Constraint-based Attribute and Interval Planning

Jeremy Frank (frank@email.arc.nasa.gov)
and Ari Jónsson * (jonsson@email.arc.nasa.gov)
NASA Ames Research Center
Mail Stop N269-3
Moffett Field, CA 94035-1000

Abstract. In this paper we introduce Constraint-based Attribute and Interval Planning (CAIP), a new paradigm for representing and reasoning about plans. The paradigm enables the description of planning domains with time, resources, concurrent activities, mutual exclusions among sets of activities, disjunctive preconditions and conditional effects. We provide a theoretical foundation for the paradigm using a mapping to first order logic. We also show that CAIP plans are naturally expressed by networks of constraints, and that planning maps directly to dynamic constraint reasoning. In addition, we show how constraint templates are used to provide a compact mechanism for describing planning domains.

1. What Should a Planner Do?

The classical definition of a planning problem is the synthesis of a sequence of activities that achieves a given set of goals. In this context, planning problems consist of a description of the world of interest, a set of operators that can be used to change the world from one state to another, a description of the state the world is in (called the initial state), and a description of the desired state of the world (called the goals). A more realistic view of planning has extended this notion to include concurrent activities that are scheduled to occur at specific times, in order to achieve results that include achieving goal conditions and maintaining conditions.

In recent years, planning has been applied to complex domains, including the sequencing of commands for spacecraft both on the ground and on-board (Jónsson et al., 2000). The domain of spacecraft operations requires controlling systems that are composed of many different primitive components. Each component may perform one and only one activity at a time, and many components have restrictions on which sequences of activities are permitted. Each activity may have both absolute and relative constraints on its start time, end time, and duration. Furthermore, activities executing on different components or subsystems may be required to interact in a variety of ways. Finally,

* Research Institute for Advanced Computer Science

resources such as memory and power are often in limited supply on spacecraft.

Until quite recently, researchers studied problems in planning represented in the STRIPS formalism (Fikes and Nilsson, 1971) or one of the various extensions thereof. In STRIPS, the world state is represented by a set of propositions, and operators change the truth values of these propositions. While this formalism is powerful and has led to numerous contributions in planning, it is difficult to represent problems involving time, resources, mutual exclusion, and concurrency in STRIPS. In order to represent time, propositions must reflect not only what is true, but when it is true. In order to represent resources, propositions reflecting each possible state of the resource must be introduced into the domain theory. In order to enforce mutual exclusions, each operator must have as preconditions an assertion that each mutually excluded state does not hold. These factors invariably lead to large numbers of propositions and domain axioms. Since STRIPS includes no inherent notion of time, it is difficult to decide what actions in a plan take place simultaneously, even should one take the trouble to create a partial-order causal-link (POCL). Additionally, it is difficult to express and meet maintenance goals in STRIPS.

The restrictive representation of STRIPS operators creates other problems. STRIPS operators cannot be used to check for illegal initial states. For example, consider the Blocks World. The initial state `On(x,table)`, `On(y,x)`, `On(z,x)` is illegal , because the intent of the operators is that only one block may be stacked on any other block. However, the operators `Move(z,table)` and `Move(w,x)` can be applied sequentially; `Move(z,table)` asserts `Clear(x)` as a consequence, even though `On(y,x)` is still true. In addition, STRIPS operators hide the sources of disjunctive preconditions, as they must be represented in separate axioms, and it is impossible to express conditional effects in STRIPS[1].

Constraint-based representations offer solutions to many of the problems that arise in static frameworks such as STRIPS. The use of variables and constraints provides representational flexibility and reasoning power that can meet the demands of domains involving time, resources, mutual exclusion and concurrency. For example, variables can represent the start and end times of an activity, and these variables can be constrained in arbitrary ways. This, in turn, is a key component of representing and reasoning about concurrent plans with absolute and relative temporal constraints. More generally, constraints can also be

[1] Extensions of the basic STRIPS formalism have provided convenient notations for disjunctive preconditions and conditional effects, but those are invariably handled by splitting the operator descriptions accordingly.

used to represent mutual exclusions, disjunctive preconditions, and conditional effects of actions. Finally, constraints can be used to represent and reason about many different types of resources.

An additional advantage of a constraint-based representation is the possibility of inheriting a host of technologies from constraint satisfaction to aid in planning. For example, techniques like no-good reasoning during search (Do and Khambhampati, 2000), domain independent heuristics (Ghallab and Laruelle, 1994; Haslum and Geffner, 2000) and linear programming (Penberthy, 1993) have already seen use in some planning systems. Finally, the underlying constraint representation permits arbitrary domain rules to be represented using procedural constraints, providing a flexible, extensible representation that can be rapidly adapted to different domains.

