N88-17248

Trimodal Interpretation of Constraints for Planning

by
David Krieger and Richard Brown
MITRE Corporation, Bedford, MA 01730

1. ABSTRACT

Constraints are used in the CAMPS[1] knowledge-based planning system to represent those propositions that must be true for a plan to be acceptable. One mode of interpreting a constraint determines its logical value. A second mode inverts a constraint to restrict the values of some set of planning variables. CAMPS introduces a third mode: the "*make*" mode. Given an unsatisfied constraint, *make* evaluation mode suggests planning actions which, if taken, would result in a modified plan in which the constraint in question may be satisfied.

These suggested planning actions -- termed *delta-tuples* -- are the raw material of intelligent plan repair. They are used both in "debugging" an almost right plan and in replanning due to changing situations. Given a defective plan in which some set of constraints are violated, a *problem-solving strategy* selects one or more constraints as a focus of attention. These selected constraints are evaluated in the *make* mode to produce *delta-tuples*. The problem-solving strategy then reviews the *delta-tuples* according to its application and problem-specific criteria to find the most acceptable change in terms of success likelihood and plan disruption. Finally, the problem-solving strategy makes the suggested alteration to the plan and then rechecks constraints to find any unexpected consequences.

2. Background

Modern planning systems usually distinguish the "what is" knowledge that captures the salient features and constraints of a plannings application from the strategic reasoning that effectively applies such knowledge to accomplish some goal.[Davis80] [Wilensky80] [Stefik81a]. The latter, typically termed meta-rules or meta-knowledge, provides an explicit and extensible representation of the control strategies required for intelligent planning. The CAMPS Metaplanning [Brown85] component provides a mechanism for posting goals to the system and utilizing a mix of declarative *metaplans* and procedural *standard control flows* to accomplish these goals. In so doing, new subgoals are posted and new metaplans are instantiated to accomplish these subgoals. The resulting hierarchy of active problem-solving agents provides global control over local planning actions (e.g., filling in a slot or checking a constraint).

The problem-solving agents provide CAMPS with a high-level, top-down view of planning and resource allocation. However, problems with planning usually arise because some *detail* is out of place. This defect in a plan is signaled to the metaplanning component via a *constraint violation*. The third of CAMPS' three modes of constraint evaluation, **make-mode**, is intended to provide a low-level, bottom-up view of the planning problem by producing a structure called a **delta-tuple** that suggests some action the problem-solver might take to resolve the problem and eliminate the constraint violation.

Make-mode evaluation is not guaranteed or even intended to provide an exhaustive list of all possible corrective actions available to the planning system. The metaplanning component itself can use *high-level strategies for resolving multiple constraint violations in a* global manner. Rather, make-mode evaluations is an attempt to extend the role of the CAMPS primitive operators (CAMPS predicates), and through the delta-tuples, involve them in the plan repair process. This does not absolve the metaplanning component from its primary responsibilities, i.e., preventing destructive subgoal interaction/annihilation and limiting the combinatorics of the search through the space of possible plans.

The ability to respond to unforeseen conditions in the environment (replan) is a major design goal of CAMPS. In order to achieve replanning capability, problem-solving strategies and predicates must communicate in an orderly manner. Delta-tuples typically suggest ways of undoing some planning decision which, due to the limitations of CAMPS look-ahead mechanism or, more importantly, due to some unforeseeable change in the planning environment, that has made a suggested plan unacceptable.

2.1 CAMPS's Application-specific Knowledge

CAMPS organizes its *domain* knowledge around an AKO hierarchy of *plan elements.* Plan element instances represent specific tasks, resources, and other objects within a CAMPS application domain. Using the nomenclature of "frames," a plan element instance has associated with it a number of *slots*, some of which may contain instances of, or sets of instances of, other plan elements. In this way, the plan element hierarchy also represents some of the relationships that can exist between objects.

