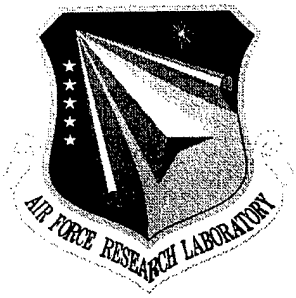
AFRL-IF-RS-TR-2000-11
Final Technical Report
February 2000



REQUIREMENTS-BASED SYNTHESIS OF CREW SCHEDULING SOFTWARE

BBN Technologies

Mark H. Burnstein

APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED.

AIR FORCE RESEARCH LABORATORY
INFORMATION DIRECTORATE
ROME RESEARCH SITE
ROME, NEW YORK

This report has been reviewed by the Air Force Research Laboratory, Information Directorate, Public Affairs Office (IFOIPA) and is releasable to the National Technical Information Service (NTIS). At NTIS it will be releasable to the general public, including foreign nations.

AFRL-IF-RS-TR-2000-11 has been reviewed and is approved for publication.

APPROVED:

JOHN F. LEMMER
Project Engineer

FOR THE DIRECTOR:

NORTHRUP FOWLER
Technical Advisor
Information Technology Division

If your address has changed or if you wish to be removed from the Air Force Research Laboratory Rome Research Site mailing list, or if the addressee is no longer employed by your organization, please notify AFRL/IFTB, 525 Brooks Road, Rome, NY 13441-4505. This will assist us in maintaining a current mailing list.

Do not return copies of this report unless contractual obligations or notices on a specific document require that it be returned.

REPORT DOCUMENTATION PAGE	Form Approved OMB No. 0704-0188

1. AGENCY USE ONLY *(Leave blank)*	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	FEBRUARY 2000	Final May 97 - Jun 98

4. TITLE AND SUBTITLE
REQUIREMENTS-BASED SYNTHESIS OF CREW SCHEDULING SOFTWARE

5. FUNDING NUMBERS
C - F30602-96-D-0058/0009
PE - 63728F
PR - 2527
TA - QB
WU - 09

6. AUTHOR(S)
Mark H. Burnstein

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
BBN Technologies
10 Moulton Street
Cambridge MA 02138

8. PERFORMING ORGANIZATION REPORT NUMBER
N/A

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)
Air Force Research Laboratory/IFTB
525 Brooks Road
Rome NY 13441-4505

10. SPONSORING/MONITORING AGENCY REPORT NUMBER
AFRL-IF-RS-TR-2000-11

11. SUPPLEMENTARY NOTES
Air Force Research Laboratory Project Engineer: John F. Lemmer/IFTB/(315) 330-3657

12a. DISTRIBUTION AVAILABILITY STATEMENT
APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED.

12b. DISTRIBUTION CODE

13. ABSTRACT *(Maximum 200 words)*

This project has explored the relationship between functional requirements and software design and development. Working with Kestrel Institute, we prototyped some new scheduling algorithms for assigning crew members to aircraft flights using Kestrel's KIDS environment to generate the algorithms from formal specifications. As these algorithms were developed, we traced how the requirements impacted various aspects of the overall development effort, including the scheduling algorithm, the data management and user interface aspects of the system. We considered how evolving requirements interact with algorithms and systems development in the process of producing usable software tools. We describe how, in our particular case, requirements impacted algorithm and data design, and how these things in turn impact the GUI that is an integral part of the software tool produced.

Our algorithm development effort used Kestrel's KIDS environment, and also looked at some alternative scheduling algorithms developed by hand. We also developed some general purpose GUI tools using JAVA that have since been used with KIDS-generated schedulers in other DARPA/AFRL funded projects for the Air Mobility Command.

14. SUBJECT TERMS
Scheduling Algorithms, KIDS Environment, Synthesizing Software, Air Crew Schedules

15. NUMBER OF PAGES
56

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
UNCLASSIFIED	UNCLASSIFIED	UNCLASSIFIED	UL

TABLE OF CONTENTS

1. INTRODUCTION AND OBJECTIVES 1

2. OVERVIEW OF THE CREW SCHEDULING PROBLEM 3

3. SUMMARY OF SCHEDULER REQUIREMENTS 5

USER/OPERATIONAL REQUIREMENTS 6

DATA DESIGN REQUIREMENTS 7

HARD CONSTRAINTS 11

SOFT CONSTRAINTS 14

OPEN REQUIREMENTS ISSUES 18

4. DEVELOPMENT APPROACH 20

ALGORITHM STATISTICS 21

5. TRACKING AND TRANSLATING REQUIREMENTS INTO CODE 23

6. DISCUSSION AND CONCLUSIONS 27

7. SOME DIRECTIONS FOR FUTURE WORK 29

APPENDIX 1 – CREW SCHEDULER REQUIREMENTS SUMMARY 30

APPENDIX 2 – KIDS THEORY OF CREW SCHEDULER 31

List of Figures

Figure 1: High-level system architecture....7
Figure 2: Semantic relations between crew scheduling domain elements....8
Figure 3: Relational Database Organization....9
Figure 4: Major data type definitions in REFINE....10
Figure 5: Algorithm I/O specification for optimizing version of KIDS scheduler...12
Figure 6: A hard constrain – a crewmen must be qualified for the position 13
Figure 7: Incremental test for separation of flights by one crew member....13
Figure 8: Graph of currency utility over time period....16
Figure 9: Graph of original periodic utility over time period....16
Figure 10: Graph of revised periodic utility over time period....17
Figure 11: Progression of algorithms developed....20
Figure 12: Flow of relationships from requirements to system components....23
Figure 13: Relationship of Hard Constrain to Search Algorithm....25

1. Introduction and Objectives

Over the past year, we have worked, with the assistance of Kestrel Institute, on the development of new scheduling algorithms for assigning crew members to aircraft flights. Such algorithms are needed to assist and support those people at Air Force and reserve wings whose job it is to ensure that all crew members receive adequate training and maintain currency on their assigned roles. The original stated goals for this project were, in addition to new algorithm development, to explore the relationship between requirements and software design and development, using Kestrel Institute's KIDS environment for synthesizing software from formal specifications and tools developed by others for requirement tracking. The goal was to produce a demonstration of a scheduling tool for a military domain and to follow the process of requirement migration into the software produced, and especially to consider how changing requirements impacted the software.

We have attempted to follow the spirit of these original objectives, despite some problems with the availability of the tools that we originally anticipated using. The original plan for the project called for the use and evaluation of a third-party prototype tool (ADM) for tracking requirements into software. Unfortunately, this tool could not be provided to BBN, and was no longer being maintained, so we have done our best to track requirements using other means, and the results of this tracking are summarized in Appendix 1.

Our original plan also called for integration of the algorithms developed with a GUI developed by MITRE Bedford that embodied a number of aspects the military crew scheduling problem. This system was also no longer being maintained, and so we pursued the path of developing our own experimental JAVA-based GUI as part of this effort. An important side benefit of this activity was that the GUI tools developed for this project were also successfully used in another scheduling effort involving KIDS-based algorithms, namely the development of a mission planning scheduler for AMC, which is currently targeted for insertion into AMC's ambitious CAMPS program.

Ultimately, instead of using and evaluating these particular development tools and GUIs, our approach has been to consider more broadly how evolving requirements can interact with algorithms and systems development in the process of producing usable software tools. We looked at how, in our particular case, requirements impacted algorithm and data design, and how these things in turn impact the GUI that is an integral part of the software tool produced.

Our algorithm development effort was centered around the use of Kestrel's KIDS environment, but not exclusively so. Loosely speaking, KIDS is designed to help users (really, software developers) apply knowledge of programming constructs to the synthesis of algorithms to achieve very specific objectives specified in formal terms. It is based on a library of rules, definitions and theorems expressed in formal

logic, and is effective in producing instances of abstract algorithms that were defined in that library. However, we wanted to explore some search techniques that were not defined in KIDS theory library, and also to apply some that had not been maintained for some number of years. In order to understand how these alternative techniques would work for our problem, we coded algorithms using them by hand, and then, where possible, implemented similar algorithms using KIDS. Although we are *greatly* indebted to Dr. Douglas Smith and his colleagues at Kestrel for their time and patience, we were also occasionally limited in our ability to produce solutions using KIDS by the lack of staff to maintain KIDS and assist outside users in understanding how to use KIDS effectively. KIDS itself is no longer under active development, and so there was little available time for key personnel at Kestrel to look at extensions and repairs to the KIDS' algorithm library for our problem. Notwithstanding these problems, our experience with KIDS gave us a much better understanding of how to properly characterize requirements, and of how those requirements translate into software, and we were able to successfully generate several versions of the crew scheduling problem using KIDS.

2. Overview of the Crew Scheduling Problem

In this section, we briefly review our understanding of the crew scheduling problem and the nature of the requirements for crew scheduling. Appendix 1 gives a more complete table of the requirements we considered, and their relationship to the algorithms developed.

We began with an general understanding of the Air Force's crew scheduling problem provided by Mr. Murray Daniels of MITRE, outlined here and described in more detail in Section 2. After an initial attempt had been made to produce an algorithm to schedule crews based on this generic model, we visited the 149th Fighter Squadron at Langley AFB, and also an F-16 Squadron of the Virginia Air National Guard. These visits gave us a more detailed view of the scheduling process and motivated a number of the requirements changes that we discuss later in this report.

The basic crew scheduling problem is allocating individuals to fill positions on scheduled flights in a manner that 'optimally' ensures that each crew member is able to participate in flights that include events, which satisfy their individual training requirements. Each scheduled flight is thus associated with a number of events (take-offs, landings, management of particular classes of loads, fighter training events, uses of on-board equipment, etc.). Each crew member in a squadron is qualified to fill particular positions on flights (pilot, co-pilot, instructor, navigator, load-master, etc.). Qualifications are associated with skills, and each crew member must 'maintaining currency' a set of skills related to his role qualifications.

Training requirements that crew members must satisfy come in two basic forms. There are so-called *currency requirements* which are of the form 'do one of these every k days', where k can be different for each requirement. There are also *periodic requirements* where each crew member with a requirement of this type must perform n of those events over a period, typically 3, 6 or 12 calendar months. A typical requirement would be for a pilot to perform some number of landings every six months.

Crew members can only be assigned to positions on flights where they meet the qualifications for that position. *Position qualifications* are both things like basic qualifications to fill a particular crew job, and also qualifications to perform that role in certain kinds of situations. For the purposes of our model, all of the specific qualifications for a role on a specific flight are specified as part of the input, as we describe in the data design.

Crew members may not be available for assignments to positions on flights either because they are assigned to another flight that overlaps in time (where that overlap may be due only to conflicting pre or post flight ground time), or because they are not on-duty, or have conflicting ground duties.

There are also some subtler, more complex requirements on the scheduling process, some of which have not yet been sufficiently addressed in our algorithm designs to date. These include such things as:

- **How to automatically schedule someone who missed a requirement deadline, and so needs to be accompanied on a training flight by an instructor to regain currency.**
- **How best to give preference to crew members who are infrequently available (esp. at reserve and guard units).**
- **How to model constraints stating preferences for crew members to work together (or not) on the same aircraft, or, for single-seat fighter aircraft,**
- **How to model constraints on the relationships between the pilots (crews) of different aircraft (esp. for fighter training, where almost all planes are one-seaters, and there are different training roles for different pilots within a group. Instructor-student relationships for fighters also fall in this class.**
- **How to expand the process to support changes to the sortie flight schedule to accommodate or accomplish training objectives for particular crew members. This issue has the potential to vastly complicate the search process and make the search space grow enormously.**

Most of these more complex issues are ones that force human schedulers to take an iterative approach to filling out the schedule. That is, they treat the process more like a planning problem where they deal with difficult or highly constrained requirements first, effectively scheduling in stages where each stage addresses some different kinds of requirements or constraints. For example, at National Guard fighter wings, where many reservists are only available several times a week, schedulers routinely assign these pilots to times when they are available. They then schedule flights for those who are more frequently available, but may have the same training needs, since (in principle) the full time people can be accomplish those requirements at other times.

Where possible, we have tried to accommodate these difficult requirements by changes to the utility function used to optimize the result, rather breaking the algorithm up into discrete, heterogeneous stages, where the overall solution cannot be 'optimized'. Although we have not been able to adequately test the quality of the solutions produced extensively, the former approach leads to a more uniform treatment of all requirements. However, it seems unlikely that it will be possible to maintain this uniform model and satisfy all of these difficult requirements. We discuss this in more detail below.

Due to time constraints and limitations in the ability of KIDS to construct heterogeneous systems, such as the staged scheduling approach just mentioned, we were forced to push some of these considerations back to the user – e.g., by having the user pre-assign some crew members to flights before the algorithm is run. The last issue in this list – that of changing the flight schedule to accommodate crew training needs, we treated as completely outside the scope of this project, in part because of the many other external factors not available to us that influence when and how flight schedules could be changed (such things as airspace reservations, opposing fighter arrangements with other wings, etc.).

3. Summary of Scheduler Requirements

From the high level description of the problem above, we divided the core scheduler algorithm requirements into a few categories. These impacted different aspects of the development, as we will describe. The categories were:

- **User/Operations Requirements – requirements impacting the use of the algorithm in a tool, including GUI design requirements.**
- **Data Design Requirements – requirements related to definition of domain objects and relations.**
- **Soft Constraints – scheduling preferences that translate into impacts on the utility criteria used.**
- **Hard Constraints – scheduling constraints that when violated produce infeasible results.**

User/Operational requirements tend to guide the overall system architecture, by dictating the relationships between data, user and algorithms.