In this paper, we introduce Constraint-based Attribute and Interval Planning (CAIP), a planning paradigm that explicitly supports time, concurrency, resources, and mutual exclusion. CAIP is built on the notions of attributes, which describe concurrent domain components, and intervals, which describe temporal extent of activities and states. The paper is organized as follows. In §2 we formally define attributes and intervals, which are the fundamental concepts of our framework. We provide a theoretical foundation for CAIP by relating the attribute and interval representation to a first-order logical representation. We then introduce configuration rules as an expressive method to define domains models, and define the notion of valid plans in this framework. In §3 we show that CAIP plans are naturally expressed by networks of constraints, and show how planning maps directly to dynamic constraint reasoning. We also present a compact mechanism for describing CAIP planning domains using constraint templates. In §4 we discuss previous work, and in §5 we conclude and discuss future work.

2. The Constraint-Based Attribute and Interval Planning Framework

The motivation for our planning framework comes from how complex concurrent systems, such as spacecraft, are typically designed and described. The system and its interfaces are divided into components and subsystems, which we refer to as *attributes*. Each attribute represents a concurrent thread, describing its history over time as a sequence of states and activities. An *interval* describes a state or an activity with temporal extent. A plan, then, consists of a sequence of contiguous intervals for each attribute, such that the planning domain rules are

satisfied. The process of planning is based on reasoning about the values of attributes in terms of temporal intervals.

In the remainder of this section, we formally define the foundation, the key concepts and the basic semantics used in this paradigm. In the following section we build on the foundation to introduce a compact and effective constraint-based approach to representing and reasoning about candidate plans.

2.1. Intervals and Attributes

In order to then be able to plan concurrent activities and states with temporal extents, we need a representation for stating that an activity or a state extends over some period of time. We use a basic notion of a state or activity that is similar to that used by STRIPS and other formalisms for planning, in that each state or activity is an atomic predicate in a finite universe. Each predicate is defined by a unique predicate name and set of typed arguments for the predicate. Temporal intervals are a natural representation for a plan of activities and states that change over time. An *interval* specifies that a certain predicate atom holds over a certain period of time. An interval can, for example, state that `Going(loc1,loc2)` holds between time 10 and time 20.

In order to facilitate reasoning about real systems, we reason explicitly about attributes, their states and activities. Associated with each attribute is the set of possible values it can possibly take on, which are described using intervals. As an example, consider a simple domain for a planetary rover. Let us suppose we only care where the rover is, and whether the rover is in one place, moving from place to place, and possibly collecting samples with a robot arm. We can model this with a `Location` attribute, which can take on the values like `At(lander)`, and `Going(lander,hill)`, and an `Arm-State` attribute, which can take on values such as `Collect(rock,hill)`, `Idle()`, and `Off()`.

When describing an attribute, we must specify the set of possible values. It is natural to extend the specification to include the permissible value transitions. These transitions enforce common-sense rules about the domain, such as the reachable physical locations and the passing of time during the course of the plan. A value like `Going(lander,hill)` can only be reached from an `At(lander)` value and can only lead to an `At(hill)` value. A value like `At(hill)` can be reached from any possible `Going(?,hill)` and can lead to any possible `Going(hill,?)` value. Similarly, an interval that holds until time 20 can only be followed by an interval that begins at time 20. The most natural representation for such transition information is a finite state machine, which specifies the possible transitions.

Based on this, we formally define an *attribute* as a set of possible values, each of which is an interval, and a finite state machine with the value set as its alphabet.

Since each attribute can only take on a limited set of values, each interval is implicitly associated with a set of attributes, namely those for which the interval predicate atom as a possible value. The interval will therefore also specify the attribute whose value it is describing. Formally, an *interval* consists of a predicate logic statement, i.e. a predicate head and a list of applicable parameters, a start time, an end time, and the attribute to which it applies. To continue our example, the above-mentioned interval, which can be written as `holds(Location, 10, 20, Going(loc1, loc2))` will specify that the attribute `Location` takes on the value `Going(loc1,loc2)` between times 10 and 20.

As we have already mentioned, each attribute can only take on one value at any one time. This corresponds to mutual exclusion rules between intervals applying to the same attribute. More formally, let `holds(A, n, m, P)` be an interval that specifies that the attribute `A` has the value `P` from time `n` to time `m`. Let `holds(A, t, s, Q)` be another interval for the same attribute. Either the time intervals `[n,m]` and `[t,s]` are disjoint, or `P` and `Q` are the same atom.

