilset : set of required abilities
size : processing capacity

$$O \approx N \times \mathcal{P}(A) \times R_+$$

Real decisions:

pr : processor reference
op : operation reference
time : duration

$$D_+ \approx P \times O \times R_+$$

Idle decisions:

pr : processor reference
$wait$: waiting time

$D_0 \approx P \times R_+$

States (of a processor):

$execime$: the total executing time of a processor
(till a decision point)

$S \approx R_+$

Initial state (of a processor):

$s_0(p) = 0 \quad \forall p \in P$

State transition:

$$f(state, dec) = \begin{cases} execime(state) + time(dec) & \text{if } dec \in D_+ \\ execime(state) & \text{if } dec \in D_0 \end{cases}$$

Duration of decision:

$$g(state, dec) = \begin{cases} time(dec) & \text{if } dec \in D_+ \\ wait(dec) & \text{if } dec \in D_0 \end{cases}$$

Constraints:

- release time constraints: $Z_c = O \times R_+$

$eval_c(sch, o, rl) = \{t \in Time \mid t < rl\} \cap op\text{-}exec\text{-}set(o)$

- deadline constraints: $Z_c = O \times R_+$

$eval_c(sch, o, dl) = \{t \in Time \mid t > dl\} \cap op\text{-}exec\text{-}set(o)$

- tasks should be completed: $Z_c = O$

let $compl(o)$ be the completed part of an operation:
$compl(o) = \sum(pr\text{-}op\text{-}dur(p_i, o, t_{max}) * speed(p_i) \mid p_i \in P_S)$
and let $\alpha = t_{\max} * compl(o)/size(o)$

$$eval_c(sch, o) = \begin{cases} \emptyset & \text{if } compl(o) \geq size(o) \\ (\alpha, t_{max}] & \text{otherwise} \end{cases}$$

– avoiding superfluous task execution: $Z_c = O$

$$eval_c(sch, o) = op\text{-}exec\text{-}set(o) \cap \{t \in Time \mid \sum(pr\text{-}op\text{-}dur(p_i, o, t) * speed(p_i) > size(o) \mid p_i \in P_S)\}$$

– required processor abilities for task execution: $Z_c = P \times O$

$$eval_c(sch, p, o) := \begin{cases} \emptyset & \text{if } abilset(p) \cap req\text{-}abilset(o) \neq \emptyset \\ pr\text{-}op\text{-}set(p, o) & \text{otherwise} \end{cases}$$

– generalized precedence constraints: $Z_c = O \times O \times R_+$

$$eval_c(sch, o_1, o_2, waiting) := \{t \in Time \mid t < end\text{-}op(o_1) + waiting\} \cap \{t \in Time \mid t > start\text{-}op(o_2)\} \text{ with } o_1 \neq o_2$$

– processor unavailability: $Z_c = P \times B(Time)$

$$eval_c(sch, p, period) := period \cap \left(\bigcup(pr\text{-}op\text{-}set(p, o_k) \mid o_k \in O_S)\right)$$

4.2. Car routing

Problem description:

Cargo (operations) has to be transported by trucks (processors) from one location to another. Transport time is dependent on the distance of the locations, the speed of a truck and the loading resp. unloading time. The amount of cargo that is transported simultaneously is restricted by a maximum volume, which is truck specific. Starting points may be different. Loading or unloading is only possible in specific time windows.

Domain specific model parameters:

L – a set of locations.
$dist : L \times L \rightarrow R_+$ a distance function.

Processors:

id	: car-identifier
start-loc	: start location
speed	: speed of the truck
maxvolume	: maximal total volume

$$P \approx N \times L \times R_+ \times R_+$$

Operations:

id : cargo-identifier
from-loc : place of depart
to-loc : destination
load-time : time to load or to unload
volume : volume of the cargo

$$O \approx N \times L \times L \times R_+ \times R_+$$

Real decisions:

pr : processor reference
op : operation reference
loading : loading or unloading

$$D_+ \approx P \times O \times \{\top, \bot\}$$

Idle decisions:

pr : processor reference
wait : waiting time

$$D_0 \approx P \times R_+$$

States (of a processor):

loc : the location of the truck
vol : the total volume of the cargo

$$S \approx L \times R_+$$

Initial state (of a processor):

$$s_0(p) = (\textit{start-loc}(p), 0) \quad \forall p \in P$$

State transition:

$$f(state, dec) = \begin{cases} (\textit{from-loc}(op(dec)),\ vol(state) + volume(op(dec))), \\ \qquad\qquad \text{if } dec \in D_+,\ loading(dec) = \top \\ (\textit{to-loc}(op(dec)),\ vol(state) - volume(op(dec))), \\ \qquad\qquad \text{if } dec \in D_+,\ loading(dec) = \bot \\ state \qquad\qquad \text{if } dec \in D_0 \end{cases}$$