Data design requirements are in large part driven by the semantics of the domain, but also by the choices of algorithms to be used and their data needs. In this sense, they are a central link in the requirements chain, and are related to and impacted by all other types of requirements.

One can divide up the many of the remaining requirements on schedulers into classes of constraints. Constraints considered by virtually any scheduling or optimization algorithm fall into two broad classes, often referred to respectively as 'hard' and 'soft' constraints. Hard constraints are those which, when violated, produce an invalid solution to the problem, such as a pilot flying a mission for which he or she is not qualified. Soft constraints are preferences that go to the quality of the solution. They may be local criteria, such as the degree to which a crew member needs to be assigned to a flight in order to improve his or her training status, or more global criteria, such as preferences for certain crew members to work together whenever possible.

A reason to distinguish these two classes of constraints when looking at requirement tracking into code is that they tend to impact very different aspects of the algorithms produced. Hard constraints are typically handled as checks on valid assignment of values to scheduler variables during search. Soft constraints must be handled as part of the optimization process. That is, one must search many valid possible assignments to maximize the utility of the overall solution. Thus, soft constraints are generally handled by modifying the criteria that go into assigning a number to the quality of any particular solution. However, this is not always the case. In many heuristic search algorithms, and in many KIDS-based schedulers, one can also handle soft constraints by appropriate use of ordering functions on the sets of possible values to assign to a variable, so that the search will tend to see ones satisfying the soft constraints well before ones that do less well. This may also allow the scheduler to search only part of the solution space, abandoning the 'holy grail' of true optimality in favor of finding a useful answer in a finite period of

time. The decision about how to handle such constraints can therefore involve a tradeoff decision between these two methods, depending on the type of algorithm used.

User/Operational Requirements

Operational requirements relate to the functionality that is directly available to the user. In the case of the schedulers that we have built on this and related projects, this category of requirements covers the need for:

1. **Persistent data storage** – All of the data that is involved in the scheduling process must be stored in a persistent database, which can be viewed, edited and visualized in multiple ways.
2. **User editing of all data and Schedules** – All scheduler inputs and outputs must be changeable by the user.
3. **Visualizations** – Appropriate visualizations of the schedule and crew member requirement status are needed. These are often more specific to the kind of units whose schedules are being developed than the generic data model will support directly.
4. **Rescheduling** – Changes made by users must be observed when revising the schedules.
5. **Scheduler Invocation** – User invokes the scheduler when all inputs have been specified. Appropriate feedback must be provided if the inputs are incomplete or inconsistent.
6. **Data Translation** - Inputs and outputs of the scheduler must be translated to/from their persistent data representations. These different representations must be semantically consistent and the persistent database structures must be equivalent to or a superset of the scheduler input and output data forms.

At this point in time, we have built three different systems using KIDS-generated scheduling algorithms for different projects (ITAS, CAMPS Mission Planner, and the Crew Scheduler), following essentially the same overall system architecture as the original ITAS scheduling system. This general architecture is reflected in Figure 1. Data is stored in a relational database to satisfy Requirement U-1, and data entry screens are developed for those data tables (Requirement U-2). Visualizations are also built that are based on the stored table data (Requirement U-3). The need for rescheduling based on prior schedule information is accomplished by presenting the prior schedule as one of the inputs to the scheduling algorithm (Requirement U-4). The scheduler is invoked from the GUI by the user (Requirement U-5). All scheduler inputs retrieved by queries from the database and translated from their relational data form to the tuple representations used by the scheduling algorithm. The output schedule is then translated back to the database form before presentation to the user (Requirement U-6).

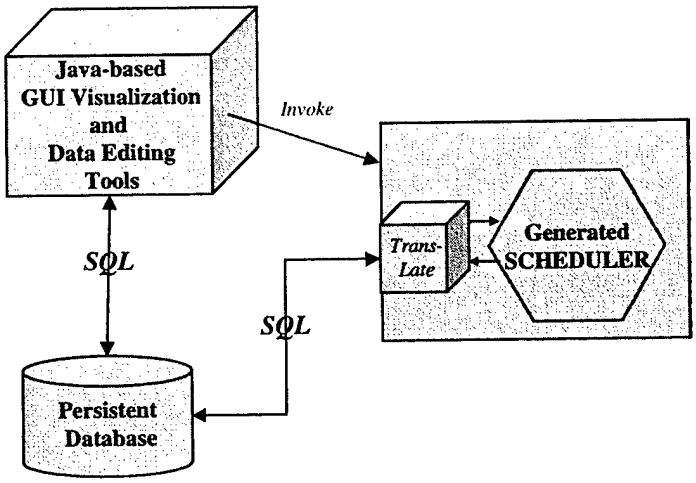


Figure 1: High-level system architecture

This general architecture was simultaneously used to develop the crew schedulers on this project and the CAMPS mission planner scheduling prototype for AMC. In fact, some of the Java-based tools developed for editing and schedule visualization were shared between the two projects. Some of the software supporting the data translation process from relational database form and the input and output forms for the scheduler was also shared. Communication between the database and the other modules was done using JDBC. The schedulers and translation code were developed in LISP.

Data Design Requirements

An early element of the design process is the laying out of the major data elements to be manipulated. This includes representations of all of the domain objects that are constrained together in coming up with a schedule. It is relationships between these objects that define the constraints to be achieved by the produced schedules. Objects to be defined for this domain included: crew members, availability times for crew members, sorties (flights) and perhaps other temporally bounded events (such as ground duties), crew positions on flights, crew member position qualifications and their type definitions, crew member training requirements and their current statuses, requirement type definitions, and crew assignments (schedule elements).

Figure 2 shows the relationships between the kinds of information involved, described as conceptual entities in a semantic diagram. This kind of representation of the domain ontology is extremely helpful in moving from a non-technical requirements view of the task to the definition of specific data types. In this diagram, we see explicitly the relationships between entities that are to be evaluated in constraints (the *satisfies* relations), those that are to be generated (the *assigns* relations), and the ones to be maintained (the *updates* relation). The *achieved-by* relation is, in effect, the rationale for the *updates*, and does not get carried through subsequent design refinements.

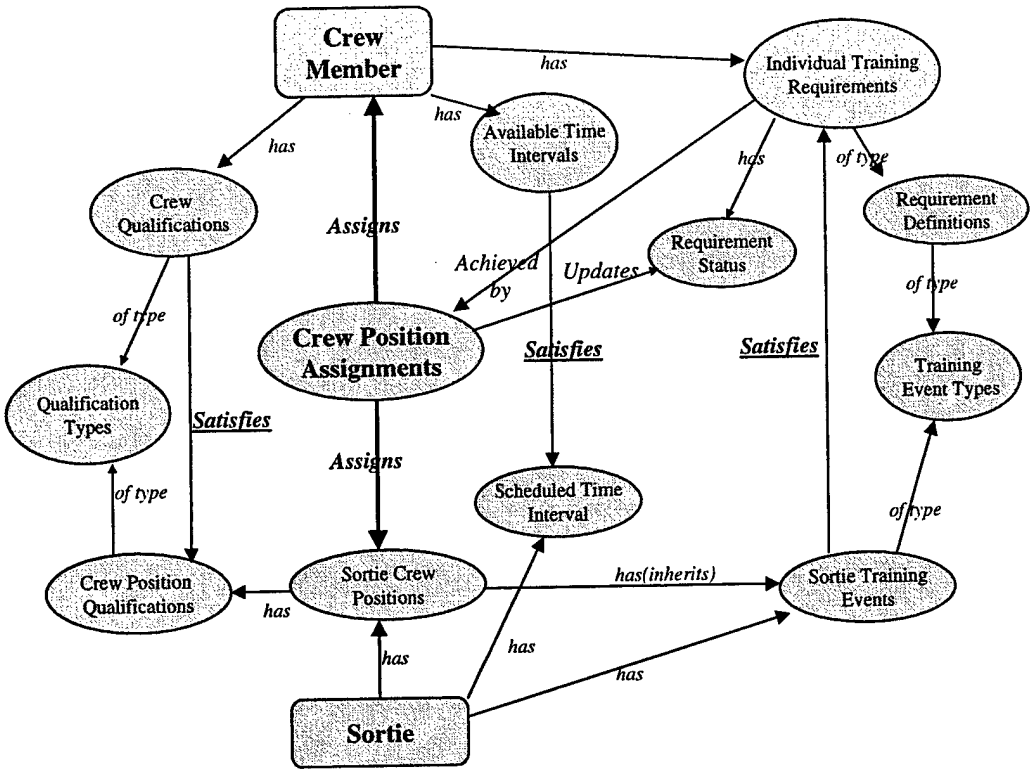


Figure 2: Semantic relations between crew scheduling domain elements.

This diagram is not itself a database or datatype design, but characterizes many of the important relationships that go into the data design. In fact, it is the basis for two distinct data models in our crew scheduler system. One is the persistent, relational database design, and the other is the more object-oriented data model used by the algorithm. This semantic model does not make explicit choices about some of the data design elements, such as when attributes are stored in the same object or table, although *has* relations that are for one-to-one relations will typically be collapsed into the parent object, while many-to-one relations will be maintained distinctly.

Figure 3 shows the organization of the major data elements as relational tables. This view reflects the form in which data is stored in a persistent database. Similar to the way a physical data design is abstracted from an ER diagram, this view can essentially be derived from the semantic diagram, by explicitly collapsing information into tables based on the arity of the relationships involved. In all of the schedulers that we have build using KIDS to date, we have maintained this partition between the model used for persistent store and the data types manipulated by the algorithms generated. This is a common distinction in many kinds of software systems.

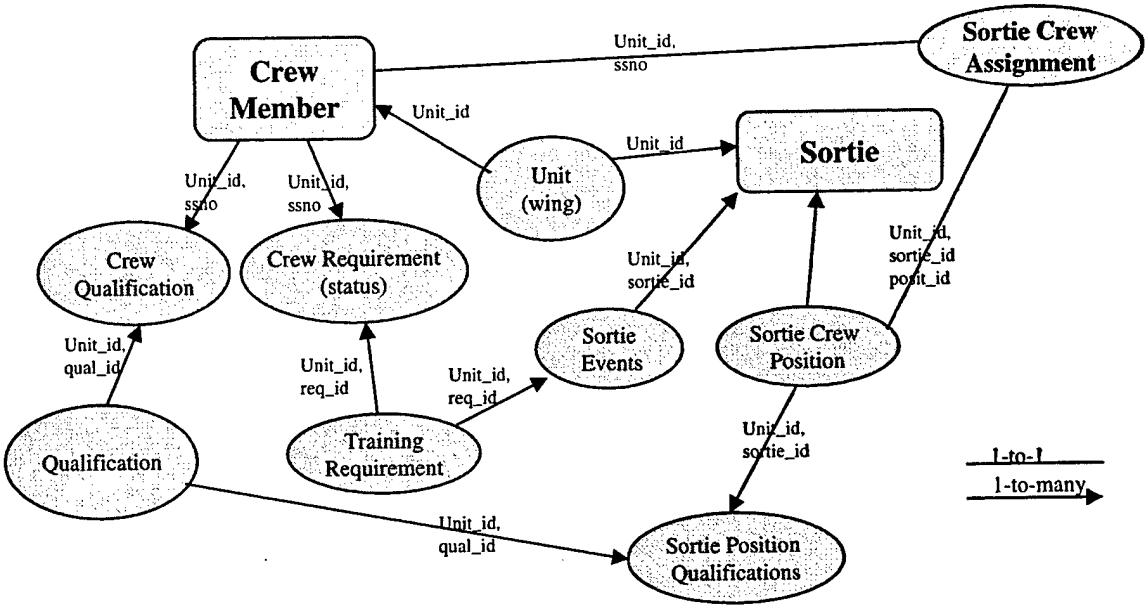


Figure 3: Relational Database Organization

The semantic model is also the basis for the object model used by the search algorithm. This form, also largely derived from the semantic relational model, is also influenced by algorithmic efficiency criteria, and the programming language constructs available. When using KIDS, data design is still left up to the developer, but interacts strongly with the definition of the constraints to be placed on the scheduling algorithm to be developed. Figure 4 shows some of the datatypes, as expressed in REFINE, the programming language used in scheduler algorithm design and generation. It is in terms of these datatypes that the developer describes constraints. These datatypes (or, really their LISP translations) are the ones used by the generated schedulers.

It should be clearly understood that there is not a one-to-one mapping of the database model onto the REFINE datatype model. Database field types must be transformed (for example, dates turn into integers representing minutes from a starting point), and data from multiple tables must be synthesized into complex objects with multiple sub-objects (really, sets of tuples or mappings stored as attributes). To support this process, the developer must implement translations between the tuples retrieved from the external database using SQL and the internal scheduler datatypes. This process is implemented by hand using LISP or REFINE at this point in time. On another project, BBN and Kestrel, along with Yale University are exploring ways to automate this part of the development process.