2.2. Domain Constraints and Configuration Rules

The basic structure of a plan is a mapping of each attribute to a sequence of intervals, such that the sequence is permitted by the finite state machine for the attribute in question. The intervals in such a sequence will be contiguous, by virtue of how the finite state machine is defined. In order to constitute a valid plan, it must also satisfy other domain constraints about interactions between attribute activities. New now turn our attention to how domain constraints are defined in this framework.

In STRIPS, the domain constraints are specified in terms of operator descriptions and implicit frame axioms. For each operator (action), the description specifies what must hold immediately before the action is executed and what must hold immediately after the execution. For each fluent (state), the operator descriptions, combined with the frame axioms, specify what must happen for the fluent to become true and what can make the fluent not be true. We use a more general definition of domain constraints. The added expressivity is a necessary component of reasoning about time and concurrent interactions, as simple STRIPS extensions are simply not sufficient to describe the complexity of real-world concurrency.

The basic notion is to specify restrictions on how attribute value intervals can appear in valid plans. For a simple example, consider our simple rover. Suppose that the interval `holds(Location, 10, 20, Going(loc1,loc2))` is part of the plan. Suppose further that the arm is fragile, and thus must be turned off while the rover is traveling from one place to another. This requirement must be satisfied by inserting an appropriate interval on the `Arm-State` attribute. An interval such as `holds(Arm-State, 10,20, Off())`, satisfies this constraint. However, the interval `holds(Arm-State, 9,21, Off())` would also suffice, as would a possibly infinite number of such intervals.

We define for each possible interval I a set of *configurations* in which I legally can appear in a valid plan. Each configuration defines a set of other intervals, each of which must exist in a valid plan containing the interval I. We refer to a set of configurations as a *configuration rule*. Notice that this formalism easily provides support for disjunctive preconditions and conditional effects. We now have enough to formally specify a planning domain.

DEFINITION 2.1. *A* planning domain $\mathcal{D}$ *is a tuple* $(\mathcal{I}, \mathcal{A}, \mathcal{R})$, *where* $\mathcal{I}$ *is a set of intervals,* $\mathcal{A}$ *is a set of attributes, and* $\mathcal{R}$ *is a set of configuration rules.*

2.3. Planning Problems

The final component is to define a planning problem in this framework. In STRIPS, the problem instance definition is limited to a complete specification of the initial values of all fluents and a set of goals to be achieved at the end of the plan. This is much too limiting for reasoning about interactive, concurrent activities over time. For example, specific activity goals may be part of the overall planning problem, activities may be ongoing at the start or end of the plan, and there may be temporal components to the overall goal of the plan. A more natural planning problem specification is that a planning problem is an incomplete plan, or a disjunction of incomplete plans, and the problem is to turn an incomplete plan candidate into a valid and complete plan. In our framework, a *problem instance* is a *candidate plan*, or a set of candidate plans, which in turn is defined as a mapping of attributes to sequences of intervals that are not necessarily continuous, along with a set of non-sequenced intervals that must be part of a final plan. The goal is to find a complete plan such that all of the configuration rules in the domain are satisfied.

Notice that this notion permits a wide variety of goals, including maintenance and achievement goals. It is also possible to use this frame-

work for generating explanations by not specifying the initial state of one or more attributes.

3. Constraint-based Interval and Attribute Planning

We have formally defined a planning framework in terms of predicate instantiations, interval instances for attributes, and grounded configuration rules. While this serves to provide a solid foundation for a planning framework that is significantly more expressive than traditional STRIPS, it is not a very practical framework for solving planning problems. In this section, we turn the formal framework into a practical approach to planning, using a constraint-based representation and reasoning.

3.1. Representing Candidate Plans

The core idea of our constraint-based representation is to generalize the notion of an interval to allow variables in place of grounded values in the parameters, times, and attributes, and then use constraints on those variables to represent domain constraints. Not surprisingly, the constraint-based approach is a very effective way to enforce and reason about domain rules in this framework.

As in traditional definitions of constraint networks, a *variable* has a *domain* that specifies the set of possible values. A *constraint* has a *scope* that specifies a set of variables, and a specification defining a set of tuples that limit the valid combination of values assigned to variables in the scope.

