State-Based Scheduling via Active Resource Solving

Paul Morris and John Bresina
Computational Sciences Division
NASA Ames Research Center
Moffett Field, California, U.S.A.
Paul.H.Morris@nasa.gov, John.L.Bresina@nasa.gov

Javier Barreiro*, Michael Iatauro* and Tristan Smith**
*SGT Inc., **MCTInc.
NASA Ames Research Center
Moffett Field, California, U.S.A.
Javier.Barreiro, Michael.Iatauro, Tristan.Smith@nasa.gov

Abstract—A mixed-initiative approach to activity planning for space mission operations was introduced in the Mars Exploration Rover mission, and has been extended and adapted to other missions. The approach involves a collaboration between a human planner and automated tools that reason about activities and constraints. One important class of constraints arises from state requirements and effects. The mixed-initiative framework passively detects and reports constraint violations. At the user’s request, it can also offer suggestions, obtained through automated planning techniques, for actively fixing certain violations. Due to the need for a rapid response, active solving previously used a timeline insertion strategy that limited the types of violations that could be fixed, whereas the passive checking employed an encoding of the state constraints as resource constraints that identified all the violations. In this paper, we report on an extension of the active solver to handle resource problems, allowing a unification of the passive and active strategies.

Keywords-component; mixed-initiative planning; automation for flight and ground operations;

I. INTRODUCTION

Activity planning is an important component of ground operations for actual and simulated space missions. For the Mars Exploration Rover mission, a tool called MAPGEN [3] was developed that introduced a semi-automated, mixed-initiative approach [1] for activity planning. This approach has been continued in Ensemble [2], which has been adapted for several space-related applications including Mars and Lunar missions and ISS scheduling.

Classical planning and scheduling systems have typically focused on a form of active solving where the choices are entirely under the control of an automated system. However, ground operations engineers in space-related applications have made clear their preference for an adjustable level of automation. This preference is based on a number of factors [3] that make it desirable for the human operator to actively participate in plan construction.

The mixed-initiative framework incorporates both passive and active enforcement of constraints. In passive enforcement, the system notifies the user about flaws or violations in the plan. For example, a medical constraint might require fasting for 12 hours before a glucose test; a plan that had a meal 5 hours before such a test would thus be flawed. In active enforcement, the system can suggest ways to fix a violation, for example moving the meal earlier or the test later. Active enforcement may also involve, as a user option, maintaining plan validity when responding to user modifications, for example keeping a minimum 12-hour separation when the user directly moves either activity. In our applications, the activities in a plan are regarded as being complete, so the active enforcement only suggests changing the time of activities, not adding new activities. Thus, it is restricted to a form of scheduling rather than full planning.

The Ensemble mixed-initiative framework relies on an underlying planning/scheduling constraint solver called EUROPA [4][5]. This system produces a flexible solution to an activity-planning problem. A flexible solution encapsulates in a concise data structure called a Simple Temporal Network [2] a family of plans or schedules with similar causal structure but varying start times. However, for ease of comprehension, the plan is presented to the human operator as a normal fixed-time schedule. This is chosen to be as close as is practical (considering the enforced constraints) to a reference schedule [3] that captures the intent of the user. In particular, the reference schedule reflects changes made directly by the operator as well as changes resulting from constraint enforcement. Constraints are enforced at the discretion of the operator.

EUROPA provides a library of built-in plan-oriented constraints to aid in constructing plans. In particular, it includes a timeline mechanism to help enforce mutual exclusion constraints. A timeline is a sequentially ordered set of intervals. Thus, it is possible to ensure that two activity instances do not overlap in time by specifying in the model that they have interval subgoals that go on the same timeline. Subgoals may have varying qualitative and quantitative temporal relationships to their activities [5]. These include EQUALS, where the subgoal is co-temporal with the activity, CONTAINED-BY, where the span of the subgoal covers the span of the activity, and MEETS and MET-BY where the activity and subgoal touch. We have previously used the timeline mechanism to support active enforcement. In this context, the reference schedule is used as the basis for a minimum perturbation heuristic that orders the placement choices. This
promotes stability in the plans incrementally constructed by the solver.