3. Defining Constraints

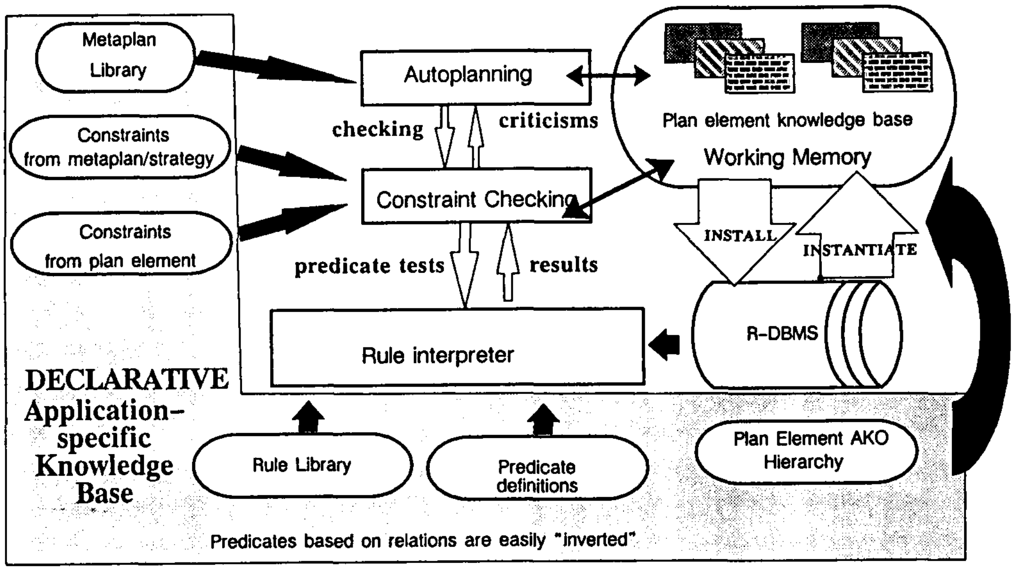
3.1 Planning as Constraint Satisfaction

CAMPS views mission planning primarily as a constraint satisfaction problem [Stefik81b] [Fox83]. Constraints describe relationships that must hold among the slots of plan elements. In its simplest form, a constraint simply says that some relationship should *always* be enforced ,i.e. there are no limiting conditions under which the constraint does not apply. This relationship is expressed as a **predicate** applied to constants and variables. Following a LISP-like syntax, a relationship may look like:

```
(PREDICATE-SYMBOL ARG1 ARG2 ...).
```

[1] This reports on work conducted by the MITRE-Bedford Artificial Intelligence Center under project 8400B, sponsored by NASA/KSC. Additional sponsorship for the implementation of the CAMPS architecture was provided by the Rome Air Development Center, COES.

Figure 1. The CAMPS architecture uses several types of declarative knowledge to support planning applications in different domains. Information about partially completed plans is kept in a relational database. When operating, information is read into working memory. Choices for filling slots in plan elements (made by the user or automaticallly in the user's behalf) are subjected to constraint checking. Finally, work is saved back into the database.


Metaplan Library
Constraints from metaplan/strategy
Constraints from plan element
Autoplanning
checking
criticisms
Constraint Checking
predicate tests
results
Rule interpreter
Plan element knowledge base
Working Memory
INSTALL
INSTANTIATE
R-DBMS
DECLARATIVE Application-specific Knowledge Base
Rule Library
Predicate definitions
Plan Element AKO Hierarchy
Predicates based on relations are easily "inverted"

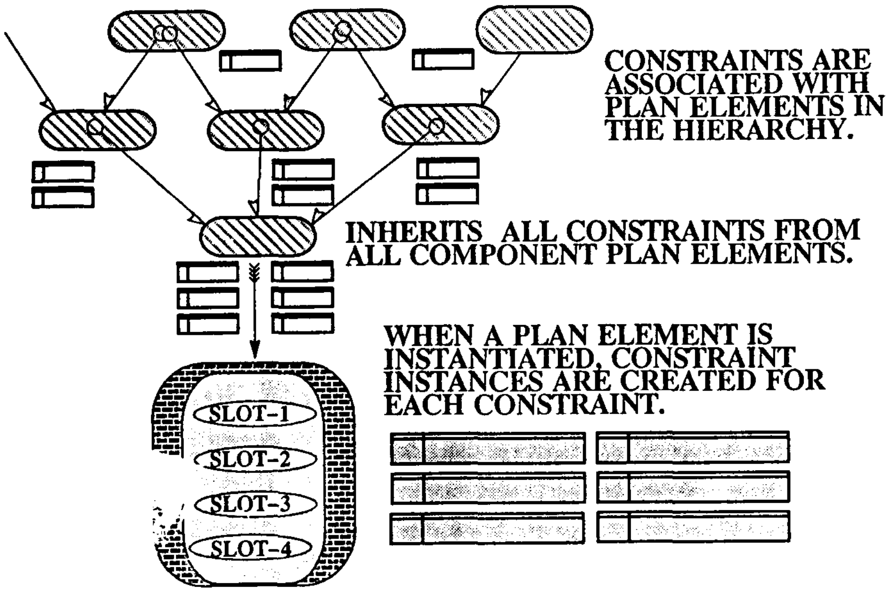

CONSTRAINTS ARE ASSOCIATED WITH PLAN ELEMENTS IN THE HIERARCHY.
INHERITS ALL CONSTRAINTS FROM ALL COMPONENT PLAN ELEMENTS.
WHEN A PLAN ELEMENT IS INSTANTIATED, CONSTRAINT INSTANCES ARE CREATED FOR EACH CONSTRAINT.
SLOT-1
SLOT-2
SLOT-3
SLOT-4

Figure 2. Planning -- solving a constraint satisfaction problem -- centers on filling slots in plan elements subject to constraints. Types ("capabilities of") plan elements are arranged into a specialization (AKO) hierarchy along which slots (but not slot values) and constraints are inherited.

