# A RULE PROCESSING SYSTEM FOR KNOWLEDGE-BASED MANPOWER PLANNING AND SCHEDULING

**M. Prokopenko, C. Lindley, M. Milosavljevic, D.-M. Zhang**

CSIRO Division of Mathematical and Information Sciences, Locked Bag 17, North Ryde, NSW 2113, Australia.

## ABSTRACT

Real world optimisation problems are usually highly constrained and complex due to the number and variety of constraints. Often these constraints are human preference rules and thus are subject to revisions and updates. The paper analyses requirements on knowledge representation formalisms imposed by a practical constrained optimisation problem and addresses the problem of validation and refinement of constraints in the context of dynamic knowledge-based scheduling. The CSIRO Division of Information Technology in collaboration with The Preston Group (TPG) has developed a prototype Rule Editor and dynamic Rule Processing system for integration with the TPG Manpower Planning and Scheduling System (MPSS). The Rule Editor allows end-users to specify and validate constraints upon schedules using easy-to-use structured menu functions. All constraints are expressed in terms used by airport staff, and there is no need for users to learn any formal constraint or programming language. All rules defined using the Rule Editor are automatically entered into a knowledge base for processing by MPSS. MPSS verifies, interprets and processes the rules using the CSIRO prototype Rule Processing System (RPS). The developed knowledge representation formalism is sufficiently general and allows to cover majority of MPSS constraints. It is possible to load the rule base at any time, resulting in a completely dynamic Rule Processing system. The paper details the analysis and design of the proposed knowledge representation formalism, and implementation of the Rule Processing System.

## INTRODUCTION

The Preston Group/CSIRO Division of Information Technology collaborative project in Rule Editing, Processing, and Schedule Interaction is a joint project to develop a prototype rule (or constraint) editing and processing environment for the TPG Manpower Planning and Scheduling System (MPSS).

The MPSS is a system for generating personnel schedules for airports. MPSS scheduling is accomplished by two optimisers. The Roster Optimiser generates monthly schedules for assigning shifts to people according to work requirements. The Task Allocation Optimiser generates daily schedules that take short-term information and data variations into account. In both cases, during execution of the optimisation algorithms partial solutions are checked against a rule base that specifies and verifies constraints that must be satisfied by a schedule; if any constraints are violated, that cycle of the algorithm is repeated. This results in a schedule that conforms to the rules in the rule base, while also being near-optimal in the sense of minimising the difference between work required and the actual resources rostered on.

The Rule Editor must be easy to use by airport staff, to allow dynamic rule validation and modification by staff during airport operations, without support by technical specialists or the MPSS developers. Dynamic verification and modification of rules during operation requires the development of a high-level rule language and rule processing system that satisfies the following requirements as specified by TPG:

1. the ability to manipulate and query the MPSS object model including the following types: "person" (~400), "shift" (~35 x 400), and "task" (~30,000 - 40,000)[1]. All of these objects can have attributes that can be referred to by rules.
2. the ability to process simple data types (integer, real, boolean, date, time, datetime).
3. to be fast enough to be used by the MPSS optimisers within the normal turnaround time for schedule generation.

All of these requirements are satisfied by the CSIRO prototype Rule Processor together with the Rule Editor[2]. The MPSS prototype Rule Editor operates as a stand-alone program for the definition and validation of MPSS scheduling rules using an application-specific structured interface system. The editor stores the rule base in a text file for input to the rule processor via global data structures within MPSS. The discrepancies of format between the Rule Editor output and the internal representation used by the Rule Processor are addressed by an automated translation process invoked before the Rule Processor is used within the MPSS. The Rule Processor is an integrated program that can be compiled into the MPSS system. When MPSS is started, a rule loading function reads and parses the rule base file, loading the rules into internal MPSS data structures. When MPSS generates a schedule, the rules in the data structures are automatically interpreted. This system allows modifications to the rule base to be made at any time, with loading into MPSS initiated by execution of the loading function.

---

[1] One person can have a number of shifts, and one shift can have a number of tasks.

[2] The rule editor is described in detail in [4]. Here we will only briefly outline its features.