Duration of decision:

$$g(state, dec) = \begin{cases} dist(loc(state), from\text{-}loc(op(dec))) * speed(pr(dec)) \\ \quad + load\text{-}time(\text{op(dec)})) & \text{if } dec \in D_+,\ loading(dec) = \top \\ dist(loc(state), to\text{-}loc(op(dec))) * speed(pr(dec)) \\ \quad + load\text{-}time(\text{op(dec)})) & \text{if } dec \in D_+,\ loading(dec) = \bot \\ wait(dec) & \text{if } dec \in D_0 \end{cases}$$

Constraints:

- an operation is executed by one processor only: $Z_c = O$

$$eval_c(sch, o) = \begin{cases} op\text{-}exec\text{-}set(o) & \text{if } \#\{i \mid pr\text{-}op\text{-}set(p_i, o) \neq \emptyset\} > 1 \\ \emptyset & \text{otherwise} \end{cases}$$

- operations should be completed: $Z_c = O$

$$eval_c(sch, o) = \begin{cases} \emptyset & \text{if } op\text{-}exec\text{-}set(o) \neq \emptyset \\ (0, t_{max}] & \text{otherwise} \end{cases}$$

- release time constraints: see job-shop

- deadline constraints: see job-shop

- loading/unloading is not allowed in certain time periods: $Z_c = L \times B(Time)$

$$eval_c(sch, loc, period) = period \cap \Big(\bigcup (end(d_{i,j}) - loadtime(op(d_{i,j})),\ end(d_{i,j})] \mid d_{i,j} \in D_+,\ (from\text{-}loc(op(d_{i,j})) = loc) \vee (to\text{-}loc(op(d_{i,j})) = loc)\Big)$$

- trucks should not be overloaded: $Z_c = P$

$$eval_c(sch, p) = \bigcup (interval(d_{i,j}) \mid d_{i,j} \in D_+,\ pr(d_{i,j}) = p,\ vol(s_{i,j}) > supvol)$$

- trucks should return to their starting points: $Zc = P$

  let $returntime(p)$ be the time necessary to return from the final location back to the starting point:

  $returntime(p) = speed(p) * dist(loc(s_{fin}(p)),\ start\text{-}\ loc(p))$

  and let $\alpha = maximum(0, t_{max} - returntime(p))$

$$eval_c(sch, p) = \begin{cases} \emptyset & \text{if } loc(s_{fin}(p)) = start\text{-}loc(p) \\ (\alpha, t_{max}] & \text{otherwise} \end{cases}$$

– two (or no) decisions are related to a processor and an operation: $Z_c = P \times O$

$$eval_c(sch, p, o) = \begin{cases} pr\text{-}op\text{-}set(p, o) & \text{if } \#\{(i,j) \mid d_{i,j} \in D_+,\ pr(d_{i,j}) = p,\ op(d_{i,j}) = o\} \neq 2 \\ \emptyset & \text{otherwise} \end{cases}$$

– cargos should be loaded before being unloaded: $Z_c = P \times O$

$$eval_c(sch, p, o) = \begin{cases} pr\text{-}op\text{-}set(p, o) & \text{if } \exists i, j_1, j_2 : j_1 < j_2,\ d_{i,j_1}, d_{i,j_2} \in D_+, \\ & \quad pr(d_{i,j_1} = pr(d_{i,j_2}) = p,\ op(d_{i,j_1}) = op(d_{i,j_2}) = o, \\ & \quad (loading(d_{i,j_1}) = \bot \vee loading(d_{i,j_2}) = \top) \\ \emptyset & \text{otherwise} \end{cases}$$

4.3. Time tables for schools

Problem description:

Weekly time tables for schools are considered. Classrooms (operations) are to be allocated to courses (processors) at certain time units. Allocation time is constant. It is assumed, that the allocation of teachers and classes to courses is not effectuated by a decision, since this is no scheduling problem. One should care, that teachers, classes and classrooms cannot be allocated to different courses simultaneously. However more than one class can participate in one course simultaneously. A classroom has a capacity, that has to correspond to the size of the classes.