Whenever a change to the scheduler requirements causes a change to the algorithms datatype structures, this must be reflected both in the database and in the translation process to and from the database as well. This is a key issue for requirement-based development, and is discussed in more detail later.

```
% These are the people to be assigned to crew positions on sorties.

type CREW-MEMBER =
  tuple(id : crew-id,
        last-name : string,
        first-name : string,
        qualifications : set(qualification),
        requirements : set(requirement-id),
        initial-status : crew-req-map)

% This is their requirement status (for time T0)
type REQUIREMENT-STATUS =
  tuple(date-last-accomplished : day,
        number-accomplished : integer,
        end-of-current-period : day)

type CREW-REQ-MAP = map(requirement-id, requirement-status)

% status of CREW-MEMBER on all REQUIREMENTS
type crew-requirement-status = map(crew-id, crew-req-map)

% There may be some number of crew members filling a type of role.
type CREW-POSITION =
  tuple(position-type : crew-position-type,
        position-index : integer)

type POSITION-DESCRIPTION =
  tuple(id : crew-position,
        required-p : boolean,
        necessary-qualifications : set(qualification),
        excluded-qualifications : set(qualification) )

type SORTIE =
  tuple(id : sortie-id,
      mssn-type : symbol,
      start-time : time,
      end-time : time,
      sub-events : set(requirement-id),
      positions : seq(position-description))

% this is the basic reservation triple,
type reservation =
  tuple(sortie : sortie-index,
        position : position-index,
        crew: crew-id)

% This is the scheduler output data structure
type crew-assignments =
  map(crew-id, seq(reservation))
```

Figure 4: Major data type definitions in REFINE.

It should also be noted that in KESTREL's newest environment, PLANWARE, some of this data type definition process is handled much more automatically, by refinement of abstract functional datatypes for the classes of algorithms being generated. Even there, however, the user is responsible for defining how the

domain attributes involved relate to these abstract functional data categories. Semantic interoperability with external data sources is not addressed.

Hard Constraints

In the category of hard constraints are all of the constraints that *must be* adhered to in the final schedules produced. For the crew scheduling problem, this includes most of the temporal constraints, and also constraints involving valid scheduling assignments.

The temporal constraints to be observed include:

1. Crew Member Availability – crew members can only be assigned to flights when they are available.
   - This applies primarily to reserves, who are only available in specific time windows each week.
   - Senior Squadron personnel may have ground duties that supercede flying.
   - In general, times for other ground duties and leave time must be known.
2. No crew member can be on two flights at the same time, or for a period before/after each flight.
3. Instructors must fly at the same time, as uncertified crewmen.
   - For cargo planes, this requirement is to fly on the same plane. For fighters, it is a requirement to fly in the same group of planes during a training exercise.

The qualification constraints are:

4. Each crew member assigned to a position must be qualified for that position.
5. Each crew member assigned to a position must not have a qualification excluded from that position.

One design decision that was made to simplify the problem was to lump all kinds of availability information that did not pertain to the flight schedule into one class of data, namely, information about the set of time intervals that each crewman was available during. This simplified our consideration of this constraint as far as algorithm development goes, but might not be the most convenient way for users to encode that information. For example, in a fully developed system, it might have made more sense to have several kinds of availability information, relating to normal on-duty time, and specified other duties. This information would then have to have been translated into periods of availability for each crew member.

Figure 5 shows the high level functional specification that was the basis of the KIDS algorithm development. It explicitly refers to the set of input and output data types involved, and specifies the hard constraints that must be satisfied for inputs and outputs in the algorithm produced. These are *consistent-crew-qualifications*, *consistent-crew-flight-separation*, and *consistent-open-positions-filled.* The first one, stating that each assignment must satisfy the necessary and excluded qualification criteria, is shown in Figure 6. The second is the constraint that all flights by each crew member must be separated in time. The last condition is that all open, required positions on all flights are filled. This last constraint is necessary for KIDS to develop an algorithm that finds a complete solution.

```
function CREW-SCHED-OPT
  (crews : crew-map,
   init-status : crew-requirement-status,
   sorties : seq(sortie),
   open-positions : seq(position-reference),
   prior-assignments : crew-assignments)
  : crew-assignments   % returns this type
% input conditions
 |  positions-refer-to-sorties(open-positions, sorties)
    & initial-status-for-crews(init-status, crews))
  returns
  (assignments : crew-assignments
% solution conditions
          | extremal(assignments,
      lambda(a1, a2)
           (sched-utility(a1, init-status, crews, sorties)
            <=
            sched-utility(a2, init-status, crews, sorties)),
      {assigns | (assigns : crew-assignments)
        consistent-crew-qualifications(crews, sorties, assigns)
      & consistent-crew-flight-separation (sorties, assigns)
      & consistent-open-positions-filled(open-positions, assigns)

      & consistent-position-usage(assigns, open-positions)
      & consistent-crew-usage(crews, assigns)
      & consistent-sortie-usage(sorties, assigns)})
  )
```

Figure 5: Algorithm I/O specification for optimizing version of KIDS scheduler

The CREW-SCHED-OPT function description is the goal for algorithm generation within KIDS. It embodies a number of assumptions used by the KIDS theorem prover, such as the input conditions that the indices (integers) of a set of open positions refer to positions in the sorties to be filled, which are described separately. It also states that the status of each crew member is given by the INIT-STATUS variable. Among the output conditions are similar statements that the crews and sorties referred to indirectly in assignments are from the input sets, and that new assignments are filling open positions.

In Figure 6, we see the definition of a typical hard constraint, referred to by the function template of Figure 5. This constraint, that crew members must be qualified to fill the positions they are assigned to in reservations turns into a test on possible variable assignments (of crewmen to positions). The form of the main constraint is as a test on a complete solution, and states that all assignments must pass this test, but its incremental form is shown as CONSISTENT-CREW-QUALIFICATIONS-FOR-RESERVATION, and tests whether a new reservation binding a crew member to a position satisfies the test. It is a variation on this latter form that appears in the synthesized code.

```
% A crew member assigned to a sortie position must have the qualifications to fill that position.

function CONSISTENT-CREW-QUALIFICATIONS
     (crews : crew-map,
        sorties : seq(sortie),
      assignments : crew-assignments) : boolean
= fa(res : reservation, res-seq : seq(reservation))
  (res-seq in range(assignments)
   &
   res in res-seq
   =>
   consistent-crew-qualifications-for-reservation(crews, sorties, res))


function CONSISTENT-CREW-QUALIFICATIONS-FOR-RESERVATION
  (crew-members : crew-map, sorties : seq(sortie),
   res : reservation) : boolean
=
  sorties(res.sortie).positions(res.position).necessary-qualifications
  subset
  crews(res.crew).qualifications
 &
  empty (
    sorties(res.sortie).positions(res.position).excluded-qualifications
    intersect
    crews(res.crew).qualifications))
```

Figure 6: A hard constraint – a crewmen must be qualified for the position to be filled.

Other hard constraints are similarly represented. The other one appearing directly is the test for CONSISTENT-CREW-FLIGHT-SEPARATION. This ultimately is a test that a reservation does not violate the constraint that two assignments for the same crew member are separated by a minimum constant time.

```
function CREW-FLIGHTS-SEPARATED-P
  (sorties : seq(sortie), reservations : seq(reservation)) : boolean
   = fa(i : fixnum)
  ( i in [1 .. (size(reservations) - 1)]
   => sorties(reservations(i).sortie).end-time
     <=
     sorties(reservations(i + 1).sortie).start-time
     - *crew-flight-separation-time*)
```

Figure 7: Incremental test for separation of flights by one crew member.

This test relies implicitly on the existence during search of an ordered list of reservations for each crew member, so that adjacent reservations can be tested for this condition. This data structure must be maintained incrementally if the search is to be carried out efficiently. KIDS does not do this automatically. It requires that the developer write the function that inserts a new reservation into the schedule so that this property is maintained, and then support KIDS by supplying appropriate axioms so that it can prove that it maintains the sorting properly.

We discuss the third major necessary constraint, namely, the availability of a crew member for an assignment, later in this report, as an example of pushing a requirement through to a code change.

Soft Constraints

In the category of soft search constraints are all of the objectives for the produced schedules that specify preferences for certain assignments over others. In virtually all optimization algorithms, these criteria must all be reduced to a single number that rates the goodness of one schedule over another, and embodies all tradeoffs between conflicting soft preferences. In this domain, the primary soft constraint is the goal to get the most urgent training done by each flight, urgency is based on a crew member's need to maintain certifications. It might seem like there is really only one primary optimization criteria, since it all comes down to how soon is the due date for each requirement for each crew member. But there are a large number of requirements for each crew member, so there is still a choice to be made about combination functions that produces one number when assigning a crew member to a position (i.e., is each requirement to be treated equally?), and another that produces a score for each whole schedule considered, so that the most training is accomplished for the squadron as a whole.

In developing an optimizing algorithm, we chose initially to use the simplest possible combination functions. That is, the urgency of a crew member filling a position is computed as the sum of the urgency of that crew member satisfying all requirements that could be satisfied by his or her assignment to that flight. A score for the overall schedule is the sum of these across all crew members. If necessary, it would be a simple matter to provide weights for each requirement (an additional attribute of each TYPE of training requirement) if that helped improve the accuracy of the generated schedules. But that would not be sufficient if certain requirements were more important for some crew members than others (say, for junior pilots versus experienced ones). Further knowledge acquisition would be required to reveal some of these subtleties.

There are also several others kinds of preferences lurking about that might need to be included in a utility function for a successful application. For example, there may be "hidden" preferences for certain people to be scheduled together. To achieve this, it might be necessary to develop a score for the cohesiveness of crew members on the same plane, and then somehow combine this with their need to be there for training purposes. Human schedulers normally try to maintain balance between a number of conflicting criteria that may not easily be reconciled and reduced to a one-dimensional utility value that properly reflects the tradeoffs involved. They do this in part by only considering training urgency when a due date is approaching, and considering other factors as decisive when training urgency is low.

From this discussion, it is easy to see why crafting a utility criterion that satisfies domain constraints is a hidden art of scheduling systems. In the end it comes down to an empirical question – can an appropriate parameterization of the problem be arrived at that produces what humans would regard as good schedules? This question remains much the same whether one uses constraint-based techniques or linear programming methods. We have not devoted as much time on this project to answering this empirical

question for the crew scheduling domain as would be necessary to field the scheduler successfully. However, we did experiment with a couple of different approaches, as described below.

First, we developed a plausible utility function that computes scores for all requirements, and show how the need for information to support this scoring function influenced the data design above. As we stated at the outset, we began with a generalized data model provided by Murray Daniels of MITRE, and his description of the general form of training requirements:

Flight Crew Members all have a number of requirements that must be maintained over time. Requirements generally correspond to "training events" that occur during flights (e.g. Low-altitude flying, Night Landing, AR, ...)

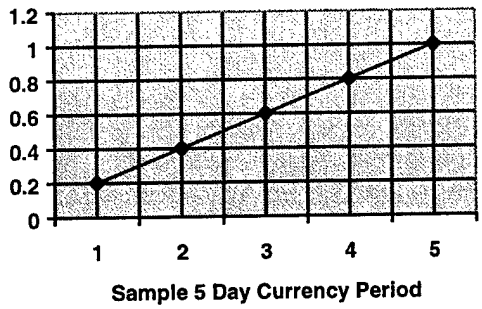
- ***Periodic* requirements have deadlines at the end of every calendar quarter, half-year or year. Some number of training events must occur in this time. (e.g. 15 flights of a particular kind every 6 months).**
- ***Currency* requirements are just time-limited (e.g. at most 15 days between flights). The clock restarts after each flight that satisfies the requirement.**

Crew Members must maintain currency in all roles they are qualified for. For example, Instructor Pilots and Evaluation Pilots must maintain currency as Pilots. Pilots are also CoPilots.

A currency violation requires some number of training flights with an instructor.

- **On cargo (multi-position) aircraft, the instructor flies in another position on the same aircraft.**
- **On most fighters (single seat, with a few exceptions) , the instructor must fly in an accompanying aircraft.**

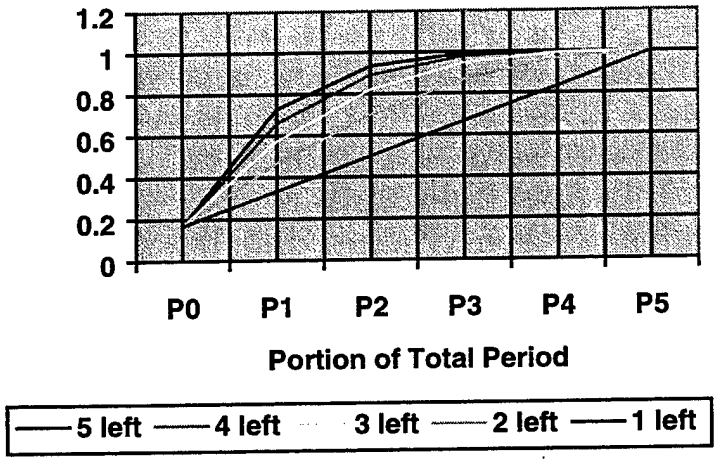
To develop an overall utility for a given assignment, we sum up the scores of how urgently that crew member needs to satisfy each of his requirements that correspond to training events scheduled for the flight in question. For currency requirements, which require one training event every **d** days, where **e** days have elapsed, and the weight to be attributed to this factor is **w**, the urgency is computed by the function shown in Figure 8. This function starts at 1/d the day after that requirement was last accomplished, and rises linearly to an urgency of 1.0 (or **w**, if **w** is not 1) the day the requirement period is to end. Later, we discuss what happens after a requirement is missed.



currency-utility(**d, e, w**) = w * (1/d + ((d – 1)/d) * e/d)

Figure 8: Graph of currency utility over time period

For periodic requirements, in which some number of flights satisfying a requirement must be flown every quarter, six months or annually, we at first tried a polynomial that reduces to essentially the currency utility when only one flight must be flown. For a period of **d** days, with **r** required events in the period, and where **e** days have elapsed, and **a** events accomplished, and letting k = r/d the score used is:



Period-utility(**d, e, r, a, w**) = w * (1/k + (1 – ((d – e)/d)$^{(r-a)}$)

Figure 9: Graph of original periodic utility over time period

We discovered, however, that this tended to exaggerate the importance of periodic requirements relative to currency requirements, which, given the relatively long amount of time to accomplish them, was more the reverse of what was needed than not. We then changed to a model that treated the periodic requirements more like a sequence of currency requirements, spaced out evenly over time. This is shown below. Basically, the urgency of the periodic requirement drops to zero after each one is done, if you are less than number-done/number-needed proportion of the way through the period.