EUROPA also provides a mechanism to track resource usage in activity plans. The global resource usage profile is calculated from the resource transactions of individual activities that are specified in the model. The system also detects violations of overall resource limits that are specified in the model, and identifies them as flaws in the plan. Although the natively supported resources are numerical, we will see that complex state conditions can be efficiently encoded and checked as numerical resources.

Previous versions of the Ensemble solver have used the EUROPA resource mechanism for passive detection of State constraint violations. However, active fixing was based on timelines. As we will see, there are certain drawbacks associated with this. Recently, EUROPA has been extended to include efficient mechanisms for active solving of resource flaws. This provided an opportunity to unify the passive and active strategies, but there remained a need to reconcile the fixed-time reference schedule of Ensemble with the flexible-time solver mechanisms of EUROPA. In this paper we describe an approach that uses the reference schedule internally within EUROPA to guide a flexible time solving process.

A. Active State Reasoning

One of the core mechanisms in the active state-based reasoning used in Ensemble and described in [7] is conflict resolution. This involves imposing ordering constraints on events with inconsistent states so that they cannot happen at the same time. For example, a test-glucose activity could have a prior food exclusion requirement that is violated by an effect of a meal activity, necessitating an ordering of the requirement and the effect so that they cannot coincide. This basic mechanism does not distinguish between state requirements and state effects; any inconsistent pair will need an ordering and either ordering may be chosen if it is consistent with the other constraints.

Conflict resolution alone is insufficient for complete state-based scheduling. For example, consider a medication activity with a stomach-full requirement and a meal activity with a stomach-empty requirement. If the meal is constrained to follow the medication, there is no concurrent inconsistency, but those two activities alone in that order would not constitute a valid plan because the stomach-full requirement would not be fulfilled until after the medication activity has already occurred.

In traditional state-based planning and scheduling, conflict resolution is augmented by a causal link mechanism where each event that requires a particular state is linked to some event that achieves the state [9]. The link imposes a precedence constraint between the two events. It also establishes an additional state (maintenance) requirement on the interval between the events. In EUROPA, causal links can be imposed by creating a separate timeline for a set of related states, and then using contained-by subgoals on that timeline for requirements and meets subgoals for effects [4]. For example, in Remote Agent [5], a thrust activity is contained-by a pointing subgoal on an attitude timeline, and this is merged with a meets subgoal of a turning activity.

The problem with causal links is that they involve a somewhat arbitrary choice of which achiever to link to a requirement. Since the causal link is protected in the forward search, a bad choice can only be corrected by backtracking, which may be deep. Furthermore, depending on the order in which activities are added to the plan and subgoals are expanded, the correct choice may not even be available at the time the selection is made, and thus the backtracking may be futile.

Experience with Remote Agent [5] suggested that finely tuned heuristics that avoid most backtracking are needed to make the causal link method workable. Unfortunately, such heuristics are typically brittle with respect to small changes in the model. Moreover, the heuristics, which are generally tuned for the forward search, tend to become less effective after backtracking.

In other experience with causal link mechanisms during the development of MAPGEN [3], it was found that poor planning search behavior was particularly associated with situations where two activities that required the same state were linked to the same achiever. This made it impossible to insert a third activity requiring a different state between the two activities without generally extensive backtracking.

In MAPGEN, as deployed, and the Ensemble scheduler, causal link solving has been avoided due to these problems; these systems use only conflict resolution for automated violation fixing. Experience shows this approach gives good search performance even without finely tuned heuristics. The drawback is incomplete fixing, i.e., some violations may remain even though a schedule exists that could fix them. Since these systems are mixed-initiative, and flag all violations, the human operator has the opportunity (and responsibility) to fix any left by the automated process.

What is needed is a way of formulating a complete systematic search that, in effect, can correct a bad causal link "after the fact" within the forward search without the need for backtracking. It turns out a resource perspective on flaws is key to achieving this. In previous work, we have used an encoding of state constraints as resource constraints to facilitate passive detection of state violations. In this paper we introduce a systematic resource solver that can actively fix state violations using the same encoding in a way that has the effect of retroactive correction of causal links.