The first part of turning our baseline framework into a constraint-based representation is to allow attribute values to be described as predicate atoms with variables. A value description therefore becomes a tuple of the form $\mathtt{p}(x_1, ..., x_k)$, where p is some predicate name and each x_i is a variable whose domain is the set of possible values defined by the type of the predicate parameter. The next step is to generalize the notion of an interval to allow predicate atoms with variables and the use of variables to describe the times and attribute. An interval now becomes a tuple of the form $\mathtt{holds}(a, t_s, t_e, P)$, where a is a variable with the set of possible attributes as its domain, t_s and t_e have the possible time values as their domains, and P is a predicate atom with variables. The last step is to introduce two extra variables h_s, h_e, to represent the *horizon* of the plan. We enforce the obvious constraints

on these variables to constrain actions in the plan to occur within the horizon [2].

As noted above, a candidate plan is a mapping of attributes to sequences of intervals, along with a set of non-sequenced intervals. The generalization to intervals with variables is straightforward, but it is worth noting that the sequence of intervals for a given attribute gives rise to a set of constraints on the time variables of each interval. To be more exact, the end time of one interval is constrained to be less than or equal to the start time of the following interval.

3.2. COMPATIBILITIES

Having generalized the representation of candidate plans, we find that constraints can be used to significantly compress the specification of domain rules, i.e., configuration rules. Consider the example of our rover, where the arm is restricted to be off whenever the rover is moving. We noted that this gives rise to a large number of configurations that satisfy this restriction. Using constraints and variables, however, we can reduce this set to a single expression of a domain rule. In essence, the rule will say that for any interval of the form $\texttt{holds}(\texttt{Location}, s, e, \texttt{going}(x, y))$, there exists another interval $\texttt{holds}(\texttt{Arm} - \texttt{State}, s', e', \texttt{off}())$, such that $s' \leq s$ and $e \leq e'$. This notion of specifying constraint rules is generalized to a construct called a *compatibility*.

3.2.1. *Compatibilities*

The idea behind compatibilities is to provide a compact representation of the constraints that the attribute definitions and configuration rules impose on valid plans. Their basic structure is similar to the configuration rules, in terms of specifying that certain intervals must exist in order for a given interval to appear in a valid plan. The remaining issue is how to fold the attribute definitions into the compatibility structure.

Each attribute definition has two components, the set of legal values, and the permitted transitions. The set of possible values corresponds to a set of legal combinations of value assignments to interval variables, which in turn can be represented by a set of constraints. For example, suppose each `Going(loc1,loc2)` interval must hold for 40 time units. This can easily be turned into a constraint on any interval of the form $\texttt{holds}(\texttt{Location}, s, e, \texttt{Going}(\texttt{loc1}, \texttt{loc2}))$, simply by requiring that $e - s = 40$. The limitations on transitions are easily encoded in the same manner as the configuration rules.

Based on these observations, we determine that a compatibility has to specify the set of intervals to which it applies, the constraints on valid

[2] There is a slight exception to this last case, discussed in §3.3

variable combinations, and the valid configurations. For reasons that will become clear, it is beneficial to describe compatibilities by referring to variables rather than direct matches with value sets. Consequently, a simple compatibility is structured as follows:

Head: $\texttt{holds}(a, t_1, t_2, P(x_1, \ldots, x_k))$
Guards: **list-of** $v_i \in G_i$
Parameter Constraints: **list-of** $C_j(Y_j)$
Disjunction of Configuration Rules: **list-of**
Configuration: **list-of**
  Temporal Relation: τ
  Configuration Interval: $\texttt{holds}(b, t_3, t_4, Q(z_1, \ldots, z_n))$
  list-of Configuration Constraints $K_m(W_m)$

Again, we utilize variables and constraints to make the representation more effective. Each of the guards specifies a domain G_i for an interval variable, i.e. a variable in the set $\{a, t_1, t_2, x_1 \ldots x_k\}$. The guards specify the set of intervals to which the rest of the compatibility applies. Each C_j is a constraint on the variables of the interval, restricting the value combinations to those permitted by the attribute definition. Each configuration rule defines a list of possible configurations. Each configuration in turn consists of a set of interval templates, described by the interval specification, constrained in their relation to the P interval by the temporal relation τ and the general constraints K_m, each of which has a scope W_m that combines variables from the parameters of the interval for P and the parameters of the interval for Q, that is, the set $\{x_1 \ldots x_k, z_1 \ldots z_n\}$. The temporal relation τ can be expressed in terms of Allen's interval algebra(Allen and Koomen, 1983), augmented with metric interval distance bounds.