ORIGINAL PAGE IS OF POOR QUALITY

For example, a constraint that enforces the condition that the start of a task must be no earlier than the task's earliest-start (as determined by some problem-solving strategy) might be of the form:

```
(*GREATER* ?START ?EARLIEST-START)
```

The variables, syntactically flagged by a leading "?", are usually associated with slots of a plan element, so that ?EARLIEST-START is a variable associated with the :EARLIEST-START slot of the plan element instance whose class includes the essential-event capability.

3.2 Constraint Declarations

Constraints are central to the topic of this paper. We will show how the various modes of constraint interpretation interact during plan construction and repair. Each constraint declaration creates a data structure with the following attributes:

Plan Element. The plan element that is the focus of the constraint.

Involved Slots The slots for which the relationship is enforced.

Conditions This is a list of predicates, some of which may be negated. The constraint is applicable only if the conditions are satisfied.

Relationship (Predicate) A constraint defines a relationship between slots of a plan element. This relationship is expressed as single, possibly negated predicate that defines the constraint.

3.3 Constraint Consequence

Ideally, a plan should not have any violated constraints. However, it is naive to believe that planning problems have solutions in which all constraints are satisfied. The CAMPS architecture uses a numeric measures of that reflects the *degree of belief* in a constraint being violated.

In addition to *degree of belief*, CAMPS also associates a **consequence category** with every constraint. The consequence category provides a *qualitative* measure of the seriousness of the constraint's violation. The consequence category helps to order the constraints for evaluation and provides a metric by which problem-solving strategies can selectively check and relax constraints. Consequence categories include *feasibility, survivability, success, efficiency* and *assumption.*

4. Variables in CAMPS

CAMPS variables occur as arguments in predicate expressions. Generally speaking, a CAMPS variable is unrelated to a LISP variable. A variable typically, but not always, corresponds to a slot in some plan element. Conversely, at an implementation level, each slot can be mapped to a CAMPS variable, which in turn holds what we heretofore have informally spoken of as "the slot's value."

The **status** of a variable reflects the nature of the its value. If the value of a variable is unknown its status is :UNATTACHED. If it is bound to a single value its status is :FIXED. If there is a problem-solving strategy that claims to be able to suggest values for the variable, the variable's status is :CANDIDATES. The strategy in this situation is termed "a generator."

Conceptually, a **generator** is a list of candidates. They are implemented as streams that return candidates from a set of possibilities. Number generators, for instance, provide a stream of all numbers between plus and minus infinity. Such a generator could receive a restriction messages that would restrict candidates to positive integers. Another :RESTRICT message sent to that generator could further restrict candidates to numbers less than seven.

CAMPS also supports generators that deal with objects other than numbers. An entire class of generators deals with plan elements. :RESTRICT messages sent to an instance of this class of generators could be based on plan element capabilities.

The primary purpose of CAMPS variables is to facilitate unification. Unification is the process whereby two patterns consisting of constants and variables are matched in such a way as to bind variables in one pattern to variables in the other. Unifying two variables not only forces them to have the same value but ensures that anything that affects one will also identically affect the other.

5. MODES of Constraint Evaluation

5.1 Normal mode evaluation

Predicates evaluated in normal mode are treated much the same as predicates in formal logic. One important distinction is that a CAMPS predicate can indicate that insufficient information is available: CAMPS predicates can return *TRUE, *FALSE (or NIL if there is not enough information to determine the predicate's logical value). Predicates in CAMPS also return two additional values: a **degree of belief** that the predicate is true and a degree of belief that the predicate is false. This last value is also known as the **degree of disbelief**. The second and third values returned support the Dempster-Schafer model of reasoning with uncertainty.

The logical value returned by the predicate is a function of the belief-disbelief values. If belief exceeds disbelief by a certain threshold, the predicate's logical value is *TRUE. If disbelief exceeds belief by that threshold, the predicate's logical value is *FALSE. If the difference between belief and disbelief does not exceed some minimum value, the predicate returns NIL meaning that the evaluation could not determine the truth or falsehood of the proposition[1].