EUROPA provides an extensible framework; state constraints could in principle be supported by adding new modeling constructs, search mechanisms, and data structures to reason about discrete states. That new mechanism could be informed by the analogy with resource solving to better handle the causal link issue. However, given that state constraints can be efficiently encoded using numeric resources, which were already supported in EUROPA, we decided to pursue that implementation path first so we could verify the new approach, while significantly reducing the need for writing new code in the short term.
II. State Resource Encoding

In this section we describe the encoding of states as numeric resources. This is similar to the encoding used for passive resource checking in [7], except that enumerated state spaces are now also handled. First we review the encoding for Boolean states that have only true and false values. The encoding uses consumable resources, also called reservoirs, that involve distinct produce and consume transactions that increment or decrement the resource at given points in time. For example, a mission crew, where crew members can be assigned to and relieved from duties, may be regarded as such a resource.

We encode a Boolean state using a consumable resource as follows. If the state is initially true, the initial capacity of the resource is a specific large number that we designate TRUE_VALUE (currently 1000.0); otherwise the initial capacity is zero. An event that changes the state from true to false consumes TRUE_VALUE amount of the resource, whereas an event that changes the state from false to true produces a like amount. An activity that requires the true state consumes a unit amount of the resource at the beginning and returns it at the end. It is considered a violation if the available capacity of the resource drops below zero. The encoding is summarized in table I.

Note that in the true state, this numerical encoding permits up to TRUE_VALUE concurrent instances of activities that require the true state, whereas in the false state any occurrence of such an activity drops the numerical resource below 0.0, and thus produces a violation. (We assume that TRUE_VALUE is chosen to be sufficiently large to accommodate the amount of concurrency that is needed for the application.)

Enumerated state spaces may be represented by using a separate Boolean for each individual state. When a state change occurs, each state Boolean is set to false except for that of the new state, which is set to true. For example, with an enumerated state space \( \{a, b, c\} \), a state change to \( b \) would set the Boolean for \( b \) to true and those for \( a \) and \( c \) to false. Note, however, that a state change to \( b \), followed by a change to \( c \), sets \( a \) to false twice.

<table>
<thead>
<tr>
<th>Table I. Numerical Encoding of Boolean State.</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Boolean</strong></td>
</tr>
<tr>
<td>Initial true</td>
</tr>
<tr>
<td>Initial false</td>
</tr>
<tr>
<td>true → false</td>
</tr>
<tr>
<td>false → true</td>
</tr>
<tr>
<td>Start require true</td>
</tr>
<tr>
<td>End require true</td>
</tr>
<tr>
<td>Violation</td>
</tr>
</tbody>
</table>

In this situation, we would like the extra false setting to simply have no effect rather than eliciting a violation. For this reason, the resource mechanism is modified to use saturated arithmetic instead of normal arithmetic. Specifically, if a resource transaction would otherwise cause the resource value to go outside the range \([1 - TRUE_VALUE, TRUE_VALUE]\), then multiples of TRUE_VALUE are added or subtracted to bring it back within that range. Thus, the saturated resource values retain no memory of previous states, and the state transitions behave like Markov processes.

Negated and disjunctive state requirements for small enumerated sets can be effectively handled within this framework by assigning an additional resource to each negated value, i.e., introducing a second Boolean with the opposite sense. For example, with an enumerated state space \( \{a, b, c, d\} \), the disjunctive requirement \( a \text{ or } b \) is equivalent to the two requirements not_c and not_d.

For many applications, an Activity Dictionary is used to statically associate activity types with state requirements and effects. However, in some applications the users have expressed a need to configure the state impact of activity instances dynamically during planning. This is achieved by encapsulating the state requirements and effects in optional sub-activities that can be attached to the parent activities via temporal constraints. The sub-activities may be temporally offset from the parent activity, as for example the fasting requirement associated with a glucose test.

To make the example more concrete, consider a plan fragment with a one-hour meal activity \( M \) starting at noon, and a 15 minute glucose test activity \( G \) at 3:45pm. Also suppose there is an eight-hour fasting requirement preceding the end of the test. We model this using a fasting resource with TRUE_VALUE as the initial value. The fasting requirement for \( G \) results in a consumer transaction at 8am and a producer transaction at 4pm, both with 1.0 as the amount. We also have a consumer transaction at noon, and a producer transaction at 1pm, with TRUE_VALUE as the amount in both cases. This plan fragment is flawed because the value of the fasting resource drops to -1.0 at noon, which is a violation. Table II summarizes the situation using negated amounts to indicate the consumer transactions. We will use this later as a running example to illustrate how resource solving operations can fix state violations.