It is worth pointing out that the structure of constraints relating the P interval and a Q interval is deliberate. The general constraints are limited to the parameters of the predicate atom, while the temporal relation provides distance bounds on the distances between the temporal variables $t_1, \ldots, t_4$. The consequence of this is that the temporal variables and the constraints connecting those form a simple temporal network, which in turn gives us nice computational complexity behavior when it comes to constraint reasoning.

3.2.2. *Properties of Compatibilities*
Any configuration rule can be expressed as a compatibility with a grounded head and a disjunction of grounded interval sets, which means that the expressivity is the same as in the theoretical framework. However, the use of compatibilities will invariably lead to an exponentially

smaller encoding of the domain constraints. Furthermore, since the compatibilities simply express sets of configuration rules, the semantics of a compatibility are fairly clear:

DEFINITION 3.1. *If attribute A takes on the value $P(X)$, and all of the guard constraints $v_i \in G_i$ are satisfied, then the following must hold:*

- *Each parameter constraints $C_i(Y_i)$ is satisfied.*
- *There is a configuration such that for each of its interval templates there exists a corresponding interval $Q_i(A)$ in the plan, such that the temporal relation $P(X)\tau_i Q_i(A)$ and the $K_m(W_m)$ constraints hold.*

It should be noted that multiple compatibilities may be applicable to a given interval. That, combined with the use of guards to determine applicability, allows conditional constraints and configuration rules to be specified with compatibilities. This is an important extension to the more traditional non-conditional STRIPS rules, as conditional constraints appear frequently in real-world domains. In fact, the compatibility mechanism is powerful enough to express a variety of complex plan constraints including conditional effects, disjunctive preconditions, and arbitrary parameter relations.

3.2.3. *Example: A Simple Compatibility*
To ground this notion, let us again consider the rover domain example, in particular the `Going` interval. Recall that we have suggested a number of restrictions on `Going`:

- The rover must be `At` the location it begins `Going`, and must end up `At` the location it terminates `Going`.
- The travel time depends on the departure point and the destination; let us assume this time is specified by a function named `travelTime`.
- While the rover is traveling, the arm must be `Off`.

The compatibility for `Going` is then specified as follows:

Head: holds(location,s_g, e_g,Going(x, y))
Parameter Constraints: $s_g + travelTime(x, y) = e_g$
Configuration Rule:
  met_by holds(location,s_{a1}, e_{a1},At(a)), $a = x$
  met_by holds(location,s_{a2}, e_{a2},At(b)), $b = y$
  contains holds(arm,s_o, e_o,Off)

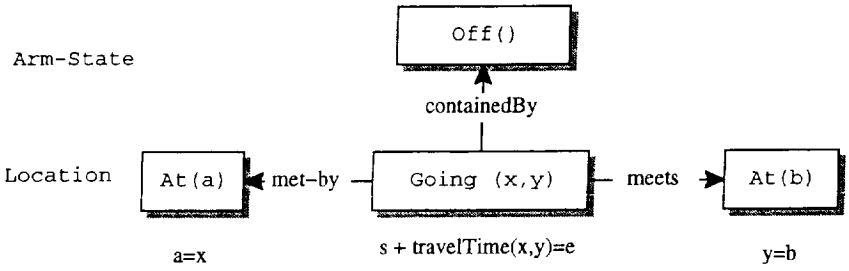

Figure 1. A graphical picture of a simple compatibility for Going(x,y). The parameter constraint is shown beneath the Going interval.

Figure 1 shows this pictorially.

3.2.4. *Disjunctions as Conditional Compatibilities*
In the above definition of compatibilities we have retained the explicit disjunction from configuration rules. However, we can represent disjunctions by using explicit variables and associated constraints, which in turn makes it possible to express more information in terms of constraint-based representation. Assuming that parameters of the predicates express the possible disjunctive choices [3], a set of compatibilities on the same interval can specify disjunctions. To see how this works, assume we have a more sophisticated rover model, in which turning the rover is modeled explicitly. Now, Going can be preceeded by At or Turning. This requires deciding whether or not to turn before traveling to the next location. The Going predicate can be augmented with variables representing the decisions about whether Turning or At precedes and follows the Going. These variables are used in compatibility guards to specify which configuration rules apply. By choosing the guards so that only one is satisfied at a time, only one set of configuration rules will be enforced. The compatibilities are shown here below, and Figure 2 shows the structure of the compatibilities graphically.