Domain specific model parameters:

K	– a set of classes.
M	– a set of teachers.
n_{max}	– maximal number of lessons in a week.

Processors:

id	: course-identifier
teacher	: teacher of the course
classes	: set of classes participating in the course
subject	: subject of the course
students	: number of participating students
lessons	: number of lessons weekly

$$P \approx N \times M \times \mathcal{P}(\mathcal{K}) \times N \times N \times N$$

Operations:

id	: classroom-identifier
capacity	: capacity of the classroom

$$P \approx N \times R_+$$

Real decisions:

pr : processor reference
op : operation reference

$$D_+ \approx P \times O$$

Idle decisions:

pr : processor reference

$$D_0 \approx P$$

States (of a processor):

$execime$: the number of lessons executed

$$S \approx R_+$$

Initial state (of a processor):

$$s_0(p) = 0 \quad \forall p \in P$$

State transition:

$$f(state, dec) = \begin{cases} execime(state) + 1 & \text{if } dec \in D_+ \\ execime(state) & \text{if } dec \in D_0 \end{cases}$$

Duration of decision:

$g(state, dec) = 1$

that means: $interval(d_{i,j}) = (j-1, j]$

Scheduling interval:

$(0, t_{max}] = (0, n_{max}]$

Constraints:

- teachers are indivisible: $Z_c = M$

$$eval_c(sch, m) = \bigcup((j-1, j] \mid \#\{i \mid teacher(pr(d_{i,j})) = m\} > 1)$$

- classes are indivisible: $Z_c = K$

$$eval_c(sch, k) = \bigcup((j-1, j] \mid \#\{i \mid k \in classes(pr(d_{i,j}))\} > 1)$$

- classrooms are indivisible: $Z_c = O$

$$eval_c(sch, o) = \bigcup((j-1, j] \mid \#\{i \mid (op(d_{i,j})) = o\} > 1)$$

- restricted classroom capacity: $Z_c = O$

$$eval_c(sch, o) = \bigcup(interval(d_{i,j}) \mid d_{i,j} \in D_+,\ op(d_{i,j}) = o,\ students(pr(d_{i,j})) > capacity(o))$$

- not too many lessons for a course: $Z_c = P$

$$eval_c(sch, p) = \bigcup(interval(d_{i,j}) \mid d_{i,j} \in D_+,\ pr(d_{i,j}) = p,\ exectime(s_{i,j}) > lessons(p))$$

- enough lessons for a course: $Z_c = P$

$$eval_c(sch, p) = \bigcup(interval(d_{i,j}) \mid d_{i,j} \in D_0,\ pr(d_{i,j}) = p,\ (lessons(p) - exectime(s_{i,j})) > (n_{max} - j))$$

- teachers unavailability: $Z_c = M \times B(Time)$

$$eval_c(sch, m, period) := period \cap \Big(\bigcup(pr\text{-}op\text{-}set(p_i, o_k) \mid p_i \in P_S,\ o_k \in O_S,\ teacher(p_i) = m)\Big)$$

4.4. Time tables for nursery staff

Problem description:

Duty rosters for nursery staff in hospitals for a fixed time period is considered. Nurses (processors) can be on or off duty. To be on duty is the only operation. Nurses have different maximal working hours and different functions (sets of abilities). A certain minimal crew is required, which is time dependent and which is expressed in numbers of necessary functions (abilities).

Domain specific model parameters:

A – a set of abilities.

Processors:

id	: identifier
abilset	: set of abilities

suptime : maximal working time

$P \approx N \times \mathcal{P}(A) \times R_+$

Operations:

id : identifier (constant)

$O \approx \{o_1\}$ (singular set)

Real decisions:

pr : processor reference
op : operation reference
time : duration

$D_+ \approx P \times O \times R_+ \approx P \times R_+$

Idle decisions:

pr : processor reference
wait : waiting time

$D_0 \approx P \times R_+$

States (of a processor):

exectime : the total executing time of a processor
(till a decision point)

$S \approx R_+$

Initial state (of a processor):

$s_0(p) = 0 \quad \forall p \in P$

State transition:

$$f(state, dec) = \begin{cases} exectime(state) + time(dec) & \text{if } dec \in D_+ \\ exectime(state) & \text{if } dec \in D_0 \end{cases}$$

Duration of decision:

$$g(state, dec) = \begin{cases} time(dec) & \text{if } dec \in D_+ \\ wait(dec) & \text{if } dec \in D_0 \end{cases}$$