## KNOWLEDGE REPRESENTATION AND CONSTRAINTS MANAGEMENT

Natural Language Interfaces (NLIs) have attracted much attention over recent years with the aim of providing users with the ability to express queries and/or commands in a natural language. Database queries, and more recently database updates [3], are examples of successful applications that have encouraged this research. Although limited, this technology provides an insight into how to design NLIs to rule bases. NLIs to Databases (NLIDBs) [1, 2] typically retrieve or modify information in database tables, whereas NLIs to knowledge bases modify rules or constraints that have a direct impact on the actual performance and results of a reasoning system. The following sections contain analysis of requirements on knowledge representation formalisms imposed by the constrained optimisation problem. Such an analysis allows us to address the problem of validation and refinement of constraints in the context of dynamic knowledge-based scheduling.

## RULE REPRESENTATION AND PROCESSING

There are several possible strategies for processing the rules defined using the Rule Editor. Among these strategies, a customised and integrated Rule Processing System (RPS) has been chosen as the cheapest and most convenient solution for MPSS. Integrating MPSS with an RPS avoids re-compilation/re-linking of the system after each modification, and allows the rule base to be changed dynamically using the Rule Editor, even during execution of schedule generation. This section describes the analysis, design, implementation and operation of the prototype RPS developed by CSIRO. Options for further enhancements to the RPS are discussed in the concluding section.

In the original MPSS system rules are expressed in C++ code. The implementation of each rule is characterised by the following general scheme:

1. introduction of a parameter representing a constrained entity, eg. 'early_shift' for the number of early shifts initially set as 0;
2. determination of a time interval and a staff category composing a domain of the constraint, eg. a constraint applies for every staff member (no qualification on person’s attribute) and for every day of the current week;
3. definition of a boolean expression representing a condition for updating the parameter introduced in step 1, eg.

```
test[k].status == 'W'  &&  time_of_day(test[k].start) <= 6.5,
```

where test[k] is a current schedule, and 'W' (“working day”) and 6.5 (06:30 am) are constants;

4. definition of an action that must be performed if the boolean expression is true; usually the action incorporates the parameter, eg.

```
early_shift++;
```

5. checking the final value of the parameter against a specified constraint and sending a return value when the constraint is violated, eg.

```
if (early_shift > 2)  broken_rule(3211);
```

The rostering rule used as an example in steps 1-5 constrains the number of early shifts per week: "A shift starting on or before 06.30 will be assigned a maximum of 2 times per rostering week".

In order to handle the problem of rule processing in an efficient way it is useful to define a constraint and a corresponding rule (rules) formally. Such a formal definition would correspond to a rule representation in the RPS. For each constraint j we define a domain of the constraint $D_j$ , k-dimensional vector of parameters $P_j$ , n-dimensional vector $X_j$ of rule attributes, vectors $A_j$ and $B_j$ of the left and right bounds respectively for vector $X_j$, an operator $F_j$ mapping a truth-valued function $e_j$ onto a function $r_j$ , and a truth-valued function $C_j$ representing constraint j itself. In other words,

$\forall\, x_1 \in (a_1, b_1), \ldots, \forall\, x_n \in (a_n, b_n)$ ( or $\forall\, X,\ A < X < B$ )
$\exists\, F_j : e_j(x_1, \ldots, x_n, P_j) \rightarrow r_j(P_j)$ & $\exists\, C_j$ э
$C_j(r_j(P_j)) \in \{\text{True, False}\}$.

The Cartesian product $(a_1, b_1) \otimes \ldots \otimes (a_n, b_n) \otimes \mathbb{R}^k$ constitutes a domain $D_j$ of a constraint. In general, the intervals can be closed. If $a_i = b_i$ (i = 1,...,n) then the corresponding rule's attribute must be equal to this value. The function $e_j$ is a boolean expression with disjunctions, conjunctions, and negations, and the function $r_j$ represents an action that must be performed if the boolean expression is evaluated as True. In general, the function $r_j$ is a recursive function defined in the domain $D_j$ of the constraint j. Complete rules in this case might be represented as:

"IF $C_j(r_j(P_j))$ is TRUE THEN constraint j is satisfied"
"IF $C_j(r_j(P_j))$ is FALSE THEN constraint j is not satisfied".

For the example considered above:

- condition $e_j$ is defined by:

  test[k].status == 'W' && time_of_day(test[k].start) <= 6.5,

  where the attribute test[k].status is bounded to be equal to 'W', and the attribute test[k].start is bounded from above by value 6.5.
- action $r_j$ is defined by:

  early_shift++;

  where early_shift is a parameter and $r_j$ is a recursive function (increment) defined in the "weekly" sub-domain of $D_j$ .
- the if-then sentence:

  if $(e_j)$ then $r_j$

sets the operator $F_j$ in the domain $D_j$ .