Head: holds(a,s_g, e_g,Going(x, y, p, f))
Parameter Constraints: $s_g + travelTime(x, y) = e_g$
  Configuration Rule:
    contains holds(b,s_o, e_o,Off) Head: holds(a,s_g, e_g,Going(x, y))
Parameter Constraints: p =at-bef-go
Configuration Rule:
  met_by holds(b,s_a, e_a,At(a)), $a = x$, a = b
Head: holds(a,s_g, e_g,Going(x, y))
Parameter Constraints: p =turn-bef-go

[3] This is notational convenience only; the disjunctive variables can be defined anywhere in the scope of the compatibility.

Configuration Rule:
    met_by holds(b,s_t, e_t,Turning(a)), $a = x$, a = b
Head: holds(a,s_g, e_g,Going(x, y))
Parameter Constraints: f =at-aft-go
Configuration Rule:
    meets holds(b,s_a, e_a,At(a)), $b = y$, a = b
Head: holds(a,s_g, e_g,Going(x, y))
Parameter Constraints: f =turn-aft-go
Configuration Rule:
    meets holds(b,s_t, e_t,Turning(a)), $a = x$, a = b

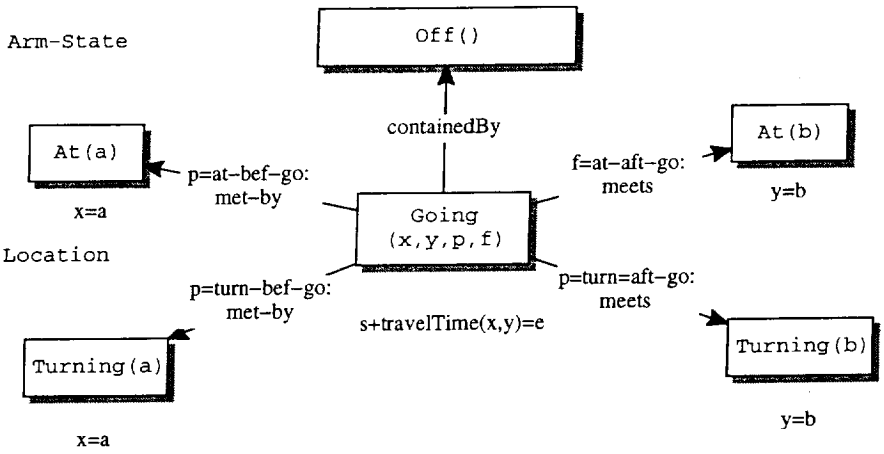

Figure 2. A graphical picture of a disjunctive compatibility for Going(x,y). The constraints are shown beneath the Going interval, and the conditional configuration rules are shown as temporal relations and intervals that must exist in the plan. Attribute equivalences are omitted.

The advantage of using conditional compatibilities rather then explicit disjunctions is twofold. First, the explicit use of variables to specify choices is a more natural and more effective representation in constraint-based reasoning. Secondly, the application and semantics of compatibilities are simplified. Since no representational power is lost, we can safely assume that conditional compatibilities are used in place of disjunctions for the remainder of this section.

3.3. Building Plans

In the constraint-based representation we have introduced, the notion of a candidate plan can be extended to include intervals with unbound variables. A candidate plan is therefore a mapping of attributes to

sequences of intervals, along with a set of non-sequenced intervals, where each interval describes a wide range of possible interval instantiations. We now turn our attention to describing how candidate plans are modified, with the goal of mapping them into valid plans.

We first define the set of valid plans that are *completions* of a given candidate plan. Given a candidate plan P_C, any valid plan P such that there is a 1-1 mapping between the intervals in P_C and an interval in P with the same predicate but with all of the variables grounded, is a valid completion. Clearly, the set of valid plans may be empty, which indicates that the candidate plan is invalid. In general, it is intractable to identify invalid candidate plans, but it is easy to see that if any constraint is violated by a candidate plan, the candidate is invalid. Such candidate plans are *inconsistent.*

Two things distinguish candidate plans from valid plans. One is that of intervals that need to appear in the plan, but are not necessarily in place yet. The other is that of unbound variables, which represent choices in the exact specification of intervals. Not surprisingly, these two issues determine the choices to be made when completing a plan. Planning proceeds by selecting from these choices, and modifying the candidate plan. As this is done, we must assess the impact of these changes on the plan by checking the compatibilities relevant to the interval that was changed. This results in a distinction between candidate plans.

A candidate plan is *full*, if the following is satisfied: For every compatibility that is applicable to an interval on an attribute, the associated parameter constraints have been posted on the interval variables, and for every configuration rule, there exists a matching interval such that the temporal relation and the other constraints are posted between variables of the two intervals. Given this, it is easy to turn a given candidate plan into a full candidate plan, by posting any missing parameter constraints, and adding any missing intervals along with the necessary constraints. The added intervals are added as non-sequenced intervals.