Constraints:

- maximal working times for nurses: $Z_c = P$

  $eval_c(p) = \{t \in Time \mid \textit{pr-op-dur}(p, o_1, t) > \textit{suptime}(p)\} \cap \textit{pr-op-set}(p, o_1)$

- minimal crew requirements: $Z_c = B(Time) \times P(A) \times N$

  $eval_c(\textit{sch}, \textit{period}, \textit{req-abilset}, \textit{min-charge}) := \textit{period} \cap$

  $\{t \in Time \mid \textit{min-charge} > \#\{i \mid \textit{busy}(p_i, o_1, t),\ \textit{abilset}(p_i) \cap \textit{req-abilset} \neq \emptyset\}$

- processor unavailability: see job-shop

5. System functionality and user actions

A prototype is implemented based on our model with a primitive user interface. This system supports step-wise planning. The user can perform *user actions* by evoking interactively *primitive functions* like adding or deleting "scheduling objects" like processors, operations, decisions or constraints. The system will evaluate these user actions, compute their consequences and do some feasibility checking.

These primitive functions are grouped with respect to the relevant object classes. After the choice of the relevant class (schedule, processor, etc.) the user may evoke a function (find, insert, etc.). The system will execute that function and perform state transitions if necessary.

A *transaction* is a sequence of user actions followed by a *commit*. After a *commit* command all constraint instances will be evaluated resulting in a set of critical regions. Moreover quality values will be calculated for all defined quality measures. In the case that any hard constraint is violated all actions performed since the last *commit* are rolled back.

Grouping of user actions by means of the transaction concept is necessary, because a single user action may result in an infeasibile schedule (some decisions may be effectuated at the same time involving different processors). To avoid the violation of hard constraints, eventually constraint checking should be done with respect to more than one user action.

With respect to the scheduling objects processor, operation, decision and constraint the following primitive functions are defined:

find, insert, update, delete

With the function *find* a specific object instance is made current within his class. Several options of this command are possible: *find direct*, *find first*, *find next*, *find last* and *find previous*. Direct search is always based on the object identifier. Sequential search for processors, operations and constraints is based on the order of the object identifiers; sequential search for decisions is based on the chronological order of their decision points. By means of the options *find within processor*, *find within operation* and *find within constraint type* the search process for decisions or constraints may be restricted.

The functions *update* and *delete* are related to scheduling objects, which are made current by previous *find* functions. When processors or operations are deleted, all related decisions and constraints are removed from the schedule as a side effect.

The function *insert* is rather obvious for processors and operations. To insert a constraint instance the relevant constraint type should be specified by the user. Before inserting a decision, first a processor, an operation and an decision should be made current by appropriate *find* functions. The new decision is inserted chronologically after the current decision and is related to the current processor and the current operation.

With respect to schedules we have the following primitive functions:

load, save, show, commit

With the function *show* the schedule will be represented on the screen. With the function *load* and *save* schedules can be loaded from or written to files, specified by the user.

By executing the function *commit* constraint instances will be evaluated and all previous user actions may be rolled back if any hard constraint is violated (see above).

It is important to observe, that the system can be used to manipulate decisions as well as processors, operations and constraints dynamically. The distinction between strategic, tactical and operational planning [4] is less strict in this context.

6. Architecture and implementation

6.1. Architecture

The architecture of a specific DSS is defined by an hierarchy of modules. On the top level we have generic modules. In these modules the *superclasses* of all scheduling objects (resource, decision, constraint, etc.) including their operators are defined. Only those attributes and methods that are common for all subclasses are defined here. Also time handling, schedule evaluation and feasibility checking belong to this level.

On a lower level we have domain-specific modules, containing the definitions of all relevant *sub-classes* of scheduling objects, with their specific attributes and operators. For example, in the job-shop case we can distinguish various constraints, like precedence-, deadline- and release time constraints. Also problem specific optimalization routines could be defined in these modules eventually.

Constraint types are defined in separate modules. This makes it possible to change and add constraint-types easily. As a consequence constraint types can be used in several planning situations.

A problem-specific DSS is generated by linking the generic modules with the relevant domain-specific modules.

Finally *object-instances* are supplied by the end user by means of the primitive actions, described in the last section. It is important to observe, that the manipulation of object instances by the end-user is related to decisions as well as to resources, operations and constraints.