A constraint that enforces a relationship between two slots of a plan element is either **satisfied** or **violated** depending on the logical value returned by its predicate. Formally, the predicate being *TRUE indicates that the constraint is VIOLATED, so that a slot is "acceptable" if the logical disjunction of it's constraints is *FALSE.

5.2 Bias mode evaluation

Following a general strategy of "delayed commitment" [Sacerdoti77], CAMPS has the ability to "look ahead" before fixing the value of a slot. In bias-mode, *predicates are evaluated for their side effects* on the generators associated with the CAMPS variables serving as arguments to the predicate. These side effects usually take the form of restricting an unattached variable to a set of acceptable candidates by attaching a generator to the variable, further restricting the set of a variable's candidates by sending its attached generator an appropriate :RESTRICT message, or (when we're lucky) fixing a variable's value to a single candidate.

As planning proceeds, constraints are evaluated in bias-mode, producing sets of candidates for each variable associated with a slot of a plan element. A slot will typically have several constraints associated with it, all of which are trying to be satisfied. One constraint will restrict a variable's value to a certain set of candidates; a second constraint might further restrict that set of candidates to a subset of the first and so on until a set of acceptable values is produced and/or some constraints post violations.

This approach postpones fixing the value of a slot until as much information as possible has been considered, guiding the constraint satisfaction process towards a successful conclusion without needless search through the space of (im)possible plans.

CAMPS provides two unification contexts. In bias-true, unification "attempts" to return a *TRUE result by binding the predicate variables appropriately; in bias-false, unification attempts

1 Note that this scheme can distinguish between **absence of information** and **contradictory information**. A belief/disbelief pair (0.5 0.5) indicates a lot of contradictory information, while (0.0 0.0) indicates a complete absence of information. Since we will use the same scheme to describe "confidence" in a suggested corrective action, this allows us to distinguish between a suggestion with little to recommend it (0.2 0.0), and a good suggestion with a lot of risk (0.6 0.4).

to impose restrictions on the variable that produce a *FALSE result The overall effect of bias mode evaluation is to avoid search by initially trying to restrict variables to values that have a good chance of simultaneously satisfying interacting constraints.

5.3 Make- mode evaluation

A violated constraint is often serious enough that a problem-solving strategy will choose to try to fix it. The basic idea behind behind make-mode evaluation is that the predicate associated with the violated constraint has useful information about how to fix itself. When an unsatisfied constraint is evaluated in a make-mode, a list of *delta-tuples* is returned. These delta-tuples suggest ways in which the violated constraint might be satisfied. They embody the *local* knowledge about the constraint, or more specifically, its associated predicate and conditions. It is the task of some higher level problem-solving strategy to evaluate these suggestions in terms of the overall planning goal and eventually choose one or more to execute.

6. Using Make- Mode Evaluation

The three modes of constraint evaluation -- **normal**, **bias**, and **make** are closely connected. **Normal** sees if the plan is in trouble; **bias** tries to keep the plan out of trouble; while **make**-mode suggests ways to keep the plan out of trouble it is already in.

6.1 Crucial Ideas

The key elements of CAMPS's solution to a planning problem, evidenced by having one or more constraint violations, depends on answering the following questions:

What change can be made to a plan that will fix the problem?
If the change is made, what is the belief that the problem will be corrected?
What is the expectation that the proposed change will trigger a *ripple* effect? That is, will fixing this problem in the prescribed manner introduce a disproportionate number of new and perhaps more difficult problems?
What problem-solving strategy is responsible for making the change?

Both local and global reasoning is required to address these issues during plan refinement and replanning. The metaplanning component of CAMPS provides the global perspective. Make-mode evaluation of constraints and the delta-tuples returned, provides the local perspective. In particular the **belief** that the proposed change will correct the difficulty combined with the **disbelief** derived from the *ripple likelihood* provide a reasonable selection criteria upon which a more global criteria can be based. CAMPS uses both of these perspectives in a combined top-down bottom-up approach to generate acceptable plans.

6.2 Delta Tuples

Specifically, a delta-tuple describes a possible way to modify a plan element in order to satisfy a violated constraint. As such, it must embody the essential features of some planning action such as rescheduling a task, using a different number of resources or using a different resource altogether. Implementationally, these planning actions typically distill down to changing the value of some slot of some plan element instance. Thus, **delta tuples** carry the following information:

Plan Element. The plan element to be modified by the delta-tuple.
Slots. A list of the slots expected to change if the action suggested by this delta tuple is taken. This list gives the responsible problem-solving agent a basis for selecting among alternative delta tuples. A reasonable selection criteria might try to localize the effects of carrying out a suggested action by minimizing the number of slots changed. Another might try to avoid changing slots that the user specifically set or prefers to not change.
Success Predicate. The predicate expression that will evaluate *TRUE if the suggested planning action is taken. This may or may not be the predicate of the constraint whose violation is being repaired.
Recipient. In all cases, the suggested action will take the form of a message to be sent to some message-receiving object. This recipient could be the plan element itself, the generator associated with a changing variable, or a responsible problem-solving strategy.
Message. The actual message sent to the recipient to effect the change.
Arguments. The appropriate message arguments, typically plan elements.
Confidence Pair. A Dempster-Schafer pair of confidence measures which give the *a priori* belief that the change proposed by the delta-tuple will satisfy the predicate or constraint.