We can now define the operations that modify plans, and can be used to search for a valid complete plan. Since these add decisions to th plan, and thus potentially reduce the set of valid plan completions, the operations are called *restrictions*:

- Putting an unsequenced interval into an attribute sequence. This can be accomplished in two different ways:
  - An interval can be *inserted* between two intervals on an attribute, along with the implied ordering constraints that relate the start and end times of the intervals involved. Notice

that if there is insufficient time to insert the new interval, the mutual exclusion constraints will be violated.

- An unsequenced interval can be *equated with an interval* in the attribute sequence. This requires that the predicates of the intervals be identical, and that the corresponding interval variables be pairwise equated.

– The domain of a variable can be restricted. The obvious choices are to assign a value to unassigned variables, but it is also possible to reduce the set of possible values.

The inverse of these operations are called *relaxations.*

In terms of constraint-based reasoning, the restriction and relaxation operations map directly into the notions of strengthening and weakening of constraint networks, as those notions are defined for dynamic constraint problems. As we will see here below, this is one of the key strengths of this approach, as there are many well-known techniques available to reason effectively about dynamic constraint networks.

Whenever a plan is modified, we must make the candidate plan a full candidate before proceeding. This is referred to as the *plan invariant.* Enforcing this plan invariant is done as follows:

– Restriction: any configuration rules that apply to the candidate plan, and are not already enforced, are enforced by adding the corresponding intervals and constraints. For example, an interval may now be on in an attribute sequence, leading to applicability of a compatibility. Another example is that a variable's domain may now match a compatibility guard guard, leading to the applicability of a configuration rule.

– Relaxation: any configuration rules that no longer apply result in the removal of the relevant intervals and constraints. For example, a compatibility guard that previously applied may no longer apply, resulting in the removal of constraints and intervals from the plan.

Let us return to the complex rover model once again for an example. Suppose we have a plan with `holds(Location,` s, e, `Going(x,y,p,f)` on the `Location` attribute , and the assignment p =`turn-bef-go` is made. The configuration guards are checked. Since a variable assignment was made, we only check the guards relevant to variable p. The guard p =`turn-bef-go` is satisfied, and so a non-sequenced interval `holds(location,` s', e', `Turning(a)` is added to the plan, along with the constraints $a = x$ and $e' = s$ (which enforces `Going(x,y)` met_by `Turning(a)`). Now suppose that the assignment p =`turn-bef-go` is

retracted. The configuration guard is no longer satisfied, and so the interval for `Turning(a)` and the attendant constraints are removed from the plan. This process is illustrated in Figure 3.

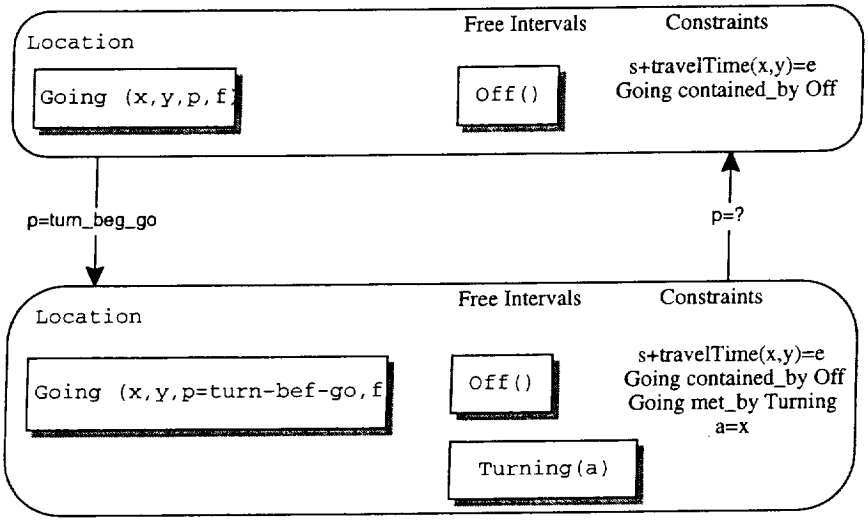

Figure 3. Illustration of the plan invariant at work. When the assignment p =turn-bef-go is made, free intervals and constraints are added to the plan. When it is retracted, the configuration guard no longer holds, and the `Turning` interval and its constraints are removed from the plan.