The classification of "stakeholders" [20] involved in the development and the use of DSS in three groups (toolsmith, designer and manager) corresponds with the different levels of abstraction in the DSS-architecture: generic modules (superclasses), domain specific modules (subclasses) and object instances. Each group is on exactly one abstraction level involved in the development or the use of the system.

6.2. Implementation

A set of modules is implemented in C++ resulting in decision support systems for several planning problems [18]. These systems run on UNIX and have the basic functionality as described in the last section.

To have a simple measure of the reusability of a certain DSS, we take the ratio of generic software within the whole system. As module size we use the number of lines of pure source code. Table 6.1 shows reusability factors of systems for different planning situations, depending on the number of implemented constraints (indicated by #c). The numbers are smoothed to eliminate the influence of the order of the constraints. Furthermore we consider all modules as specific, that can be used in several but not

#c	jobshop	car-routing	time-tables for schools	time-tables for nursery
0	0.66	0.68	0.69	0.73
1	0.63	0.66	0.66	0.70
2	0.61	0.64	0.63	0.67
3	0.59	0.62	0.61	0.64
4	0.57	0.60	0.59	–
5	0.55	0.58	0.57	–
6	0.53	0.56	–	–
7	0.52	0.54	–	–
8	0.51	–	–	–

Table 6.1. Reusability factors

all planning situations. Especially this applies to modules describing constraint-types. Though matters are rather simplified, the table gives a rough idea, in how far our approach can lead to software reusability and code reduction.

7. Object definitions

In this section we give some relevant object definitions based on our model by means of a pseudo-language. We will define both generic and domain specific object classes with respect to the job-shop problem. The definitions are simplified for the sake of clearness.

7.1. Object representation

To describe our objects and to write pseudocode we use in this paper a fictive pascal-like object-oriented pseudolanguage. To define classes we have in this language structures called *objects*. Objects are comparable with records, with the difference that object-components can be both attributes and *object-methods* (functions or procedures):

```
object figure
   size: integer;
   colour: colourtype;
   method moveto(x,y: integer);
   method draw
end;
```

To define subclasses we can construct hierarchies of objects. An object is denoted as descendant of an other object in a hierarchy by specifying the name of the ancestor in parentheses (after the object name); attributes and methods of the higher class are inherited:

object triangle(figure)

If in the declaration of an object the name of a method is followed by the keyword *virtual*, this means that this method can be used in different subclasses under the same name, but with specific functionality (polymorphism). The semantics of such a method are not determined until run-time, when the method is invoked (dynamic binding).

We use the pseudo language structure *list of* to specify sequences of elements of a certain type

real_sequence = **list of** real

Moreover, we assume the following type as predefined

```
interval = record
             left, right: real;
           end;
```

7.2. Generic object classes

In this section we define the generic object classes:

```
schedule = object
  sch_processors : list of processor;
  sch_operations : list of operation;
  sch_decisions  : list of decision;
  sch_constraints: list of constraint;
  method schedule_evaluate: boolean;
  method schedule_show;
end;

scheduling_object = object
  s_id : integer; (*object identifier*)
  method s_insert (s: schedule); virtual;
  method s_delete (s: schedule); virtual;
  method s_retrieve (s: schedule); virtual;
  method s_update (s: schedule); virtual;
end;

processor = object (scheduling_object)
  p_initstate: state;
end;

operation = object (scheduling_object)
end;
```

```
state = object
  state_time: real;
  method state_transition (dec: decision); virtual;
  method state_show; virtual;
end;

decision = object (scheduling_object)
  d_pr    : processor;
  d_op    : operation;
  d_state : state;
  d_idle  : boolean;
  method d_getduration: real; virtual;
  method d_getdecisionpoint: real;
  method d_evaluate;
end;

real_decision = object (decision)
end;

idle_decision = object (decision)
end;

time_region = object
  crit_timeset = list of interval;
  method crit_length: real; (*total length*)
end;

constraint = object (scheduling object)
  c_hard   : boolean;
  c_critreg : time_region;
  method c_evaluate (s: schedule); virtual;
  method c_show_critical_region;
end;

quality-measure = object (scheduling object)
  q_list   : list of constraint; (*list of constraints*)
  q_value : real;
  method q_aggregate (rlist: list of real): real; virtual;
  method q_calculate_value (s: schedule);
end;
```

One of the most important methods is the function *schedule_evaluate* belonging to the object-class *schedule*, which checks the feasibility of a schedule by evaluating all its constraints. Constraint evaluation on its turn is performed by the method *c_evaluate* belonging to the object-class *constraint*. The semantics of the latter method are not determined until run-time, when the method is invoked and the constraint is initialized. This property is called *dynamic binding* and it is above all this property, which makes it possible to treat constraints so generically.