<table>
<thead>
<tr>
<th>Table II. Meal/glucose test interacts with fasting resource.</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Time</strong></td>
</tr>
<tr>
<td>Initial</td>
</tr>
<tr>
<td>08:00</td>
</tr>
<tr>
<td>12:00</td>
</tr>
<tr>
<td>13:00</td>
</tr>
<tr>
<td>16:00</td>
</tr>
</tbody>
</table>
III. Active Resource Solving

EUROPA provides several mechanisms for solving planning and scheduling problems. There is a basic flaw-based Solver cycle that

1) detects flaws in the current plan,
2) selects a flaw for resolution,
3) chooses a way of resolving the flaw, and
4) backtracks if the flaw cannot be fixed.

In the case of resource scheduling problems, there are built-in methods for each of these steps. For step 1, a resource profile is constructed for the current plan that shows where the resource limits are exceeded. The selection in step 2 is done with user-specified heuristics. Step 3 installs an ordering constraint between two resource transactions with the potential to resolve the flaw. Step 4 undoes previous steps to return to a point where an alternative choice is available. For the search to be complete and non-redundant, it suffices that the choices in step 3 be mutually exclusive and exhaustive.

There are hooks to easily customize the various Solver steps, and some variations on the methods are also provided. In particular, there is a special grounded version of step 1. As mentioned earlier, EUROPA normally produces flexible solutions to activity planning problems. It does this by analyzing the constraints so that it can detect flaws that it knows must occur in some schedule that is consistent with the current set of temporal constraints, using the maximum flow approach of [8]. It then adds new constraints to exclude those flaws. The final product is a “safe” set of constraints that guarantees that every schedule satisfying them has no flaws.

The grounded resource flaw detection method departs from this in that it only looks for flaws in an evolving grounded schedule, i.e., one in which every activity has a definite time. This avoids the high overhead associated with the maximum flow method. As it adds constraints, it updates the grounded schedule to one that satisfies the new constraints. The final product is a grounded schedule that has no flaws, and an accompanying set of constraints. However, in this case the set of constraints does not necessarily constitute a safe flexible solution; there may still be flaws in some of the other schedules that satisfy those constraints. In our applications this is not an important issue because the user generally only needs (and sees) the grounded solution.

In principle, any grounded schedule that satisfies the constraints can be used for grounded solving. For example the earliest-time schedule, where every activity starts at the earliest time allowed by the constraints, is an obvious possibility. However, in our applications plan stability is important. This requires that as constraints change, the grounded schedule should change as little as practical in order to satisfy the new constraints. Our approach uses reference times [1] as the grounded schedule. This requires that the reference times be updated as constraints are added to remove the flaws. In [1], a solution grounding algorithm is provided that does the updating. However, active solving of resources requires an update after each solver step. Consequently, we have implemented a new more efficient grounding algorithm, which will be discussed in a forthcoming paper.

It should be noted that flexible solutions do have advantages over single solutions. In particular, the flexibility may be useful in dealing with temporal uncertainty during execution [8]. In our case, the above methods can be regarded as producing a flexible temporal solution by ignoring the reference times and just keeping the precedence constraints introduced to eliminate flaws. This flexible solution is not necessarily safe because it may have other grounded solutions besides the reference schedule and those might not satisfy the resource constraints. However, it may be safe, if the grounded schedules have exposed the possible flaws. Safety can be verified by a single maximum flow calculation using the methods of [8].

If not safe, the above procedure can be followed by the EUROPA flexible solving procedure to eliminate the remaining flaws. However, an efficient alternative is to augment the constraints with additional producer/consumer and consumer/producer precedences extracted from the grounded solution. When the precedences are complete in this way, every solution satisfying the temporal constraints is flawless if any one is. This augmenting approach is unlikely in general to provide as much flexibility as the flexible solving method, although for many applications this approach offers an appropriate trade-off between solution speed and flexibility.