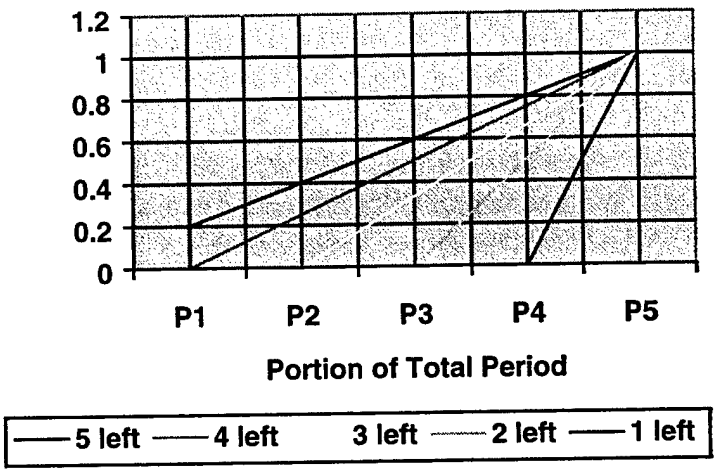


Figure 10: Graph of revised periodic utility over time period

We allow each type of requirement (that is, the requirement to fly some event) to be either a periodic or a currency utility or both. If one doesn't apply, then its urgency is zero. The total urgency to fill some requirement on a flight is computed as the **maximum** of the currency and periodic utility for that event. Since most requirements are either one or the other, this just means no single requirement is double counted.

The priority of assigning a crew member to a flight is currently computed as the sum of his/her urgencies to accomplish all events scheduled on that flight. This combination function is somewhat problematic, as it may mean that urgent near-term requirements are outweighed by crew members who can satisfy several requirements simultaneously on a flight, leading to more deadline violations than desirable. We are still looking at alternative formulations of this utility combiner, such as the **maximum** of the individual requirement utilities, a weighted average or a sum of squares, each of which would improve the visibility of urgent requirements. Although using maximum might seem appealing at first, it has the flaw that there is then no weight given to satisfying multiple requirements on one flight, which clearly is a cost and efficiency savings. Using less linear individual costs so that each requirement's utility is low until near the due date is another approach that has been suggested, but not yet tried. This can be done either by using multi-segment linear models or by quadratic formulas, such as the sum of squares approach to utility combination.

The overall utility of a schedule is the amount that everyone's urgency for all of their requirements at the end of the schedule is reduced by the assignments they are given during the scheduling period. The goal for the optimizer is to find the schedule that maximizes schedule utility, given that all of the hard constraints of crew availability and qualification are met.

Open Requirements Issues

Some open issues remain with the current formulation of the (especially soft) constraints as described above. As just mentioned, nearly due requirements, which are perceived as critical to human schedulers are not sufficiently weighted in the schedules we have produced so far.

Fliers "in training" or who are overdue on a requirement must go with instructors for flights satisfying those requirements. This needs to be handled as a separate phase of scheduling, preceding the scheduling of other participants, for several reasons. First, although our requirement utility functions have values exceeding 1.0 when a requirement is overdue, it is not correct to say that these requirements indicate the crew member should be scheduled in preference to those whose requirements are approaching but not exceeding their due dates. Once a requirement is overdue, it no longer has the same *kind* of urgency, since the crew member must be accompanied on that kind of mission. It is more correct to say that crew members who have overdue requirements are unavailable for those kinds of missions, unless explicitly scheduled with instructors. Our current solution to this requirement is to allow the human scheduler to schedule such assignments manually, prior to the system automatically scheduling crew members who just need to maintain currency on their requirements.

Another issue that is a requirement for schedulers at reserve units particularly is that many pilots are only available a few hours a week, and so must be scheduled within those time periods. To handle this, human schedulers will typically schedule these people before those who are available most of the time. For an automated system, it is preferable to consider this kind of limited availability as sharpening the urgency of assigning these people when they are available. We are still considering how to modify the utility function to get this effect. Our current approach is to weight the urgency of these crew members by the fraction of the time they are available during the schedule period. However, we have not included this mechanism in the scheduler delivered on this project.

A final unresolved issue is a much bigger problem for scheduler automation generally. Human schedulers do not just schedule positions on existing flights, they adapt the flights to fit the needs of those being scheduled, or to address the general training needs of a group over time. In some settings, this amounts to setting up a long-range 'syllabus' of training that emphasizes one kind of training one week, and another kind another week. Within this they also routinely assign training events to flights based on the current needs of their crewmembers.

We have chosen for this project to maintain this as part of the manual process of setting up the flight schedules prior to running the crew assignment scheduler for several reasons. First, it is a process highly dependent on a number of external factors including what aircraft are available, what kinds of missions can be run on a given day based on things such as requirements to perform real missions during which training can occur, availability of adversaries from other squadrons, availability of appropriate airspaces for

conducting particular kinds of training missions, etc. Human schedulers look opportunistically at all of these kinds of conditions as well as crew member training needs when laying out a flight schedule.

Second, the process of designing a curriculum to maintain the currency of a set of pilots or other crew members over the course of a year only interacts weakly with the crew assignment process. There are probably things that could be done to give the scheduler options about what kinds of flight events occur on a given flight, or to adjust the times of flights somewhat to accommodate crew member availability, but all of these activities change the basic nature of the search required, and we did not have the time or resources to pursue these directions during the course of this one year project.

Perhaps a more practical solution to this problem is to provide the human schedulers with tools to rapidly identify short term needs or pending overdue requirements as they build the flight schedules, so that appropriate opportunities for crew assignments are available to the automated scheduler. If these opportunities are for specific crewmen, then the human scheduler should be given a mechanism to suggest or enforce that particular slots are reserved for particular crews as they are building the schedule. In any case, better interactive tools for building the flight schedules should be designed to work hand in hand with the crew assignment scheduler. As our main focus on this project was algorithm design and development, we did not pursue this during the limited time available.

4. Development Approach

We followed a two-tiered strategy to algorithm development on this project. After the initial requirements elicitation period, we began developing simple versions of our scheduler by hand in LISP in order to understand better how the requirements translated into code. After each attempt to build an algorithm by hand, we went back and developed a comparable algorithm using KIDS, much as the original KIDS developers tended to do as they were developing the initial library of synthesis routines. This approach was in part due to our initial ignorance about what was available in KIDS and later to limitations in the KIDS algorithm library. Figure 11 shows the progression of algorithms developed. Our algorithms then became requirements for extensions (albeit minor ones) to the KIDS system, in order to support the kinds of products we were looking for. These extensions were generally the addition of missing or underspecified logical rules needed to make a particular kind of automated reasoning go through. These extensions required assistance from Kestrel staff, mostly from Doug Smith or Stephen Westfold.

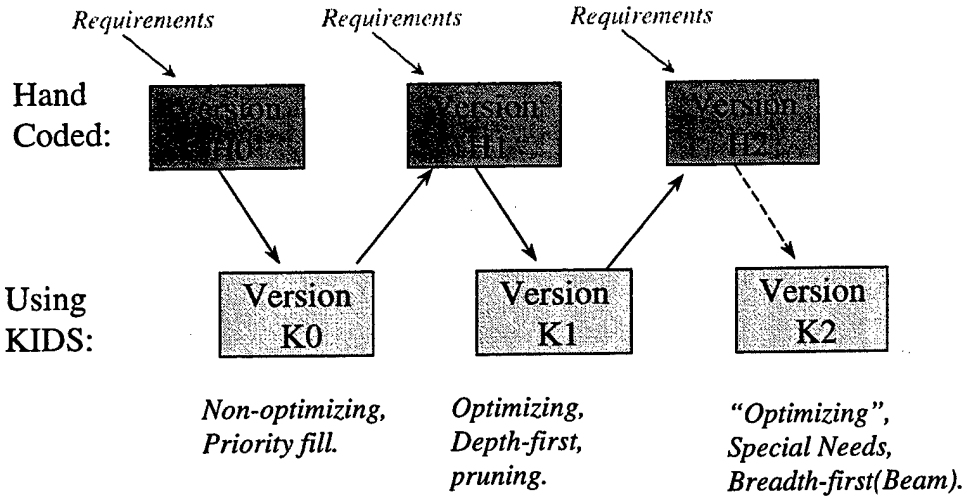


Figure 11: Progression of algorithms developed

Over the course of the year, we developed three very different algorithms plus several minor variations by hand, and two different algorithms using KIDS. Our 'Version 0' schedulers were based on the model used in ITAS. These were non-optimizing algorithms that essentially looked for a solution satisfying the necessary requirements for each position in temporal order from the first flight to the last. During this pass, most of the data structures and the necessary constraints were designed and modeled, and an initial cut at an interface was developed. We also developed more familiarity with the use of KIDS.

The 'Version 1' schedulers were optimizing search algorithms, that would search the space of feasible solutions (ones satisfying the hard constraints), while trying to be efficient by pruning parts of the space where less optimal solutions were found. Several handwritten variations on this algorithm were produced

that would not search the whole space, but would, for example, only search the n best branches at any particular point in the search, while still pruning less effective solutions. This turned out to be necessary even for very small problems, such as scheduling one day with less than 20 flights where some 30-50 crew members were available to participate in those flights. As the number of branches explored went from 1 to 5 or more, times grew exponentially from less than 1 second to several hours – just to do one day's worth of schedule.

Unfortunately, it was difficult to reproduce this variation on our version 1 hand algorithm using KIDS, in part because of the difficulty in expressing the concept of 'heuristic optimality' – where you are not really able to prove that you've found a good answer logically. We are still discussing with Kestrel how such approximate solutions should be expressed as objectives.

The final version that we produced for the search algorithm is a variation on the heuristic search technique used at CMU in such systems as DITOPS. It is called a 'beam search' and is a limited-width breadth-first as opposed to depth first search, which all of our prior algorithms had been. We did not have time to explore the generation of this algorithm using KIDS, although it is certainly possible in theory to do it, and KIDS has been used to produce breadth-first searches. Again, however, since it is a heuristically bounded search, it is different than the algorithms that KIDS normally produces.

Algorithm Statistics

As the architecture diagram (Figure 1) illustrates, there are pieces of the system that are written in different languages. The Version 0 systems were all LISP-based, including the GUI, but the search algorithm developed in KIDS is actually generated in the REFINE language, and then 'compiled' into LISP. The LISP code supporting the persistent database information was approximately 5000 lines. Another 800 lines of LISP code, much of which was provided by Kestrel, implemented the REFINE language constructs used in LISP. Approximately 2000 lines of code supported the data translation between persistent database formats and the REFINE datatypes used in the search algorithm. Another 500 lines of LISP code implemented the utility functions used in the optimizing search algorithms, although the original versions of these were hand-written in REFINE.

For version 0, the LISP-based GUI was approximately 11,000 lines, while the version 1 JAVA GUI was 23,000 lines (of course, one should not compare 'lines' of JAVA and LISP directly, as JAVA is a much more verbose language). Although we did not dwell on detailed interface requirements, clearly the amount of effort that can be required is substantial for this aspect of a system. Of course, there are also commercial GUI development tools that can substantially reduce the burden of this aspect of development. Furthermore, much of the code above (except for the GUI and data translation subsystems) is shared across several scheduler projects.

As for the scheduling algorithms themselves, which was our main focus on this project, the Version 0 REFINE 'theory', containing all of the data structure definitions, constraints, and hand written refine functions supporting the search algorithm, was about 1000 lines of REFINE, and generated an algorithm of about 600 lines of REFINE (of which approximately 500 were directly present in the original theory file). This in turn produced about 600 lines of very densely packed, unreadable LISP code after being 'compiled'. The Version 1 theory was approximately 3500 lines, producing an algorithm of about 1000 lines of REFINE.

For the hand written LISP Versions 1 and 2, the system used much of the same shared code, but replaced the core search algorithms, each of approximately 3-500 lines.

5. Tracking and Translating Requirements into Code

A key issue for this project is the examination of the relationship between requirements and software changes. Looking at the requirements summary in Appendix 1, several things in particular stand out clearly. First, different requirements impact different aspects of the system. With respect to scheduling algorithms specifically, the distinction between hard and soft constraints is crucial. Hard constraints impact the search mechanism by introducing pruning rules on variable assignments. Soft constraints can either impact the utility function used to search for optimal or near-optimal solutions, or, if handled heuristically, the order of consideration of variables and their possible values.

Second, *the requirements that impact the data design impact most pieces of the system.* For example, to add the requirement that crew members could only be scheduled when available, we had to add a mechanism for describing their availability to the system, add a database table to hold that information, add a mechanism to translate that information into a new scheduler input, and add a constraint to the scheduler's search that ruled out assignments that overlapped the unavailable periods. More generally, we observed the following pattern of influences between requirements and system changes (Figure 12).