7.3. Domain specific object classes (job-shop)

In this subsection we define the domain specific object classes for the job-shop problem.

```
ability = (a1,...,an);

rcps_processor = object (processor)
  p_abilset : set of ability;
  p_speed   : real;
end;

rcps_operation = object (operation)
  o_abilset : set of ability;
  o_size    : real;
end;

rcps_real_decision = object (real_decision)
  d_duration: real; (*duration of assignment*)
  method d_getduration: real; virtual;
end;

rcps_idle_decision = object (idle_decision)
  d_waiting: real; (*waiting time*)
  method d_getduration: real; virtual;
end;

rcps_state = object (state)
  busytime: real;
  method state_transition (dec: decision); virtual;
  method state_show; virtual;
end;

deadline_constraint = object (constraint)
  c_op       : rcps_operation;
  c_deadline : real;
  method c_evaluate (s: schedule); virtual;
end;

precedence_constraint = object (constraint)
  c_op1, c_op2 : rcps_operation;
  c_waiting    : real
  method c_evaluate (s: schedule); virtual;
end;
```

```
availability_constraint = object (constraint)
  c_pr        : rcps_processor;
  c_intervals : list of interval;
  method c_evaluate (s: schedule); virtual;
end;
```

8. Conclusions and extensions

Although scheduling problems vary a lot with respect to their optimalization aspect, they have much more in common, when we restrict ourselves to stepwise planning. When designing a decision support system in a more generic way, we should start therefore with components for pure schedule management. Algorithmic components may be implemented later in layers on top of the basic software. This may be the case both for flexible generic optimization procedures (like simulated annealing) as for powerful problem specific algorithms.

In view of the natural hierarchy of scheduling problems, an object oriented implementation seems to be suitable. It leads to a clear separation between the generic and the domain specific components of the schedule manager, minimizing redundant code and resulting in software with a high degree of maintainability and modularity.

In this paper we made a simplification by assuming only two levels of abstraction. The classification tree of scheduling problems has in fact more levels, which should reflect in the system architecture. Software modules should correspond always to a specific node in that tree. Another simplification is the assumed homogeneity of resources, operations and decisions, which may be not always realistic.

We have to emphasize again, that the optimality of schedules is not an objective in this paper. Important is the optimality of the software in the sense of maintainability, flexibility and reusability.

References

[1] Aarts, E.H.L., A.E. Eiben, K.M. van Hee (1989), A General Theory of Genetic Algorithms, Computer Science Notes, Eindhoven University of Technology.

[2] Alter, S.L. (1980), Decision Support Systems: Current Practice and Continuing Chalenges, Addison-Wesley.

[3] Anthonisse, J.M., J.K. Lenstra, M.W.P. Savelsbergh (1988), Behind the Screen: DSS from an OR point of View; Decision Support Systems 4, Elsevier Science Publishers (North-Holland).

[4] Anthony, R.N. (1965), Planning and Control Systems: A Framework for Analysis; Harvard University Press, Studies in Management Control, Cambridge Mass.

[5] Baker, K.R. (1974), Introduction to Sequencing and Scheduling, J.Wiley, New York.

[6] Bennet, J.L. (ed.) (1983), Building Decision Support Systems, Addison-Wesley.

[7] Bonczek, R.H., C.W. Holsapple, A.B. Whinston (1983), Specification of Modelling Knowledge in Decision Support Systems; Processes and Tools for Decision Support (Ed. H.G.Sol), North-Holland Publishing Company.

[8] Bots, P.W.G., A. Verbraeck (1989), Object Oriented Task Description: Concepts, Tools and Applications; paper presented at the Session on Object Oriented Approaches in Information System Design, at the 15th IFIP WG 8.1 meeting, June 7th, 1989, in Sesimbra, Portugal.

[9] Coffman, E.G. (1976), Computer and Job Shop Scheduling Theory, J. Wiley, New York.

[10] Eiben, A.E., K.M. van Hee (1990), Knowledge Representation and Search Methods for Decision Support Systems; Data, Expert Knowledge and Decisions (eds. W. Gaul, M. Schader), NATO ASI Series, Vol. F61, Springer-Verlag.