IV. Search Behavior

We now consider how the resource encoding affects search and backtracking behavior. A resource flaw occurs when the value of the resource falls outside preset upper and lower limits. We will confine the discussion to the lower limit; the upper limit is similar or can be avoided by using complementary resources. The state-based encoding uses only the lower limit.

Given a fixed schedule, lower limit flaws can only occur at points where there are consumer transactions that decrement the resource below the limit. Consumers that contribute to the resource shortfall at a flaw point are called culprits; it will be useful to refer to those at the flaw itself as prime culprits and the others as secondary culprits. Producers that positively contribute to the resource value at the flaw point, and so partially mitigate the shortfall are helpers. Other producers that are not currently helpers, but might be if moved, are called saviors. Clearly, culprits and helpers precede the flaw and saviors strictly follow the flaw.

Fig. 1 illustrates this for the glucose testing example. The -1000.0 consumer at noon (start of meal), where the flaw occurs, is a prime culprit, while the -1.0 consumer at 8am (start of fast) is a secondary culprit.
The +1000.0 initialization is a helper, and the +1000.0 producer at 1pm (end of meal) and +1.0 producer at 4pm (end of fast) are saviors.

Intuitively, the flaw shortfall can only be reduced by moving a savior to before the flaw or moving a culprit to after the flaw. However, the latter move would merely postpone the flaw unless the culprit is moved far enough to come after some savior. Consequently, we can say the only possibilities for eliminating the flaw involve modifying the plan in a way that makes some culprit follow some savior. This may come about by moving the culprit or the savior or both. Constraints may force other activities to move also.

In the example, possible fixes are to modify the plan so that the -1.0 culprit (start of fast) comes after the +1000.0 savior (end of meal), or so that the -1000.0 culprit (start of meal) comes after the +1.0 savior (end of fast). In other words, the meal will come either before or after the fast. The other culprit/savior combinations of the -1.0 consumer (start of fast) after the +1.0 producer (end of fast) or the -1000.0 consumer (start of meal) after the +1000.0 producer (end of meal) are not possible fixes because of constraints.

Consider a slight modification of the example where the meal overlaps the beginning of the fasting period. In this case, the flaw occurs at the start of the fast. The roles of primary and secondary culprit are reversed with respect to the activities. However, the culprit/savior combinations and the resulting fixes are essentially the same.

A particular culprit/savior move, even if allowed by the constraints, might not eliminate or even reduce the flaw. For example, moving a culprit may force some helper to also move because of the constraints, which may render the move futile or counterproductive. If one such move does not resolve the flaw, we can try another. Consider, for example, a case where two culprits \( c1 \) and \( c2 \), each of resource value 3, are constrained to strictly precede a helper \( h \) with resource value 4. Since only one of these culprits can be moved in a single solver step, the first move will also move the helper and will increase the flaw shortfall by 1, but the second move will then decrease it by 3.

To ensure that the search is complete and systematic, instead of directly moving the culprit or savior, we add a constraint \( s \leq c \), where \( s \) is the savior and \( c \) the culprit, to enforce a suitable move. The solution grounding algorithm then produces a new reference schedule where the savior precedes the culprit. If the added constraint would be inconsistent with the existing constraints, the move is disallowed. If there are no allowed moves, then the flaw cannot be fixed and the search must backtrack and revise a previous choice, if possible. If backtracking returns to a point where we added some \( s \leq c \), we remove it, instead add the complementary constraint \( c < s \), and resume forward search. These two choices are mutually exclusive and exhaustive, fulfilling the condition for a complete and systematic search.

In the state encoding, the examples we considered show that the flaw resolution steps may, in effect, move an activity with an unsatisfied requirement either to a later or earlier interval where the desired state holds. In general, search that involves movements both backwards and forwards could potentially lead to cycles. However, that is not possible here because each flaw resolution movement comes with an added constraint that excludes the previous reference schedule. The systematic search is actually in the flexible space of constraints; the grounded schedules are used only to restrict which flaws are considered.