6.3 Generating Delta-Tuples

The information needed to generate delta-tuples resides with the predicates. Each predicate has a set of general *operations* for changing the value of one or more of its arguments such that the predicate will be satisfied. For example, CAMPS has a predicate that enforces the relationship that ARG1 must be greater than ARG2.

```
(*GREATER* ARG1 ARG2)
```

Associated with the predicate are two make-true operations, :MAKE-GREATER and :MAKE-LESS. If a violated constraint involving this predicate is evaluated in make-true mode, CAMPS will attempt to create a delta-tuple by applying the :MAKE-GREATER operation to ARG1 and another delta-tuple by applying the :MAKE-LESS operation to ARG2.

In order for these operations to have any meaning in terms of the plan being generated, CAMPS must first trace the source of each variable serving as a predicate's argument. In the simplest case, the variable corresponds directly to a slot of a plan element. In that case, the delta-tuple would specify a *message* to be sent to a *plan element* to restrict the value of that slot variable's generator to the newly computed set of acceptable values.

In summary, make-mode evaluation is built on the following approach:

Each predicate has associated with it a set of general operations that if applied to its arguments would satisfy the predicate.
Each variable bound to a predicate argument is examined to determine if and how these operations could be properly applied to the variable.
These operations are then applied to the variable, yielding a new suggested value or range of values for the variable that promises to satisfy the predicate. It is at this point that a delta-tuple is created.

7. Example

Make-mode constraint evaluation enables CAMPS to intelligently resolve the resource conflicts that present an especially difficult problem. Most planning systems deal with many types of resources and many tasks competing for those resources. The consequence of modifying a resource allocation to satisfy one task will likely effect the viability of some other task. Our example is from the

NASA cargo loading application. Integration of experiments into racks for one mission will typically overlap with the rack-experiment deintegration of previous missions. This situation is further exacerbated by schedule changes and mission payload reassignments that force replanning and rescheduling.

CAMPS represents sets of homogeneous resources as resource-pools. In the NASA application, each rack used for holding a Space-Lab experiment is distinguishable, and is represented as a RACK unit quantity resource-pool so that racks are tracked individually (e.g., by serial number). A separate plan element including the RESOURCE-UTILIZATION capability records a single resource requirement being met from a single pool; it includes the following slots:

:CONSUMER the task requesting the resource
:RETURNING-TASK the task returning the resource.
:SUPPLIER the resource-pool supplying the resource
:BEGIN the time that the resource is first need, directed to the :START slot of the :CONSUMER.
:END the time the resource is returned to the pool, directed to the :FINISH slot of the :RETURNING-TASK.
:EARLIEST-BEGIN is the :EARLIEST-START of the :CONSUMER.
:LATEST-END is the :LATEST-FINISH of the :RETURNING-TASK.

RACKS include the RESOURCE-POOL capability, which tracks availability of the resource. Each RESOURCE-POOL includes the following information in an internal (i.e., "non slot") form:

:ALLOCATIONS A list of resource-utilizations
:AVAILABILITY A list of sublists, each of which indicates a duration and the quantity of resources available during that duration.

A constraint attached to the RESOURCE-UTILIZATION capability requires that the :SUPPLIER have sufficient quantity of the resource between :BEGIN and :END times. As planning progresses this constraint, which uses the *RESERVE* predicate in its relationship, is checked in bias-true mode. When satisfiable, a *RESERVE* predicate evaluated in bias-true mode will have several side effects:

The new RACK-UTILIZATION is pushed on to the pool's ALLOCATION list and the pool is put into the RACK-UTILIZATION's :SUPPLIER slot. Also, availability of the pool is updated to reflect the new resource utilization.

A failed reservation constraint means that the designated RACK could not supply the quantity of resources (in this case, 1) requested by the :CONSUMER because some other task(s) reserved the resource at an overlapping time interval. For example, changing a mission's launch date will usually cause once-successful rack utilizations to suddenly have violated constraints involving *RESERVE* predicates.