[11] Fox, M.S. (1983), Constraint Directed Search: A Case Study of Job Shop Scheduling; PhD Thesis, Computer Science Dept., Carnegie Mellon University.

[12] Fox, M.S., S.F. Smith (1984), ISIS – a Knowledge-based System for Factory Scheduling; Expert systems, Vol. 1, No. 1.

[13] French, S. (1982), Sequencing and Scheduling, J.Wiley, New York.

[14] Hee, K.M. v., A. Lapinski (1989), OR and AI Approaches to Decision Support Systems; Decision Support Systems Vol. 4.

[15] Jansen, A., L. Klieb, C. Noorlander, G. Wolf (1990), 'PLATE', a Decision Support System for Resource Constrained Project Scheduling Problems, Designing Decision Support Systems Notes, NFI 11.90/03, Eindhoven University of Technology.

[16] Keen, P.G.W., M.S. Scott Morton (1978), Decision Support Systems: an Organizational Perspective, Addison-Wesley.

[17] Kim, W., F.H. Lochovsky (Eds.) (1989), Object-Oriented Concepts, Databases and Applications; ACM Press, Addison-Wesley.

[18] Litjens, R.T.J. (1991), Een objectgeorienteerde benadering van algemene roosterproblemen; Master Thesis, Computer Science Dept., Eindhoven University of Technology.

[19] Meyer, B. (1988), Object Oriented Software Construction, Prentice-Hall.

[20] Spraque, R.H., E.D. Carlson (1982), Building Effective Decision Support Systems, Prentice-Hall.

In this series appeared:

89/1	E.Zs.Lepoeter-Molnar	Reconstruction of a 3-D surface from its normal vectors.
89/2	R.H. Mak P.Struik	A systolic design for dynamic programming.
89/3	H.M.M. Ten Eikelder C. Hemerik	Some category theoretical properties related to a model for a polymorphic lambda-calculus.
89/4	J.Zwiers W.P. de Roever	Compositionality and modularity in process specification and design: A trace-state based approach.
89/5	Wei Chen T.Verhoeff J.T.Udding	Networks of Communicating Processes and their (De-)Composition.
89/6	T.Verhoeff	Characterizations of Delay-Insensitive Communication Protocols.
89/7	P.Struik	A systematic design of a parallel program for Dirichlet convolution.
89/8	E.H.L.Aarts A.E.Eiben K.M. van Hee	A general theory of genetic algorithms.
89/9	K.M. van Hee P.M.P. Rambags	Discrete event systems: Dynamic versus static topology.
89/10	S.Ramesh	A new efficient implementation of CSP with output guards.
89/11	S.Ramesh	Algebraic specification and implementation of infinite processes.
89/12	A.T.M.Aerts K.M. van Hee	A concise formal framework for data modeling.
89/13	A.T.M.Aerts K.M. van Hee M.W.H. Hesen	A program generator for simulated annealing problems.
89/14	H.C.Haesen	ELDA, data manipulatie taal.
89/15	J.S.C.P. van der Woude	Optimal segmentations.
89/16	A.T.M.Aerts K.M. van Hee	Towards a framework for comparing data models.
89/17	M.J. van Diepen K.M. van Hee	A formal semantics for Z and the link between Z and the relational algebra.

90/1	W.P.de Roever- H.Barringer- C.Courcoubetis-D.Gabbay R.Gerth-B.Jonsson-A.Pnueli M.Reed-J.Sifakis-J.Vytopil P.Wolper	Formal methods and tools for the development of distributed and real time systems, p. 17.
90/2	K.M. van Hee P.M.P. Rambags	Dynamic process creation in high-level Petri nets, pp. 19.
90/3	R. Gerth	Foundations of Compositional Program Refinement - safety properties - , p. 38.
90/4	A. Peeters	Decomposition of delay-insensitive circuits, p. 25.
90/5	J.A. Brzozowski J.C. Ebergen	On the delay-sensitivity of gate networks, p. 23.
90/6	A.J.J.M. Marcelis	Typed inference systems : a reference document, p. 17.
90/7	A.J.J.M. Marcelis	A logic for one-pass, one-attributed grammars, p. 14.
90/8	M.B. Josephs	Receptive Process Theory, p. 16.
90/9	A.T.M. Aerts P.M.E. De Bra K.M. van Hee	Combining the functional and the relational model, p. 15.
90/10	M.J. van Diepen K.M. van Hee	A formal semantics for Z and the link between Z and the relational algebra, p. 30. (Revised version of CSNotes 89/17).
90/11	P. America F.S. de Boer	A proof system for process creation, p. 84.
90/12	P.America F.S. de Boer	A proof theory for a sequential version of POOL, p. 110.
90/13	K.R. Apt F.S. de Boer E.R. Olderog	Proving termination of Parallel Programs, p. 7.
90/14	F.S. de Boer	A proof system for the language POOL, p. 70.
90/15	F.S. de Boer	Compositionality in the temporal logic of concurrent systems, p. 17.
90/16	F.S. de Boer C. Palamidessi	A fully abstract model for concurrent logic languages, p. p. 23.
90/17	F.S. de Boer C. Palamidessi	On the asynchronous nature of communication in logic languages: a fully abstract model based on sequences, p. 29.