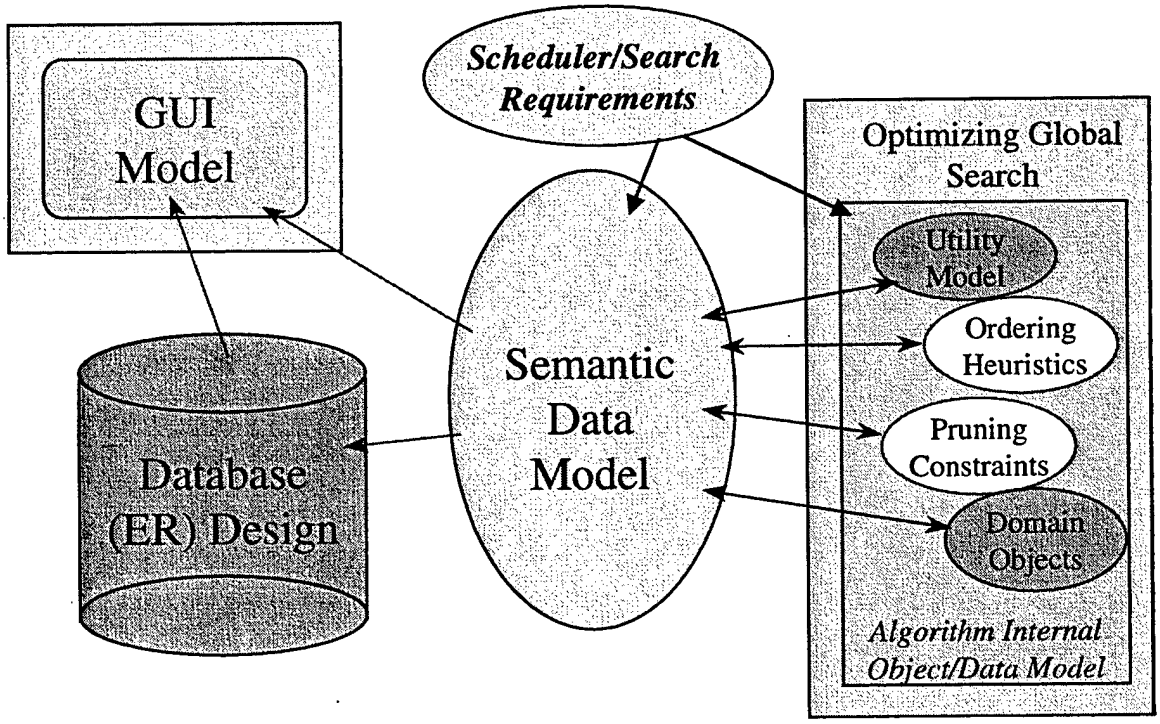


Figure 12: Flow of Relationships from Requirements to system components.

In the figure, we see that scheduling requirements translate into ordering heuristics, pruning rules on possible variable assignments at nodes in the search space, and/or aspects of the formulas used as part of the overall utility calculation for a schedule. Each of these typically depends on some amount of

information supplied as input to the scheduler, and this in turn affects the data design. Given the way we organized the system, the data design is used both for a database design and, separately, in the design of the data structures used as actual inputs to the scheduler. The database design is in turn both influenced by and influences the GUI design, due in part to the USER/SYSTEM requirements for data persistence and edibility.

Let's now consider in a little more detail an example of a new requirement being added to the system. The example we will use is the one mentioned above, that of adding a mechanism for limiting crew availability times. A general statement of this constraint is: "Don't use crew members that are not available for reasons other than they are flying." To incorporate this constraint into the system, we must first examine its impact on the core scheduling algorithm. As it is to be treated as a hard constraint, its impact is on the process of selecting candidates for assignment at each node in the search. When using KIDS, this is done straightforwardly by writing a new constraint on the possible assignments in the final schedule. The constraint is first written as a global constraint on the schedule, and then an additional rule is written to describe how this constraint impacts each new assignment, so that KIDS can infer the relationship between the node constraint and the output constraints. In a hand-written scheduler, the implementer must find the portion of the code that filters the possible assignments and modify that code.

In order to write the constraint, one must first define it in terms of input data. This means extending the set of inputs to the scheduler, with a set of tuples for the availability data, and describing the form of these tuples. The first step is to extend the semantic data model to include availability information, by a relationship like (AVAILABLE <crew member> <time interval>). This translates into a set of scheduler input data tuples in that reference a crew member by their identifier (a SSNO), and two times for the interval beginning and ending times. Given this semantic model and a form for the crew availability data type in the scheduler, the constraint can be written.

The second thing to be extended is the persistent database model. In this case it means defining a new data table, related to crew members, to hold the interval beginning and ending times. Correspondingly, the data editing capability in the GUI must be extended to have a mechanism for entering and editing this data (and, ultimately, disposing of old data, if necessary). Finally, code must be written to retrieve this availability information from the database when the scheduler is invoked, and translate that into a set of input tuples to the scheduler that is generated with the additional constraint and input parameter.

1. Represent semantic relationship and abstract (KIDS) scheduler data type:
   <availability> = map: crew member -> <time interval, avail/not>*
2. Represent DB model:
   Define Table: CREW_AVAILABILITY
   Fields: SSNO, START_TIME, END_TIME, AVAIL?
3. Extend Data reformulation of DB -> Scheduler Data Model
4. Extend GUI to support Entry/Edit of new information

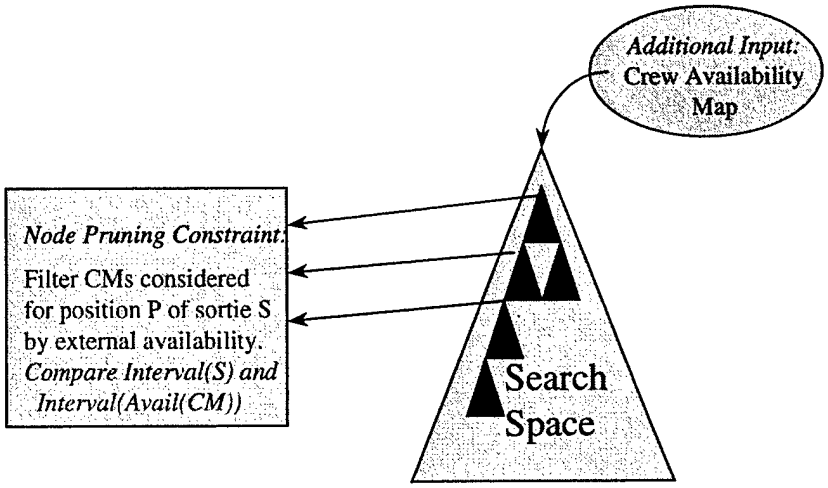


Figure 13: Relationship of Hard Constraint to Search Algorithm.

Figure 13 shows schematically the impact of the new constraint on the search. A new input to the scheduler is defined that is used to consider tests on the viability of candidates for assignment at each point in the search. To provide a developer assistance in understanding how a new constraint impacts a scheduling algorithm, there would need to be a mechanism to identify the *type* of the constraint. In this case, we have a HARD constraint. The system might then provide assistance in translating that constraint into formal terms (as a constraint on the scheduler output for KIDS, or as a piece of code to be inserted at a specific point in the search algorithm otherwise).

Soft constraints are harder to place properly as pieces of code in the kinds of algorithms we have produced, because they may result in different kinds of tests. If (and this can be a big if – see earlier discussion) they can be formulated cleanly as independent contributors to a global schedule utility calculation, then it is not so complicated. Then, as with hard constraints, the impact is localized to a particular place in the algorithm that computes that utility value for each schedule (or partial schedule) generated. However, such calculations are most easily dealt with if they are incremental. That is, given a partial schedule and a new reservation, what is the *incremental* utility of the new reservation? When the utility function can be represented in these terms, the generation of the search mechanism is greatly simplified. When it is not the case, and the calculation of utility must be viewed with respect to the schedule as a whole, then only certain kinds of algorithms will work.

We did not consider algorithms based on truly global utility calculations in our design process, although another group at BBN has been looking at a similar kind of scheduling problem using genetic algorithms (GAs), and GA-based schedulers are in the class of search algorithms that ONLY look at complete solutions. For genetic algorithms, soft constraints are easier to add than hard ones, just the reverse opposite of the class of algorithms that we were working with.

Earlier, we described some of the considerations that went into designing the utility functions that we used in our scheduler. Many of the same considerations must be weighed each time a new soft constraint is added, and the whole scheduler must be examined empirically, to see if the schedules being produced make appropriate tradeoffs. Take the previous example of adding crew availability time as a constraint. One could extend the impact of this information to satisfy the (still open) requirement that crew members who are only available a very limited amount of the time be selected over others who are available essentially all of the time. The "easiest" to do this is by a creating *global preference, considered as an optimization factor*, by design of a utility function that weighs this factor together with the other needs of the crew members to satisfy their training requirements. Since it impacts all training requirement utilities, it would most likely be done by weighting the total urgency of any particular crew member filling a particular seat on some mission. But then one must look at a large number of cases to determine what a reasonable weighting scheme might be: Does someone who has an urgent need (approaching 1.0) get preference over someone who has only a weak need (urgency .33) but is only available one third of the time? Or is something more complex required? Where is the "break even" point? We have not had the opportunity to discuss this particular issue with domain expert schedulers, and it is unclear whether they would be able to answer the question in this form anyway. Only by a detailed case analysis might we come up with a reasonable heuristic for this. What they do right now is schedule people who are seldom available first, but we would need to understand more clearly how seldom is seldom enough to get this treatment.

Another way to go is to implement this more closely to the way it is done by hand, that is by a multi-pass scheduling system that takes the special cases first. Were we to do this, then the impact of such a requirement would be to introduce a new pass for scheduling limited availability crew members. If the system were already designed for this, then the change would be relatively simple and localizable, although the localization would be to a search using a different set of inputs. A question that would arise in this case is the order of placement of this particular new scheduling pass to others that came before it was. For example, if the scheduling of pilots with instructors was also done as a pre-pass, then would limited availability crew scheduling precede it or follow it? And what should be done with limited availability crew members who needed instruction?

The final issue with respect to the choice of a multi-pass search versus a single pass approach that uses an overall utility model is the problem that one cannot guarantee any kind of 'optimality' or even near-optimality with the multi-pass approach. We tend to believe that this is not particularly a problem, as even when there is one global utility function, there are senses in which conflicting considerations cannot be 'optimized' together. Ultimately, the decision comes down to what experts see as a 'good' schedule. This is almost never just along one dimension of utility. The answer to this question depends on whether the schedules produced are both effective and understandable to the users.

6. Discussion and Conclusions

This project has considered the tracking of requirements for and the generation of a class of scheduling algorithms for flight crew assignment, using both hand-written and automatically generated search algorithms. Due primarily to our use of KIDS, we focused on global search algorithms, as opposed to local search algorithms that work by considering a population of schedules, and modifying them to improve their overall utility (a crew scheduling system based on genetic algorithms, a local search technique, was also recently developed by BBN). However, we have seen some principles emerge that would apply equally to that class of schedulers when used in particular applications.

One of our main observations is that some broad user and system requirements for data management, visualization and editing motivate the separation of data supports (DBMS, GUI) from the search algorithms, independent of the search technique used. Given this, and the general architecture that it implies for most scheduling applications (Figure 1), the process of arriving at a data design, and maintaining and updating that design as requirements evolve is central to the tracking of requirements into system modifications and new code. By considering the impact of a scheduling requirement on the data design (including consideration of the formal representation of the constraints needed to satisfy that requirement), one is in a position to infer the impacts of new classes of data on all of the major components of the system architecture (database, translation software, scheduler input data types).

Another observation is that the distinction between hard and soft constraints is critical. Although different classes of schedulers (i.e., using local vs. global search) locate the impact of hard and soft constraints in different portions of their algorithms, each treats these two classes of constraints distinctly. Global search algorithms tend to incorporate new hard constraints more easily, while local search algorithms tend to incorporate new soft constraints more easily. The ones that are difficult for each are difficult because they are less localized in the structure of the algorithm used for the search process. With global algorithms, for example, soft constraints may be handled either by a global utility function or (for incomplete search models) by heuristic ordering techniques (or both). With local algorithms, in contrast, hard constraints are caught in the process of formulating complete candidate schedules. The upshot is that insertion of new constraints of the 'difficult' variety require more detailed knowledge of the kind of search being carried out, and the processes necessary to set up that search, or extract valid schedules from it, in order to locate where the code must change.

A third general observation is that a general language for defining scheduler search requirements would help greatly in the automation of the generation of all classes of schedulers. Some attempts have been made at this in industry, for particular classes of scheduling domains. However, we believe that Kestrel's current approach, that of building abstract models of scheduling algorithm requirements using Specware and Planware, holds much promise for putting a great deal of this work on a much firmer

foundation. The language they are developing will make it possible for automated reasoners to relate constraint descriptions to impacts on a variety search algorithms, while easing the burden on developers to reinvent a constraint language ontology each time.

One final note: The architecture and supports for both persistent data and scheduler GUIs developed on this project were successfully applied to the CAMPS Mission Planner, also developed jointly by BBN and Kestrel. This project was responsible not only for the refinement of the architecture and support tools that made that project a success, but of sharpening the process by which changes to that system occurred.

7. Some Directions for Future Work

In closing, we mention several areas for future work on tools that would be very useful to support the translation of requirements into scheduling systems, and more generally using the general architecture of Figure 1, which includes both scheduling and simulation systems, among others.