The user or an automated problem-solving strategy can focus on one of these constraints, and ask CAMPS to suggest corrective actions by evaluating the *RESERVE* predicate in the **make-true** mode. The resulting delta-tuples typically include:

Reschedule the task's start or end time to the closest time when the required resource becomes available for the required duration. This would amount to sending a :RESTRICT message to the generator attached to the task's START or END variable in order to restrict its value to a new acceptable range.

Change the resource requirement if the task's requirements can be met by some other resource with similar capabilities or by some other resource-pool.

Modify conflicting tasks by finding those reservations in the resource pool whose rescheduling would enable the task-at-hand's resource requirement to be met and reschedule those reservations.

Reduce the quantity of resources requested by the task to the number of resources available at the time. For resource-pools of only one resource, (as is the case with experiment racks) this is not an option and for those cases CAMPS will not generate a delta-tuple that suggests using zero of that resource.

A simple example dealing with rack resources is shown in figure-3. The RACK-DEINTEGRATION task of mission STS-1002[2] overlaps with the RACK-INTEGRATION task of STS-1003. In this particular case, the same rack SN003-s is desired by both tasks. The *rack-resource-utilization* associated with STS-1003 posts a constraint violation that reflects its failure to reserve its desired resource. Figure-6 shows the delta-tuples returned by re-evaluating that constraint in make-mode.

The belief/disbelief pairs shown embody several general considerations. Each one of these tasks has an earliest and latest start and earliest and latest finish that represent acceptable slack in the schedule. For the delta-tuples dealing with changing start and end times, the belief decreases and the disbelief increases as the difference between the original and suggested time increases.(figure 5) For the delta-tuples that suggest using another resource, the belief is a function of the number of other resources in that pool and/or the number of pools with the same rack capability[3].

Only a glance at figure-6 shows that one delta-tuple seems the most likely to succeed. The final step in the planning process is to invoke the chosen delta-tuple by sending the message and arguments off to the specified recipient.

ORIGINAL PAGE IS OF POOR QUALITY

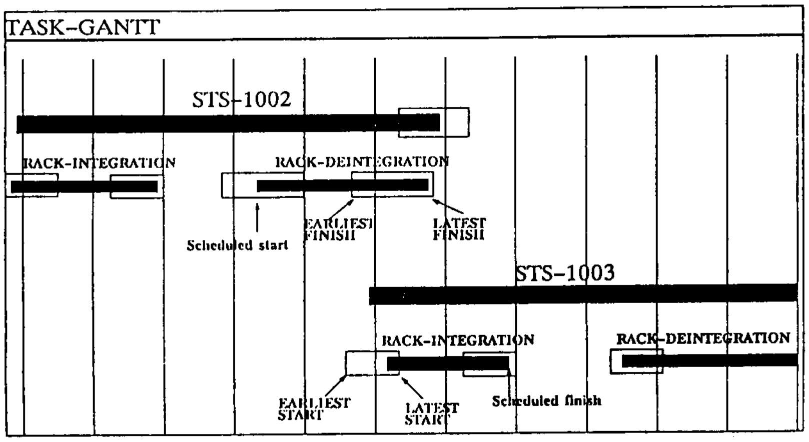


Figure 3. A Gantt chart showing a very small portion of a payload preparation for two hypothetical space lab missions. Rack integration -- putting experiments into racks and racks into the module -- is the consumer task of racks, while rack deintegration readies the rack for reuse.