- the negation

  not (early_shift > 2)

  defines $C_j$ taking as an argument the final value $r_j(P_j)$ = early_shift of the parameter $P_j$.

The task of validation of a rule and verification of the rule-base can be reduced then to

- the problem of identification of the domain of the constraint (roster period, day type, staff category);
- construction of the boolean condition(s);
- constrained parameter specification and logical constraint description;
- type of constraint (type of action) determination (feasibility constraint leads to a non-recursive function and aggregation constraint leads to a recursive function),

and each of these sub-tasks can be associated with a respective part of a constraint expressed using the Rule Editor.

## RULE PROCESSING SYSTEM

The MPSS Rule Processing System is written in C++ and integrated with the MPSS. Every constraint is represented in the following structures (frames): *Rule*, *Rule_index*, and *Constraint*.

The structure *Rule* comprises the following constraint's attributes (slots): *number, counter, domain, status, person* and includes also left and right bounds for every attribute of a shift such as its start and finish times, meal break times, etc. The *counter* slot numerically indicates what is being counted: number of times condition is satisfied (ie., number of "events") (*counter* = 1), or number of worked hours (*counter* = 7), or number of rest hours (*counter* = 8), etc., where numbers in brackets correspond to predefined types of aggregation. The *domain* slot analogously sets the type of a constraint's time domain: a single shift (*domain* = 1), a week (*domain* = 2), etc. The status slot is responsible for the type of day property: working day ("W"), day off ("O"), etc. and finally the *person* slot represents a constrained staff category. In other words, the structure *Rule* depicts a type of a constraint's domain and sets bounds (vectors A and B) for shift's attributes (vector X).

The structure *Rule_index* sets types of a relation for all relevant (mentioned in a constraint's condition) attributes according to the following table:

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
|---|---|---|---|---|---|---|---|---|---|---|
| N/A | X = A | X ≠ A | X < B | X ≤ B | X > A | X ≥ A | A < X < B | A < X ≤ B | A ≤ X < B | A ≤ X ≤ B |

where the second row displays types of relation and the numbers in the first row encode these types numerically. X is the value of the corresponding shift's attribute. A and B are two bounds (left and right, respectively) of the attribute and are stored in the *Rule* structure.

The *Constraint* structure contains left and right values of a constrained parameter and the type of a relation encoded according to the aforementioned table. Every constraint is represented in every structure only once. Consider an example. The constraint "A shift starting on or before 06.30 will be assigned a maximum of 2 times per rostering week" is represented in the RPS in the following way:

```
Rule {. . . . . . 3211, 1, 2, 'W', 0, 0, 6.5, 0, . . . , 0, 0 };
Rule_index {. . . 3211, 1, 0, 4, 0, . . . . . , 0 };
Constraint {. . . 3211, 0, 2, 4 };
```

The semantics of these syntactic descriptions is:

- the RPS has to count number of events (*Rule.counter* = 1);
- the event is defined by the condition:

| | | |
|---|---|---|
| *Shift.status* = 'W' | ( *Rule_index.status* = 1 and *Rule.status* = 'W') | and |
| *Shift.start* ≤ 6.5 | ( *Rule.index.start* = 4 and *Rule.start_right* = 6.5) | |

- nothing else is applicable since other slots in Rule_index are 0's (by default *Rule.person* = 0 means that a rule applies for all types of employees).
- the number of events is less than or equal (*Constraint.index* = 4) to 2 (*Constraint.value_right* = 2): n ≤ 2, and this constraint has to be checked for every week (*Rule.domain* = 2).

The Rule Processing System consists of blocks (nested while/for loops) selected during run-time according to the value of the *Rule.domain* slot (week/roster, etc.). Each block matches the current shift to a condition retrieved from *Rule* and *Rule_index* structures by using a pattern-matching function. The function returns *Rule.counter* only if the condition is satisfied (otherwise it returns 0). This feature provides the conjunctive connection among all restrictions on rule attributes with non-zero indexes. Depending on the returned value of the *Rule.counter* a subsequent action is performed and a constrained parameter is computed. Finally, the computed value is checked against decoded constraint's bounds.