One direction is in the area of tools to support semantic data design, and the relationships between data design requirements and the various representations of data that are found in complex, heterogeneous systems more and more often today. The systems that we have developed all were based on maintaining multiple representations of the same kinds of data, due in part to the need to give users appropriate visualization tools for the data and in part to the need to make data persistent – by using relational database representational mechanisms. Thus, even in a single system designed from scratch, we are faced with multiple data representations and the proliferation of changes that must be accomplished when the scheduling model's requirements change.

A tool that can help users capture and represent requirements semi-formally will be much more effective if it also provides a way to represent semantic entities involved in those requirements, perhaps tying them into an abstract ontology (as Kestrel's Planware is beginning to do), and then helps to translate those into the various system data models that are going to be used. Planware helps automate one aspect of this, the generation of specific scheduling algorithm data types. An interface that supports more interactive design of domain semantics would be required to make this a general purpose tool. It would then, I believe, be a straightforward extension of Planware's basic approach to generate alternative representations for the same semantic elements that could be used for persistent data storage. This, it seems to me is a key near-term goal for future work.

A related direction would be to use the same high level semantic representation, in conjunction with its derived manifestations as "implementation-level" data structures for both efficient search and persistent storage, to automate the translation between those alternative representations. In this case, the approach would be to use program synthesis to automate the translation between two representations known (by their shared derivation from the same semantic models) to be semantically equivalent or nearly so. This should be a sufficiently limited instance of what is, in general, a very difficult semantic interoperability problem to be solvable using current or near term technology.

Appendix 1 – Crew Scheduler Requirements Summary

Type	Description	Approach	Other Impacts
User/System	Persistent data store	Store all data in RDBMS	Data translation betw DB, sched – changes with scheduler AND data changes.
	User can edit all data	GUI works off DB representation.	Managing inputs to scheduler that include edited schedules
	Schedule visualizations	Timeline, calendar displays	Very dependent on specific type of crew scheduling (e.g. fighters vs airlift are different)
	Observe edits during resched.	Prior schedule as input to scheduler.	Additional constraints to check in scheduling.
	Scheduler invocation /user input validation	Hand coded	Changes with scheduler and data changes.
	Data translation betw DB, sched	Hand coded	Changes with scheduler and data changes.
Data Design	Must cover all inputs, outputs, reporting reqs.	See Figure 2. and discussion.	Changes to data design can come from scheduler design and from user reqs. Changes can impact all aspects of developed system.
Hard Constraints	Crew Member Availability	Add table to describe available times, add constraint to scheduler.	Additional editor for available times.
	Crew Member Conflicts (crew-flight-separation)	Additional scheduler constraint/filter on candidates	
	Instructors fly w. trainees	Manually scheduled	
	Crew Qualification constraints – Necessary requirements, exclusion requirements	Adds qualifications, position requirements, matching constraints	Additional data entry screen, input translation code
	Fill all open positions	A search termination constraint	
Soft Constraints	Periodic training requirements (n times each quarter/half or full yr)	Part of requirement definition table, search utility fn.	Requirements editor, sortie event editor.
	Currency training requirements (once every k days)	Part of requirement definition table, search utility fn.	Requirements editor, sortie event editor.
	Equitable combination of utilities	See discussion of Soft Constraints.	Not user tunable – no GUI impacts.
Open Issues	Scheduling Crews with overdue requirements	Manually scheduled due to need for accompanying Instructor	
	Prefer least available crew members	Manually scheduled	
	Redesign flight schedule to accomplish crew requirements	Manually revised flight schedule	