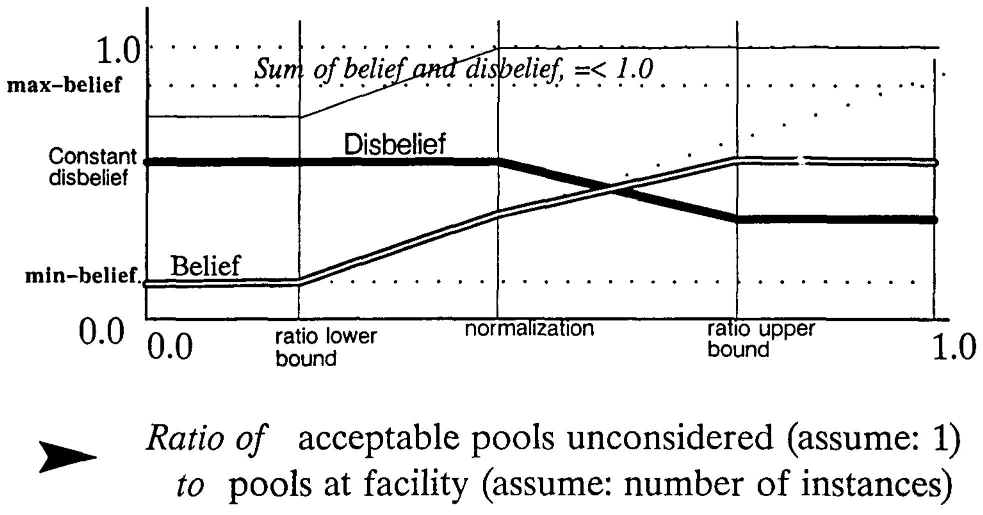


Figure 4. Calculating a belief and disbelief for a suggestion. Given the number of pools believed acceptable but not yet considered (e.g., the number of remaining candidates) and the number of pools, calculate the ratio. A low ratio indicates that most potential candidates have been rejected. Disbelief is a constant, perhaps derived from the type of the resource. Normalization forces the belief and disbelief to total less than 1 by multiplying each by 1/(belief+disbelief) when their sum exceeds 1.0.

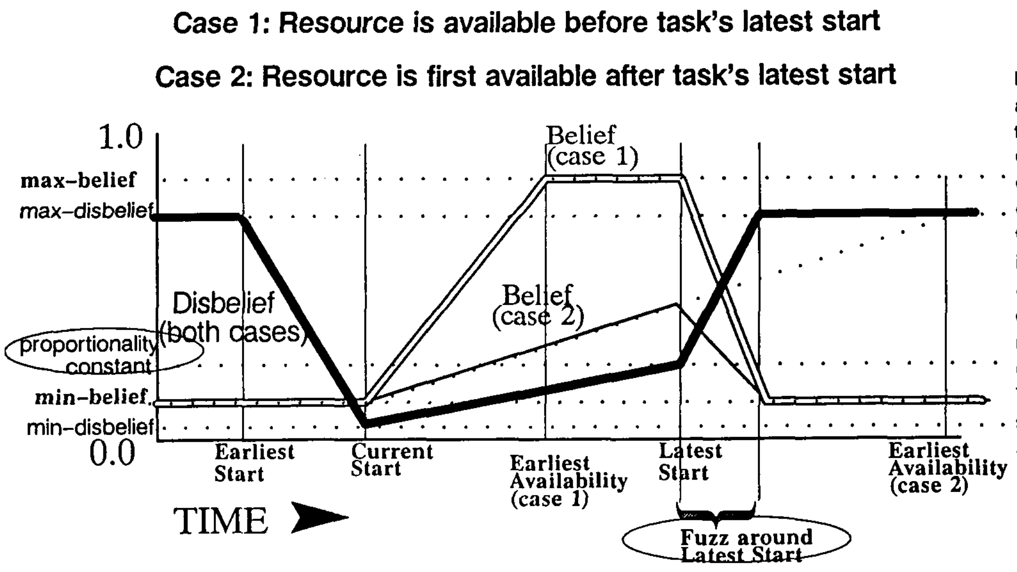


Figure 5. Unnormalized belief and disbelief as a function of the delayed start. Two arbitrary constants are used: a constant describing how much more disbelief increases as a function of time after the start is delayed beyond the currently assumed "latest start;" and a constant which, when multiplied by the task's current duration, gives a tolerance. Two cases are shown: (1) shows the resource becoming available before the task's latest start; (2) shows the resource becoming available after the task's latest start.

Plan Element	Slots	Message	Recipient	Confidence Pair
Overall-Rack-Deintgration-1002	(:END) *Suggests to finish deintegration earlier*	:DELTA-RESTRICT	Overall-Rack-Deintgration-1002	(0.65 0.35)
Overall-Rack-Integration-1003	(:START) *Suggests to start integration later*	:DELTA-RESTRICT	Overall-Rack-Integration-1003	(0.35 0.45)
STS-1002	(:START) *Suggests to switch in mission sequence*	:DELTA-RESTRICT	STS-1002	(0.1 0.8)
Rack-Resource-Utilization-1002	(:SUPPLIER) *Suggests using a different rack*	:DELTA-RESTRICT	Rack-Resource-Utilization-1002	(0.1 0.8)

Figure 6. Given the constraint violation of figure 3, EMPRESS-II returned four delta-tuples as shown by this table. While we argue that selecting the "best" delta-tuple should be mediated by higher-level problem-solving strategies, we can see that simply picking the "most believed" suggestion is a good general heuristic.