90/18	J.Coenen E.v.d.Sluis E.v.d.Velden	Design and implementation aspects of remote procedure calls, p. 15.
90/19	M.M. de Brouwer P.A.C. Verkoulen	Two Case Studies in ExSpect, p. 24.
90/20	M.Rem	The Nature of Delay-Insensitive Computing, p.18.
90/21	K.M. van Hee P.A.C. Verkoulen	Data, Process and Behaviour Modelling in an integrated specification framework, p. 37.
91/01	D. Alstein	Dynamic Reconfiguration in Distributed Hard Real-Time Systems, p. 14.
91/02	R.P. Nederpelt H.C.M. de Swart	Implication. A survey of the different logical analyses "if...,then...", p. 26.
91/03	J.P. Katoen L.A.M. Schoenmakers	Parallel Programs for the Recognition of *P*-invariant Segments, p. 16.
91/04	E. v.d. Sluis A.F. v.d. Stappen	Performance Analysis of VLSI Programs, p. 31.
91/05	D. de Reus	An Implementation Model for GOOD, p. 18.
91/06	K.M. van Hee	SPECIFICATIEMETHODEN, een overzicht, p. 20.
91/07	E.Poll	CPO-models for second order lambda calculus with recursive types and subtyping, p.
91/08	H. Schepers	Terminology and Paradigms for Fault Tolerance, p. 25.
91/09	W.M.P.v.d.Aalst	Interval Timed Petri Nets and their analysis, p.53.
91/10	R.C.Backhouse P.J. de Bruin P. Hoogendijk G. Malcolm E. Voermans J. v.d. Woude	POLYNOMIAL RELATORS, p. 52.
91/11	R.C. Backhouse P.J. de Bruin G.Malcolm E.Voermans J. van der Woude	Relational Catamorphism, p. 31.
91/12	E. van der Sluis	A parallel local search algorithm for the travelling salesman problem, p. 12.
91/13	F. Rietman	A note on Extensionality, p. 21.
91/14	P. Lemmens	The PDB Hypermedia Package. Why and how it was built, p. 63.

91/15	A.T.M. Aerts K.M. van Hee	Eldorado: Architecture of a Functional Database Management System, p. 19.
91/16	A.J.J.M. Marcelis	An example of proving attribute grammars correct: the representation of arithmetical expressions by DAGs, p. 25.
91/17	A.T.M. Aerts P.M.E. de Bra K.M. van Hee	Transforming Functional Database Schemes to Relational Representations, p. 21.
91/18	Rik van Geldrop	Transformational Query Solving, p. 35.
91/19	Erik Poll	Some categorical properties for a model for second order lambda calculus with subtyping, p. 21.
91/20	A.E. Eiben R.V. Schuwer	Knowledge Base Systems, a Formal Model, p. 21.
91/21	J. Coenen W.-P. de Roever J.Zwiers	Assertional Data Reification Proofs: Survey and Perspective, p. 18.
91/22	G. Wolf	Schedule Management: an Object Oriented Approach, p. 26.
91/23	K.M. van Hee L.J. Somers M. Voorhoeve	Z and high level Petri nets, p. 16.
91/24	A.T.M. Aerts D. de Reus	Formal semantics for BRM with examples, p. .
91/25	P. Zhou J. Hooman R. Kuiper	A compositional proof system for real-time systems based on explicit clock temporal logic: soundness and complete ness, p. 52.
91/26	P. de Bra G.J. Houben J. Paredaens	The GOOD based hypertext reference model, p. 12.
91/27	F. de Boer C. Palamidessi	Embedding as a tool for language comparison: On the CSP hierarchy, p. 17.
91/28	F. de Boer	A compositional proof system for dynamic proces creation, p. 24.