Appendix 2 – KIDS Theory of Crew Scheduler

```
%%% -*- Mode: RE; Package: RE; Base: 10; Syntax: Refine  -*-

%%% File: crew-sched.re

#|| Theory for optimizing version of CREW-SCHEDULING

||#

!! in-package("RE")
!! in-grammar('THEORY-GRAMMAR, 'REGROUP)

THEORY CREW-SCHED-OPT

%-----------------------------------------------------------------------
THEORY-IMPORTS {}

%-----------------------------------------------------------------------
THEORY-TYPE-PARAMETERS {}

%-----------------------------------------------------------------------
THEORY-TYPES

% time represented in minutes  - no need to get crazy.
type time = fixnum  % used to import these from SIMPLE-TIME theory
type duration = fixnum
type day = fixnum  % ceiling(time / *minutes-per-day*  )

constant *max-time* : time = 2000000000
constant *min-time* : time = 0
constant *minutes-per-day* : time = 1440

% constant *secs-in-day* : duration = 86400           % 24hr

constant *currency-utility-factor* : real = 1.0
constant *period-utility-factor* : real = 1.0

% needed?
var *current-date* : time = 0

type QUANTITY = fixnum
type NM = fixnum %% Nautical miles

% see below for definitions
var *requirements-database* : requirements-map = {| |}
% var *crew-requirement-status*

var *requirements-info-database* : crew-req-info-map = {| |}

% minimum duration from end of one flight to begining of next by a crew member.
var *crew-separation-time* : time = 120 % minutes

%------------------------------ Crew Members ----------------------

% Personnel QUALIFICATIONS are actually hierarchically organized,
% but we flatten and assume  mutliple qualifications (pilot, AC commander..)
% for each individual.
type qualification = symbol

% REQUIREMENT-IDs  are codes for  kinds of mission events,
% like take-off, land, night landing, bombing, firing guns, ...
type requirement-id = symbol

% A requirement that an individual must accomplish with some regularity.
% Unfortunately, the data here are too generic. The frequency with which
% a crew member must meet a requirement varies by experience level, etc.
%  There are two kinds of constraint on requirements:
%  1. Currency Reqt: has a maximum separation
```

```
%  2. Frequency Reqt: must have accomplished n in some calendar interval.
% A currency requirement max-separation of 0 means NO CURRENCY REQUIREMENT.
% A frequency requirement number-per-period of 0 means NO FREQUENCY REQUIREMENT.

type requirements =
  map(requirement-id,
      tuple(requirement-name : string,
% {'annual, 'biannual, 'quarterly, 'monthly}
            period : integer,      % 91, 182 , 365 days
            default-number-per-period : integer,
            default-max-separation : duration)) % typically 30, 60, 90 days.

type requirements-map =  map(requirement-id, requirements)

% this stuff shouldn't change -
% EXCEPT I don't know what to do with end-of-current-period - probably goes above.
% to simplify things a bit, we might put the general requirement info in here too.
type crew-requirement-info =
  tuple(
        currency-duration : duration,    % unfortunately, this can vary by individual
        number-to-accomplish : integer, % unfortunately, this can vary by individual
        period-length : integer          %
        )

type CREW-ID = symbol % should be a string representing the SSNO, but this is better.


type crew-req-info-map =
  map(crew-id, map (requirement-id, crew-requirement-info))



% These are the guys to be assigned to crew positions on sorties.
type CREW-MEMBER =
  tuple(id : crew-id,
        last-name : string,
        first-name : string,
        qualifications : set(qualification),
        requirements : set(requirement-id),
        initial-status : map(requirement-id, requirement-status)
        )

type CREW-MAP =  map(crew-id, crew-member)

% This is their requirement status (for time T0)
type requirement-status =
  tuple(date-last-accomplished : day,
        number-accomplished : integer,
        end-of-current-period : day
%       urgency : fixnum % 1 to 10 - this is really just caching
        )

type crew-req-map = map(requirement-id, requirement-status)


% status of CREW-MEMBER -> REQUIREMENTS
type crew-requirement-status =
  map(crew-id, crew-req-map)

% Sortie crew positions can be filled if the crew member has the
% NECESSARY QUALIFICATIONS, but not any EXCLUDED ones.
% The position must be filled if it is REQUIRED-P.


% e.g. pilot, copilot, navigator, load master, ...
type crew-position-type = symbol

% there may be some number of crew members filling a type of role.
type crew-position = tuple(position-type : crew-position-type,
                           position-index : integer)
```

```
type position-description =
  tuple(id : crew-position,
        required-p : boolean,
        necessary-qualifications : set(qualification), % typically only one
        excluded-qualifications : set(qualification)
        )

% there is one of these for each sortie.
% type crew-position-map =
%   map(crew-position, position-description)


% A sortie is a flight. It is the thing the CREW-MEMBERS are assigned to (in specific
positions).
% It includes a number of sub-events for which crew members need to maintain experience
% levels.
type sortie-id = symbol % by fiat
type sortie-index = fixnum
type position-index = fixnum % in sequence of position-descriptions of sortie

type SORTIE =
  tuple(id : sortie-id,
        mssn-type : symbol, % provides only default information
        start-time : time,
        end-time : time,
        sub-events : set(requirement-id),
        positions : seq(position-description) %crew-position-map
        )

type position-reference = tuple(sidx : sortie-index, pidx : position-index)


% not used - replaced by seq
% type sortie-map =
%   map(sortie-id, sortie)


% this is the basic reservation triple,
type reservation =
  tuple(sortie : sortie-index, position : position-index, crew: crew-id)

type crew-assignments =
  map(crew-id, seq(reservation))

% there should be one set of reservations for each sortie.
% we may want to make the set of reservations into a map by position?
type sortie-assignments =  seq(set(reservation))



%-----------------------------------------------------------------------------
THEORY-OPERATIONS

function day-of(tm : time) : day
 = lisp::ceiling(tm,*minutes-per-day*)
function time-of(d : day) : time = (d - 1) * *minutes-per-day*

function apply-seq (sorties: seq(sortie), i : fixnum): sortie
  = sorties(i)

% for reference
%function map-with
%   (m:map(alpha, beta), a: alpha, b: beta): map(alpha, beta) =
%   {| x -> (if x = a then b else m(x)) | (x) x in domain(m) with a |}

function insert-crew-res (res : reservation,
                          rseq : seq(reservation),
                          sorties : seq(sortie))
  : seq(reservation)
% test the likely boundary case first
```

```
  = if sorties(res.sortie).start-time > sorties(last(rseq).sortie).start-time
  then append(rseq, res)
  else insert(rseq,
              lisp::position-if(lambda(r)
                                (sorties(res.sortie).start-time
                                 < sorties(r).start-time),
                                rseq),
              res)


% add (insert) a reservation to the crew-schedule, which is time ordered by start time.
function add-crew-reservation
  (crew-sched : crew-assignments, res : reservation, sorties : seq(sortie))
  : crew-assignments
  = {| cr-id -> (if cr-id = res.crew
                 then insert-crew-res(res,crew-sched(cr-id), sorties)
                 else crew-sched(cr-id))
       | (cr-id) cr-id in domain(crew-sched) |}


% for the global case - doesent have to be efficient! - here's a recursive version
function add-crew-reservations
  (crew-sched : crew-assignments, res-seq : seq(reservation),
   sorties : seq(sortie))    : crew-assignments
   = if empty(res-seq) then crew-sched
     else add-crew-reservations(
           add-crew-reservation(crew-sched, first(res-seq), sorties),
           rest(res-seq), sorties)


function initial-crew-req-stat (crews : crew-map) : crew-requirement-status
  = {| cr-id -> crews(cr-id).initial-status | (cr-id : crew-id)
     cr-id in domain(crews) |}



% now update an individual crew members reqs
function update-crew-req1(cr-id : crew-id,
                          reqs : set(requirement-id),
                          cr-stat : crew-requirement-status,
                          srtie : sortie) : map(requirement-id, requirement-status)

  =
  {| reqid ->
     if (reqid in srtie.sub-events)
%     req status is <date-last-accomplished,
%                    number-accomplished, end-of-current-period>
% and maybe cache              urgency
     then <day-of(srtie.end-time),
           (req-stat.number-accomplished + 1),
           req-stat.end-of-current-period> % this may change!
     else req-stat

     | (reqid : requirement-id, req-stat : requirement-status)
     reqid in reqs & req-stat = cr-stat(cr-id)(reqid) |}



function update-crew-reqs(crew-req-stats : crew-requirement-status,
                          res : reservation,
                          crews : crew-map,
                          sorties : seq(sortie)) : crew-requirement-status
  = crew-req-stats
    +* {| res.crew -> update-crew-req1(res.crew,
                                       crews(res.crew).requirements,
                                       crew-req-stats,
                                       sorties(res.sortie)) |}

%% this one works on the seq(reservations) of one crew member
function update-crew-status-1 (init-stat : crew-requirement-status,
                               res-seq : seq(reservation),
                               crews : crew-map,
```

```
                              sorties : seq(sortie)
                              ) : crew-requirement-status
  = if empty(res-seq) then init-stat
    else update-crew-status-1(update-crew-reqs(init-stat,  first(res-seq), crews,
sorties),
                              rest(res-seq), crews, sorties)



function update-crew-status (init-stat : crew-requirement-status,
                             assignments : crew-assignments,
                             cr-ids : seq(crew-id),
                             crews : crew-map,
                             sorties : seq(sortie)
                              ) : crew-requirement-status
   = if empty(cr-ids) then init-stat
     else update-crew-status(update-crew-status-1(init-stat, assignments(first(cr-ids)),
                                                  crews, sorties),
                             assignments, rest(cr-ids), crews, sorties)



% used by best-remaining-cost - lose extra arg
function update-crew-stat (init-stat : crew-requirement-status,
                             assignments : crew-assignments,
                             crews : crew-map,
                             sorties : seq(sortie)
                              ) : crew-requirement-status
  = update-crew-status(init-stat, assignments, set-to-seq(domain(assignments)), crews,
sorties)

%% defined elsewhere (in lisp) - compute the utility of a current status
function csched-utility (status : crew-requirement-status,
                         req-info-map : crew-req-info-map,
                         current-day : day) : real


% defined elsewhere
function crew-position-utility (req-stat : crew-req-map, s : sortie)


% function crew-util (p : position-reference, crew : crew-id
%                     crew-status : crew-requirement-status, sorties : seq(sortie))
%  = crew-position-utility(crew-status(crew), sorties(p.sidx))


function sched-utility(assignments : crew-assignments,
                       init-status : crew-requirement-status,
                       crews : crew-map,
                        sorties : seq(sortie),
                       current-day : integer
                         ) : real
  = csched-utility(update-crew-stat(init-status, assignments, crews, sorties),
                   *requirements-info-database*,
                   current-day)

function best-position-cost (p : position-reference,
                             c-r-status : crew-requirement-status,
                            crews : crew-map,
                             sorties : seq(sortie)
                             ) : real
  = reduce(max,
           image(lambda(cr-id)
                  crew-position-utility(c-r-status(cr-id), sorties(p.sidx)),
                 domain(crews)))

function best-remaining-cost
                       (psched : crew-assignments,
                        remaining-positions : seq(position-reference),
                       init-status : crew-requirement-status,
                       crews : crew-map,
```

```
                    sorties : seq(sortie)) : real

  = let(current-status : crew-requirement-status % THIS IS INCREMENTALLY MAINTAINED
        = update-crew-stat(init-status, psched, crews, sorties))

    reduce(max,
           image(lambda(p : position-reference)
                    best-position-cost(p, current-status, crews, sorties),
                 remaining-positions))


%---------------------- CREW-SCHED ---------------------------------------

function CREW-SCHED
  (crews : crew-map,
   init-status : crew-requirement-status,
   sorties : seq(sortie),
   open-positions : seq(position-reference),
%  prior-assignments : crew-assignments
   current-day : integer
   |  positions-refer-to-sorties(open-positions, sorties)
      & initial-status-for-crews(init-status, crews))
  returns (assigns : crew-assignments |
             consistent-crew-qualifications(crews, sorties, assigns)
             & consistent-crew-flight-separation (sorties, assigns)
             & consistent-open-positions-filled(open-positions, assigns)
%output conditions:
             & consistent-position-usage(assigns, open-positions)
             & consistent-crew-usage(crews, assigns))

%---------------------- CREW-SCHED-OPT -----------------------------------

function CREW-SCHED-OPT
  (crews : crew-map,
   init-status : crew-requirement-status,
   sorties : seq(sortie),
   open-positions : seq(position-reference),
%  prior-assignments : crew-assignments
   current-day : integer
   |  positions-refer-to-sorties(open-positions, sorties)
      & initial-status-for-crews(init-status, crews))
  returns (assignments : crew-assignments
           | extremal(assignments,
                      lambda(a1, a2)
                       (sched-utility(a1, init-status, crews, sorties, current-day)
                       <= sched-utility(a2, init-status, crews, sorties, current-day)),
                       {assigns | (assigns : crew-assignments)
                       consistent-crew-qualifications(crews, sorties, assigns)
                       & consistent-crew-flight-separation (sorties, assigns)
                       & consistent-open-positions-filled(open-positions, assigns)
%output conditions:
                       & consistent-position-usage(assigns, open-positions)
                       & consistent-crew-usage(crews, assigns)})
                       )

%---------------------- CREW-SCHED-OPT -----------------------------------
%---------------------- CREW-SCHED-OPT -----------------------------------

function make-reservation
  (sindx :sortie-index, pindx : position-index, crew : crew-id)
  : reservation
  = <sindx, pindx, crew>


% in all likelihood, this will only be used to generate the
% input variable INITIAL-CREW-ASSIGNMENTS
function empty-crew-schedule
  (crews : crew-map) : crew-assignments
  = {| cr-id -> {} | (cr-id : crew-id) cr-id in domain(crews) |}
```

```
%----------------------CONSTRAINTS-----------------------------------

% this input condition relates position-references to the sorties, positions they
reference

function POSITIONS-REFER-TO-SORTIES (positions : seq(position-reference),
                                    sorties : seq(sortie)) : boolean
 = fa(posit : position-reference)
  (posit in positions =>
   posit.sidx in domain(sorties) &
   posit.pidx in domain(sorties(posit.sidx).positions))

function INITIAL-STATUS-FOR-CREWS (init-stat : crew-requirement-status, crews : crew-map)
  : boolean
 = domain(init-stat) = domain(crews)


%---------------------------------------- CONSISTENT-CREW-QUALIFICATIONS
% A crew member assigned to a sortie position must have the qualifications to fill that
position.


function CONSISTENT-CREW-QUALIFICATIONS
  (crews : crew-map, sorties : seq(sortie),
   assignments : crew-assignments) : boolean
  = fa(res : reservation, res-seq : seq(reservation))
  (res-seq in range(assignments) & res in res-seq =>
   consistent-crew-qualifications-for-reservation(crews, sorties, res))


function CONSISTENT-CREW-QUALIFICATIONS-FOR-RESERVATION
  (crews : crew-map, sorties : seq(sortie),
   res : reservation) : boolean
   =
    sorties(res.sortie).positions(res.position).necessary-qualifications
    subset crews(res.crew).qualifications
    &
    ({} = (sorties(res.sortie).positions(res.position).excluded-qualifications
    intersect crews(res.crew).qualifications))


function CONSISTENT-CREW-QUALIFICATIONS-FOR-NEW-RESERVATIONS
  (crews : crew-map, sorties : seq(sortie),
   res-seq : seq(reservation)) : boolean
   = fa(res : reservation)
   (res in res-seq
    => consistent-crew-qualifications-for-reservation (crews,sorties,res))


%---------------------------------------- CONSISTENT-CREW-FLIGHT-SEPARATION
% A crew member cannot be on flights that are less than *crew-separation-time* apart.
% note: crew-assignments is a map of crew-id to a seq(reservations)

function CONSISTENT-CREW-FLIGHT-SEPARATION
  (sorties : seq(sortie), crew-scheds : crew-assignments ) : boolean
  = fa(res-seq : seq(reservation))
       ( res-seq in range(crew-scheds)
       => crew-flights-separated-p(sorties,res-seq))

function CREW-FLIGHTS-SEPARATED-P
  (sorties : seq(sortie), reservations : seq(reservation)) : boolean
   = fa(i : fixnum)
  ( i in [1 .. (size(reservations) - 1)]
   => sorties(reservations(i).sortie).start-time
     <=
     sorties(reservations(i + 1).sortie).start-time
     - *crew-separation-time*)


function CONSISTENT-CREW-FLIGHT-SEPARATION-FOR-ADD
   (sorties : seq(sortie),
```

```
   crew-scheds : crew-assignments,
   res : reservation) : boolean
   = crew-flights-separated-p(sorties,
                              insert-crew-res(res,
                                              crew-scheds(res.crew),
                                              sorties))


function CONSISTENT-CREW-FLIGHT-SEPARATION-for-new-reservations
   (sorties : seq(sortie),
   crew-scheds : crew-assignments,
   res-seq : seq(reservation)) : boolean
   = empty(res-seq) or
     (consistent-crew-flight-separation-for-add
        (sorties, crew-scheds, first(res-seq))
      &
      consistent-crew-flight-separation-for-new-reservations
               (sorties,
               add-crew-reservation(crew-scheds, first(res-seq), sorties),