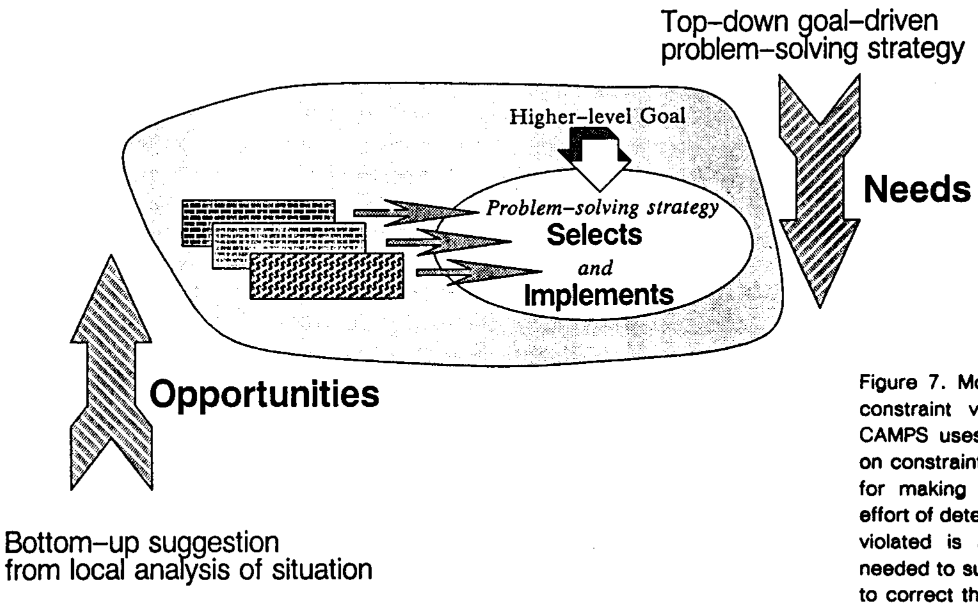

Top–down goal–driven problem–solving strategy
Higher–level Goal
Problem–solving strategy
Selects
and
Implements
Needs
Opportunities
Bottom–up suggestion from local analysis of situation

Figure 7. Most planning systems view constraint violations as "problems." CAMPS uses MAKE- mode evaluation on constraints to suggest opportunities for making a plan better. Note: the effort of determining that a constraint is violated is about 90% of the effort needed to suggest the "obvious" ways to correct the violation.

8. Conclusion

Like many recent planning systems, CAMPS builds on a hierarchically structured metaplanning component. This component represents the planning system's "self-knowledge" and is used to control the application of primitive operators (CAMPS predicates) towards plan realization. In addition, make-mode constraint evaluation imbues the CAMPS primitive operators themselves with a degree of "self knowledge." When the planning process runs into difficulties, a dialog commences between a globally-oriented problem-solving strategy and the local constraint instances. The dialog is initiated by a problem-solving strategy that examines the violated constraints and selects one or more to evaluate in make-mode. These constraint instances dutifully return a list of suggestions embodied in the delta-tuples.

The returned delta-tuples are examined by the problem solving strategy which can choose one or more to apply or seek some alternative means of plan repair altogether. The purpose of the delta-tuples is to pose those simple alternatives that would typically cause a minimum of plan disruption and reconstruction. They embody the generally accepted heuristic of any intelligent problem solver: "Consider the easy things first."

9. Bibliography

[Brown85] Brown, Richard. Agendas: A Metaplanning Mechanism. M-series M85-26, MITRE Corporation, Burlington Road, Bedford MA 01730, July 1985.

[Brown86] Brown, Richard, "A Solution to the Mission Planning Problem" in *Proceedings of the Second Aerospace Applications of Artificial Intelligence*, Dayton, Ohio, October 13-17, 1986.

[Davis80] Davis, Randall. Meta-rules: Reasoning about Control. *Artificial Intelligence*, 15, 1980.

[Fox83] Fox, Mark S. Constraint-Directed Search: A Case Study of Job Shop Scheduling. PhD thesis, Carnegie-Mellon University, 1983.

[Sacerdoti77] Sacerdoti, E. D., A Structure for Plans and Behavior, American Elesevier, New York, 1977.

[Stefik81a] Stefik Mark. Planning and Meta-Planning. (MOLGEN: Part 2), Artificial Intelligence 16(2) 1981.

[Stefik81b] Stefik Mark. Planning with Constraints. (MOLGEN: Part 1), Artificial Intelligence 16(2) 1981.

[Wilensky80] Wilensky, Robert. Meta-Planning. First NCAI 334-336, August 1980.

2 None of these mission numbers correspond to ones actually planned. They are derived from a NASA/KSC planning exercise that determined whether there were enough racks for missions through the year 2000.

3 These numbers are based on general considerations and cannot reflect situation specific knowledge that might clearly favor a suggested course of action in spite of its low belief. However in most cases these numbers provide a good indication of the relative merits of one suggestion over another. It is typically better to shift a subtask a little than reschedule its entire parent task; and using another resource has a better chance of succeeding if many similar resources can be found.