The above discussion and example dealt with a Boolean state space. For enumerated state spaces, there is a further issue. Consider the following sequence of three state changes and one requirement: \( \cdots \rightarrow a \rightarrow b \rightarrow c \rightarrow \text{req}(b) \). This plan has a flaw because it requires state \( b \) after a change to state \( c \). Fig. 2 shows the effect on the state resource for \( b \). This could be fixed by moving the -1000 culprit at \( c \) after the +1 savior. However, there is another possibility: reordering the sequence to be \( \cdots \rightarrow a \rightarrow c \rightarrow b \rightarrow \text{req}(b) \) also fixes the flaw, but this moves the \( c \) culprit earlier rather than later.

As noted earlier, enumerated state spaces use saturated arithmetic where repeated subtractions of 1000 have no further effect. The intuition that a flaw shortfall can only be reduced by moving a culprit after a savior depends on an assumption that the value of a sum does not depend on the order of the summands, but this is no longer true for saturated arithmetic. For example, \((-1000 - 1000 + 1000) - 1000 = 0 \), but \((-1000 - 1000) + 1000 = 1000 \). This creates an opportunity to resolve a flaw in another way: instead of moving a -1000 culprit after a savior, we can move it before a +1000 helper, which essentially “masks” the culprit. Thus, for saturated arithmetic, the solver decision method in step 3 needs to consider this option also. The “culprit after savior” and “culprit before helper” options are analogous to promotion and demotion as discussed by Chapman [10], but are simpler in the grounded schedule approach used here.

We now consider how this approach helps to avoid the issues described earlier with causal link mechanisms. At a particular point in the search, a consumer transaction may be satisfied in the sense that it is not a prime culprit for a

\[
\begin{array}{cccccc}
-1000 & +1000 & -1000 & -1 & +1 \\
a & b & c & \text{req}(b)
\end{array}
\]

Figure 2: Enumerated State Flaw
flaw in the reference schedule. In the state-based encoding, this may mean that an activity that requires a state follows a state change to that state without any intervening change to a different state. The enabling state change might then be regarded as an “achiever” for the requirement. However, unlike what happens with the causal link mechanism discussed earlier, there is no commitment on the part of the forward search to continue to use that achiever. For example, in order to resolve some flaw elsewhere, it may be necessary to put a change to a different state in between the “achiever” and the requirement. That just creates a new flaw that can be resolved in the forward search, which results in a new achiever. With the causal link mechanism, it would be necessary to backtrack to the point where the causal link selection was made in order to revise that choice.

V. Closing Remarks

We have presented a new approach to active state-based scheduling that addresses flaws left unfixed by conflict resolution, while avoiding the search pitfalls associated with traditional causal link mechanisms. The method uses a blend of grounded flaw selection and flexible flaw resolution that results in a grounded solution that can be extended to a flexible solution.

The system was tested by taking an existing multi-crew scheduling application that involved state-based planning and modifying the model so that one state requirement (crew members should be hungry before eating) was modelled using the new approach, while other requirements continued to use the existing conflict resolution method. A somewhat unexpected bonus was that the modified system took substantially fewer steps to solve the modified requirement without affecting the solution of the other requirements. Our current hypothesis to explain this is that looking for a single grounded solution is inherently an easier task than searching the flexible space (even when the latter is only doing conflict resolution). In the future we would like to evaluate this hypothesis empirically with more extensive testing.

We would also like to experiment with new ways of extending a grounded solution to a flexible one that might provide more flexibility than the method discussed earlier. One promising direction is to use the grounded solution as a precomputed oracle to influence the choices within the full flexible solving algorithm.

Our experience suggests that explicit specialization of the resource solving method to state-based reasoning would be beneficial. In particular, the set of useful culprit/savior pairs could be pruned by noting that certain combinations would not provide any benefit to the flaw. For example, suppose a flaw occurs because a particular state requirement S is not met. If some other activity that requires S occurs entirely before the flaw, it is futile to move that activity beyond the flaw because the movement of the secondary culprit at the activity start is undercut by the concomitant movement of the helper at the activity end.

In future work, we would like to dispense with the numeric encoding of states, and devise direct flaw detection and resolution methods inspired by the numeric encoding. This should bring about significant performance improvements and simplified modeling, as it allows a more compact representation for book-keeping instead of needing a separate resource for each state in the enumerated set. Also, as we reason directly on states we should be able to perform more intelligent propagation on the domains of the variables involved in the state constraints, which should translate into more efficient search.

References