               rest(res-seq)))



%------------------------------------------ CONSISTENT-OPEN-POSITIONS-FILLED

% All required positions on all sorties must be filled.

function CONSISTENT-OPEN-POSITIONS-FILLED
  (open-posits : seq(position-reference),
   assignments : crew-assignments) : boolean
  = fa( posit : position-reference)
  (posit in open-posits => position-filled (posit, assignments))


function reservation-for-position (res : reservation, posit : position-reference) :
boolean
 = (posit.sidx = res.sortie
    & posit.pidx = res.position)


function position-filled (posit : position-reference,
                          assignments : crew-assignments) : boolean
 = ex(res-seq : seq(reservation), res : reservation)
   (res-seq in range(assignments) & res in res-seq
    & reservation-for-position(res, posit))


%%%% INCREMENTAL VERSION OF THIS


function OPEN-POSITIONS-FILLED-UPTO
  (sched-posits :  seq(position-reference),
   assignments : crew-assignments) : boolean
 = if empty(sched-posits) then true
   else position-filled(first(sched-posits),assignments)
        & open-positions-filled-upto(rest(sched-posits), assignments)

%------------------------------------------ CONSISTENT-POSITION-USAGE


% all assignments come from the given sorties - this should be a noop - always true.

%given the use of the top level arg open-positions, change this to
% say all reservations are for positions from the open positions list..


function CONSISTENT-POSITION-USAGE
  (assignments : crew-assignments, open-posits : seq(position-reference))
  : boolean
  = fa (res-seq : seq(reservation) , res : reservation)
```

```
  (res-seq in range(assignments) & res in res-seq    % res.sortie in domain(sorties))
   =>  ex(posit : position-reference)
         (posit in open-posits & reservation-for-position(res, posit)))


%%%% NEED INCREMENTAL VERSION OF THIS?


%----------------------------------------- CONSISTENT-CREW-USAGE

% remember: type crew-assignments = map(crew-id, seq(reservation))
% all crew assignments are made using the given crew members

function CONSISTENT-CREW-USAGE
  (crews : crew-map, assignments : crew-assignments) : boolean
  = fa(cm-res : seq(reservation) , res : reservation)
  (cm-res in range(assignments) & res in cm-res => res.crew in domain(crews))

%-----------------------------------------------------------------
%  INCREMENTAL FORMS FOR UPDATE SCHEDULE UTILITY

function UPD-SCHED-UTIL-FOR-NEW-RESERVATION
    (cur-util : real,
     res : reservation,
     cur-status : crew-requirement-status,
     sorties : seq(sortie))
   = cur-util + crew-position-utility(cur-status(res.crew), sorties(res.sortie))

function UPD-SCHED-UTIL-FOR-NEW-RESERVATIONS
    (cur-util : real,
     reservations : seq(reservation),
     cur-status : crew-requirement-status,
     crews: crew-map,
     sorties : seq(sortie))
% does cur-util want to be left as csched-utility(cur-status,*requirements-info-
database*,current-day) ?
   = if empty(reservations) then cur-util
     else upd-sched-util-for-new-reservations
           (upd-sched-util-for-new-reservation(cur-util,first(reservations),cur-
status,sorties),
            rest(reservations),
            update-crew-reqs(cur-status,first(reservations),crews,sorties),
            crews,
            sorties)


%--------------------- ORDER-FNS ----------------
% Fold ALLOWED-CREWS into the ordering fn.
% function ALLOWED-CREWS (crews : crew-map, pos : position-index, srt : sortie) :
seq(crew-id)

function ORDER-CREWS-FOR-POSITION (crews : crew-map,
                                   srt : sortie,
                                   pos-idx : position-index,
                                   req-stat : crew-requirement-status)
                           : seq(crew-id)


%--------------------- SCHED-EXTENDS ----------------


function SCHED-EXTEND (sched : crew-assignments, psched : crew-assignments,
                       posits : seq(position-reference))



%------------------------------------------------------------------------ THEORY-LAWS
THEORY-LAWS

%-------------- Base Case laws --------------------
assert BASE-CASE-CONSISTENT-CREW-QUALIFICATIONS
```

```
  fa (crews : crew-map, sorties : seq(sortie))
  (consistent-crew-qualifications(crews, sorties, empty-crew-schedule(crews))
   = true)

% how do we tell the system that INITIAL-CREW-ASSIGNMENTS = emty-crew-schedule?
% to generalize this, we must be able to mandate that this base case applies to ANY
% given INITIAL-CREW-ASSIGNMENTS

assert BASE-CASE-CONSISTENT-CREW-FLIGHT-SEPARATION
  fa (sorties : seq(sortie), crews : crew-map)
  (consistent-crew-flight-separation (sorties, empty-crew-schedule(crews))
   = true)

#||
assert MONOTONE-OPEN-POSITIONS-FILLED-UPTO
  fa (i : sortie-index , j : sortie-index, posits : seq(position-reference),
      sched : crew-assignments)
  ( (i <= j) =>
   (open-positions-filled-upto(posits, sched, j)
    =>
    open-positions-filled-upto(posits, sched, i)))
||#

% this should be derivable from the fn def? (tests for empty sched positions)
assert BASE-CASE-OPEN-POSITIONS-FILLED-UPTO
  fa(sched : crew-assignments)
  (open-positions-filled-upto(sched, []) = true)

% maybe later we'll do duty day
% assert BASE-CASE-CONSISTENT-CREW-DUTY-DAY-SEPARATION

%-------------- Distributive laws --------------------


% ---------  CREW QUALIFICATIONS

assert DISTRIB-CONSISTENT-CREW-QUALIFICATIONS-OVER-ADD
  fa (crews : crew-map, sorties : seq(sortie), psched : crew-assignments,
      res : reservation)
  consistent-crew-qualifications(crews, sorties, add-crew-reservation(psched, res,
sorties))
   =
   (consistent-crew-qualifications(crews, sorties, psched)
   & consistent-crew-qualifications-for-reservation(crews,sorties,res))


assert DISTRIB-CONSISTENT-CREW-QUALIFICATIONS-OVER-CONCAT
   fa(crews : crew-map, sorties : seq(sortie),
       psched : crew-assignments, res-seq : seq(reservation))
  consistent-crew-qualifications(crews, sorties, add-crew-reservations(psched, res-seq))
   =
   (consistent-crew-qualifications(crews, sorties, psched)
    &  consistent-crew-qualifications-for-new-reservations (crews,sorties,res-seq))


% ---------  FLIGHT SEPARATION


assert DISTRIB-CONSISTENT-CREW-FLIGHT-SEPARATION-OVER-ADD
  fa (sorties : seq(sortie), cscheds : crew-assignments,
      res : reservation)
  consistent-crew-flight-separation
   (sorties, add-crew-reservation(cscheds, res, sorties))
   =
   (consistent-crew-flight-separation (sorties, cscheds)
    & consistent-crew-flight-separation-for-add (sorties, cscheds, res))


assert DISTRIB-CONSISTENT-CREW-FLIGHT-SEPARATION-OVER-CONCAT
 fa (sorties : seq(sortie), cscheds : crew-assignments,
```

```
      res-seq : reservation)
  consistent-crew-flight-separation
   (sorties, add-crew-reservations(cscheds, res-seq, sorties))
   =
  ( consistent-crew-flight-separation (sorties, cscheds)
    &
   consistent-crew-flight-separation-for-new-reservations
    (sorties, cscheds, res-seq))


#||
% i think this was to handle the relationship between crew-assignments and sortie-
assignments
% in the original theory.

% this might also  be written as  an implication like
% csched = c-s-m(c,s,sched) => c-s-m(c,s,add-res(sched,r)) = add-crew-res(csched,r)
assert HOMOMORPHIC-CREW-SCHED-MAP-ADD
  fa (sorties : seq(sortie), sched : sortie-assignments ,
      crews : crew-map, res : seq(reservation))
  (crew-sched-map(crews, sorties, add-reservations(sched, res))
   =
   add-crew-reservations(crew-sched-map(crews, sorties, sched),
                         res,
                         sorties))
||#
% ------------------------------------  COST RULES


assert CREW-SCHED-EXTENDS-UTIL-LE
 fa (sched : crew-assignments, psched : crew-assignments,
     remaining-positions: seq(position-reference),
     sorties : seq(sortie),
     crews : crew-map,
     init-status : crew-requirement-status, day : integer)
 (sched-extend(sched, psched, remaining-positions)
%  & things needed to unify w. remaining-positions, crews, sorties, init-status??
   => sched-utility(sched, init-status, crews, sorties, day)
        <= sched-utility(psched, init-status, crews, sorties, day)
           + best-remaining-cost(psched, remaining-positions, init-status, crews,
sorties, day))



assert CREW-SCHED-UTIL-FOR-NEW-RESERVATION
 fa (psched : crew-assignments, new-psched : crew-assignments,
     res : reservation,
     init-status : crew-requirement-status,
     crews : crew-map,
     sorties : seq(sortie),
     day : integer)

  sched-utility(add-crew-reservation(psched, res, sorties), init-status, crews, sorties,
day)
   = upd-sched-util-for-new-reservation(sched-utility(psched, init-status, crews,
sorties, day),
                                        res,
                                        update-crew-stat(init-status, psched, sorties,
crews),
                                        crews, sorties)

%sched-utility(psched,  init-status, crews, sorties, day)
%     + crew-position-utility(c-r-status(cr-id), sorties(p.sidx))

% which can also be written in terms of csched-utility, used to define sched-utility
% csched-utility(update-crew-stat(init-status,
%                                 add-crew-reservation(psched, res, sorties),
%                                 sorties, crews),
%             *requirements-info-database*, day)
%  = let (c-r-stat : crew-requirement-status
```

```
%            = update-crew-stat(init-status, psched, sorties, crews))
%     csched-utility(c-r-stat, psched, *requirements-info-database*, day)
%    + crew-position-utility(c-r-stat(res.crew), sorties(res.sortie))


%-----------------------------------------------------------------------
THEORY-RULES


function CREW-SCHED-BASE-CASE-CONSISTENT-CREW-QUALIFICATIONS ()
  rb-compile-simplification-equality
  BASE-CASE-CONSISTENT-CREW-QUALIFICATIONS

function CREW-SCHED-BASE-CASE-CONSISTENT-CREW-FLIGHT-SEPARATION ()
  rb-compile-simplification-equality
  BASE-CASE-CONSISTENT-CREW-FLIGHT-SEPARATION

%function MONOTONICITY-OF-OPEN-POSITIONS-FILLED-UPTO ()
%  rb-compile-implication-backward
%  MONOTONE-OPEN-POSITIONS-FILLED-UPTO

function CREW-SCHED-DISTRIB-CONSISTENT-CREW-QUALIFICATIONS ()
  rb-compile-simplification-equality
  DISTRIB-CONSISTENT-CREW-QUALIFICATIONS-OVER-ADD

function CREW-SCHED-DISTRIB-CONSISTENT-CREW-QUALIFICATIONS-CONCAT ()
  rb-compile-simplification-equality
  DISTRIB-CONSISTENT-CREW-QUALIFICATIONS-OVER-CONCAT

function CREW-SCHED-DISTRIB-CONSISTENT-CREW-FLIGHT-SEPARATION ()
  rb-compile-simplification-equality
  DISTRIB-CONSISTENT-CREW-FLIGHT-SEPARATION-OVER-ADD

function CREW-SCHED-DISTRIB-CONSISTENT-CREW-FLIGHT-SEPARATION-CONCAT ()
  rb-compile-simplification-equality
  DISTRIB-CONSISTENT-CREW-FLIGHT-SEPARATION-OVER-CONCAT

%% THIS IS USED TO BOUND THE COST OF THE SCHEDULE.
function RULE-CREW-SCHED-EXTENDS-UTIL-LE ()
  rb-compile-le
  CREW-SCHED-EXTENDS-UTIL-LE

%function CREW-SCHED-HOMOMORHIC-CREW-SCHED-MAP-ADD ()
%  rb-compile-simplification-equality
%  HOMOMORPHIC-CREW-SCHED-MAP-ADD

%-----------------------------------------------------------------------
THEORY-MISC-LAWS

#||
% in basic-boolean this is a rewrite; here it is a revocable equality
function KTS-RULE-DISTRIBUTE-VAR-DEFINITIONS-IN-SOME-OP (a)
   computed-using
   a = 'some(local-b)( $Q & @b-r = @b-r-term)'
   & binding-ref(b-r)
   & ref-to(b-r) = local-b
   & ref-to(b-r) ~in free-vars(b-r-term)
   & substitution = form-subst-by-zip([ref-to(b-r)], [b-r-term])
   & new-a = make-structure(
          '##r RB-GRAMMAR
           (rule-instance-make UNDEFINED,
             if @(conjunctify(
                   {subst-simultaneous(Q-conjunct,substitution)
                  | (Q-conjunct) Q-conjunct in Q}))
             then @(c-t(b-r-term))
             else undefined,
             ${}, ${},
             0, KTS-RULE-DISTRIBUTE-VAR-DEFINITIONS-IN-SOME-OP)')
   => KTS-RULE-DISTRIBUTE-VAR-DEFINITIONS-IN-SOME-OP(a) = new-a
```

```
||#

%-----------------------------------------------------------------------
THEORY-MISC-RULES

#||
rule def-of-sched-extend (a)
  a = 'sched-extend(@assignments, @crew-schedule)'
  &  Y = GETNEWBINDING('Y)
  --> a = 'SEQEQUAL(@assignments, add-crew-reservations(@crew-schedule, Y))'
%%% %THIS NEEDS TO BE MAPEQUAL IF USED.



%rule SEQ-THEORY-RULE-DEF-OF-EXTENDS (var EXPR-65)
%  EXPR-65 = 'EXTENDS(@X, @W)'
%    & Y = GETNEWBINDING('Y)
%  --> EXPR-65 = 'SEQEQUAL(@(C-T (X)), @(C-T (W)) ++ Y)'
%        & PROGRESSION-SEQ-INDEX(Y) = 0

rule basic-boolean-theory-distribute-some-over-or-into-ex-form
  (a: operation)
   also {| index-on -> <'rb-simplification-rules, 'some> |}
   a = 'some(local-b)( or(@p, $dis))'
   & size(dis) > 0
   & test-for-defining-equation(p,local-b)
   & x = getnewbinding('x)
   & local-b1 = copy-binding(local-b)
   & local-b2 = copy-binding(local-b)
   & px = subst-vv(p, {| local-b -> x |})
   & p1 = subst-vv(p, {| local-b -> local-b1 |})
   & dis1 = subst-vv-set(dis, {| local-b -> local-b2 |})
   --> a = 'if ex(x) @px
            then some (local-b1) @p1
            else some (local-b2) or($dis1)'

rule basic-boolean-theory-distribute-some-over-or-into-let-form
  (a: operation)
   a = 'some(local-b)( or(@p, $dis))'
   & size(dis) > 0
%   & ~test-for-defining-equation(p,local-b)
   & local-b1 = copy-binding(local-b)
   & local-b2 = copy-binding(local-b)
   & p1 = subst-vv(p, {| local-b -> local-b1 |})
   & dis1 = subst-vv-set(dis, {| local-b -> local-b2 |})
   & z = getnewbinding('z)
   --> a = '(let(z = some (local-b1) @p1)
             if defined?(z) then z
             else some (local-b2) or($dis1))'
||#

%-----------------------------------------------------------------------
THEORY-MISC-FORMS

#||

form remove-SOME-OP-SIMPLIFICATION-RULES
  remove-rb-simplification-rules('some,
   {'basic-boolean-theory-rule-distribute-var-definitions-in-some-op})

form ADD-equality-rules
  ADD-RB-EQUALITIES
  ('some, {'PACAF-KTS-RULE-DISTRIBUTE-VAR-DEFINITIONS-IN-SOME-OP})

form ADD-AND-SIMPLIFICATION-RULES
  add-rb-simplification-rules('and,
                              {'basic-boolean-theory-rule-distribute-and-over-or,
                               'basic-boolean-theory-distribute-some-over-or-into-let-
form})
```

```
form add-pairwise-set-extend-rule
  add-rb-simplification-rules('pairwise-set-extend, ('def-of-pairwise-set-extend ))

||#

%-----------------------------------------------------------------------
%  GS THEORY FOR CREW SCHEDULING
%-----------------------------------------------------------------------

form INDEX-GS-theory-for-crew-scheduling
 true -->
%   index-gs-theory(
   '##r cypress-grammar
    (Global-Search-Theory GS-theory-for-CREW-SCHED-OPT1

         input-types crew-map, crew-requirement-status,
                     seq(position-reference), seq(sortie)

 % , integer
         output-types crew-assignments
         input-vars crews, init-status,  open-positions,
                    sorties1
  % , current-day
         output-vars assignments

         input-condition positions-refer-to-sorties(open-positions, sorties1)
                         & initial-status-for-crews (init-status, crews)

         output-condition
             consistent-position-usage(assignments, open-positions)
             & consistent-crew-usage(crews, assignments)

         subspace-types
                          seq(position-reference),
                          seq(position-reference),
                         crew-assignments,
                         crew-requirement-status
%                         , real
         subspace-vars
                         scheduled-positions,
                         remaining-positions,
                        crew-schedule,
                        crew-req-status
%                         , current-utility
         subspace-split-vars
                        new-scheduled-positions,
                        new-remaining-positions,
                        new-crew-schedule,
                        new-crew-req-stat
%                         , new-current-utility
% constraints on subspace vars
         subspace-vars-constraint
             consistent-position-usage(crew-schedule, scheduled-positions)
             & consistent-crew-usage(crews, crew-schedule)
             & open-positions-filled-upto(scheduled-positions, crew-schedule)

         initial-space (<[], open-positions,
                        empty-crew-schedule(crews),
                        initial-crew-req-stat(crews)>)

         satisfies
           sched-extend(assignments, crew-schedule, remaining-positions)

         split
%do i need to make POSITION a quantifier var?
           (ex(crew : crew-id, res : reservation,
               current-sortie : sortie , pos : position-reference)
            (remaining-positions ~= []
            & pos = first(remaining-positions)
            & new-remaining-positions = rest(remaining-positions)
            & new-scheduled-positions = prepend(scheduled-positions,pos)
```

```
%commenting this out fixes this
            & current-sortie = apply-seq(sorties1,pos.sidx)

% filter and order crews
            & crew in order-crews-for-position
                                (crews,
                                 pos.pidx,
%                               allowed-crews(crews, pos.pidx, current-sortie),
                                current-sortie,
                                crew-req-status)
            & res = make-reservation(pos.sidx, pos.pidx, crew)
            & new-crew-schedule = add-crew-reservation(crew-schedule, res, sorties1)
            & new-crew-req-stat =
                  update-crew-reqs(crew-req-status, crews, res, sorties1)
%             & new-current-utility =
%                   upd-sched-util-for-new-reservation(current-utility,res,crew-req-
status,sorties)
            ))


          extract assignments = crew-schedule
          Splittable ~ empty(remaining-positions)
          Extractable empty(remaining-positions)
          feasibility-filter true
        )' in gs-theories-prop(find-global('crew-assignments))


% also need to update crew-requirement status when a reservation is made.
% this is subject to backtrack, and relies on time-ordered  reservation making

%-------------------------------------------------------------------------

end-theory
```

MISSION OF AFRL/INFORMATION DIRECTORATE (IF)

The advancement and application of information systems science and technology for aerospace command and control and its transition to air, space, and ground systems to meet customer needs in the areas of Global Awareness, Dynamic Planning and Execution, and Global Information Exchange is the focus of this AFRL organization. The directorate's areas of investigation include a broad spectrum of information and fusion, communication, collaborative environment and modeling and simulation, defensive information warfare, and intelligent information systems technologies.