Since a rostering rule defined for every shift (ie. for every day) does not suppose any kind of aggregation, only feasibility constraints can be checked in the single shift domain. This leads to non-recursive actions of the form: r(k) = x, where k is the shift number. This means that the final value to check after successful pattern-matching is just equal to the value of a corresponding slot (shift's start, finish, working hours, etc.). If pattern-matching was unsuccessful then the current constraint is not applicable to the current shift and the RPS moves to the next constraint in the external loop. If none of shifts violates any constraint then the RPS accepts the current partial solution. Otherwise the partial solution is rejected.

In the weekly domain case the RPS has to count the number of "events" (counter returned after pattern-matching is equal to 1), the number of working hours (counter = 7), or the number of some other countable slot's values (counter > 1) for every week of rostering period. This counting is conducted for all shifts matched to a pattern inside the internal loop (all days of week) by a recursive function r(k) = r(k-1) + x. A constraint is checked immediately after this loop. If it is satisfied then the RPS moves to next week in the external loop. Again, as in the previous case, if none of the weeks violates the constraint then the RPS accepts the candidate solution.

The RPS handles also cases of a whole roster and a seven days rolling period by using similar algorithms. This rule representation is sufficiently general to cover 90% of MPSS constraints.

A constraint should be entered into the editor [4] with care, taking into account exactly what information is and is not required for the Rule Processor. The section "Applies to:" specifies the logical entities to which the constraint applies. In some cases, these are not applicable (n/a) to the rule. The first entity is the type of person for which the rule must be true. At this stage, there are five different staff categories listed in the menu: all staff; Airport Service Officers (ASOs) only;

supervisors only; pregnant staff; new ASOs. The type of day is another important factor. For example, one of rules specifies the number of days off per rostering week, and so this field will be selected as “days off”. The menu also includes “working days”, “training days”, and “holidays”. The roster period for which the rule applies is often required and can be selected from the menu containing “roster week”, “rolling 7 day period”, “roster month”, whole roster period”, and “single shift”.

The “Conditions:” area allows the user to enter the logical conditions which must be true for the constraint to take effect. Both conditions and constraints have the same input format in which a variable has a particular boundary on it’s value. The variable is selected from the menu listing all shift’s attributes such as shift start time, shift finish time, meal start, etc., and is followed by a relationship ( $=$, $\neq$, $<$, $\leq$, $>$, $\geq$) selected from another menu. The shift’s attributes menu offered for constraint(s) input is extended by quantitative characteristics such as number of days, number of shifts, etc. The value of the boundary is then entered by the user.

## CONCLUSIONS

This project has successfully developed a prototype Rule Editor and Rule Processor suitable for integration with the TPG Manpower Planning and Scheduling System (MPSS). These enhancements to MPSS will facilitate validation and modification of constraints upon schedules by airline staff, eliminating the need for code changes, rebuilding, and reinstallation of the MPSS system. Additional areas of potential ongoing work include generalisation of the editor so that object and relationship types are modifiable data values, instead of being hard-coded as in the current prototype; and development of redundancy and conflict detection for rules expressed using the editor. The Rule Processing System can also be enabled to gather statistics about rule violation, and these statistics can be used to substantially reduce verification and rule processing time (and therefore schedule generation time) by reordering the rule base so that more frequently violated rules are checked first. These enhancements will increase the generality of the Rule Processing System, allowing it to function as a generic software module suitable for integration with a wide range of scheduling systems in different application domains.

## REFERENCES


1. I. Androutsopoulos, G.D. Ritchie and P. Thanisch, *Natural Language Interfaces to Databases - An Introduction*, University of Edinburgh, Department of AI, Edinburgh, Scotland, 1992.
2. S.K. Cha, *Kaleidoscope: A Model-Based Grammar-Driven Menu Interface for Databases*, PhD thesis, Stanford University, Department of Computer Science, California, Report No. STAN-CS-92-1405, 1991.
3. R. Fagin, J. Ullman and M. Vardi. On the semantics of updates in databases, in the *Proceedings Second ACM Symposium on Principles of Database Systems*, Atlanta, CA, 1983, pp. 352-365.
4. C. Lindley, M. Prokopenko, M. Milosavljevic, D.-M. Zhang. A Rule Editor and Processing System for Manpower Planning and Scheduling, in the *Proceedings of the Workshop on Verification, Validation and Refinement of KBS* (The 4th Pacific Rim International Conference on Artificial Intelligence, PRICAI'96), 1996, pp. 37-46.