There remains one complication in the process for building plans with activities and states that have temporal extent, which is related to the notion of a *planning horizon*. The purpose of a planning horizon is to limit the plan being built to the given temporal interval. Consequently, the planning effort is limited to those intervals that necessarily intersect the horizon. Now suppose that it is *possible* for an interval to be wholly contained outside the horizon. Such an interval need not be a part of the plan, and thus does not have to be sequenced on an attribute. This notion is important, even when the planning horizon is infinite, as it distinguishes between an interval following a finite interval and an interval following a possibly infinite interval.

As a final note on the tools we have defined for building valid plans, it is worth pointing out that the definition of a valid plan can be relaxed to allow some uninstantiated variables and unsatisfied interval constraints. The reason for allowing this is that the agent executing the plan may well be intelligent enough to complete the plan on the fly, which in turn allows for more flexibility during execution. This relaxation can be done by defining a plan identification function that indicates whether a plan is sufficiently complete or not. The full details on this generalization can be found in (Jónsson et al., 2000).

3.4. Planning correctness

In the preceding section we formalized the notion of a valid plan, and in this section we have presented a constraint-based representation and reasoning mechanism for specifying candidate plans and transforming those into valid plans, when possible. The only remaining issue is to show that the operations proposed here above are complete, in the sense that they can be used to build any given valid plan completion. It turns out that as long as no attribute is mapped to an empty sequence in the initial candidate plan, the operations are complete. We will now outline why this is the case; a formal proof can be found in (Jónsson et al., 2000).

Consider a candidate plan P_C and a valid completion P. For each attribute, the sequence of intervals in P is permitted by the associated finite state machine. Assuming that conditional compatibilities are used in place of disjunctions, we note that a fully grounded interval uniquely determines the previous and following interval. For a given attribute, let I be an interval on the attribute in P_C. We now instantiate each unassigned variable, in accordance with the instantiation in the final plan P, which automatically gives us the unsequenced predecessor and successor intervals. We then sequence those in the same way as they are in the final plan P. This is repeated until the attribute has the same interval sequence as in the final plan. By repeating this for each attribute separately, we can turn P_C into the complete plan P. By definition, all other domain constraints are satisfied in P.

3.5. Constraint reasoning

We have already noted that each candidate plan gives rise to a constraint network, and that the operations to restrict and relax plans map directly to strengthening and weakening operations for constraint networks. This makes it possible to bring results from the wide literature on CSPs to bear on the constraint networks.

There are many constraint reasoning techniques that can be used to make the planning effort more effective. The only limitation is that the constraint reasoning methods preserve the set of valid plan completions. For consistency-maintenance techniques such as the temporal constraint propagation, arc consistency maintenance, higher-level consistency enforcement, and the correct procedure application, as described in the procedural framework , this is indeed the case. The reason is straightforward; these techniques never eliminate values or value combinations that could be part of a solution to the constraint network instance, which in turn means that the addition of any such value or value combination would have made the candidate plan invalid.

Among the most useful of these techniques are those that efficiently reduce the domains of variables. The temporal constraints can be addressed using the algorithms described in (Dechter et al., 1991). Arithmetic constraints can be enforced using bounds consistency techniques (Marriott and Stuckey, 1998). However, many of the constraints will be specific to the particular planning domain. Consequently, a general constraint reasoning framework is required. We use the *procedural constraints* framework (Jónsson, 1997). In this framework, each constraint is embodied as a procedure. The benefits of this are an efficient and compact representation of the constraints, as each procedure can take advantage of specific techniques. The framework is extensible, as new constraints can be easily added. The framework requires each procedure to provide a definitive answer when all variables in its scope are assigned singleton values. However, procedures can do much more, such as enforce arc consistency or bounds consistency. Continuous valued variables can be in the scope of any constraint, as long as they are *dependent variables*, that is, their values are a function of the discrete variables in the constraint.

Resources are easily represented using our framework. Unary resources can be directly handled by attributes, since it is sufficient to enforce binary mutual exclusion. Some discrete multi-capacity resources can be represented using multiple attributes of the same type; new intervals may be inserted onto any appropriate attribute. Continuous resources can be modeled by using a single attribute with a parameter to represent either the use or capacity of the resource. Arithmetic constraints associated with each interval dictate how the resource is affected. The key issue in managing resources in planning is to be able to effectively reason about the impact of the current state of the resource on the plan. Techniques such as edge-finding (Nuijten, 1994) and the balance constraint (Laborie, 2001) can both determine the state of resource consumption, and add other constraints to the plan to ensure that adequate resources are available. Fully integrating these techniques into a planning framework is the subject of future work.

4. Previous work

Allen and Koomen (Allen, 1991; Allen and Koomen, 1983) developed a sophisticated framework for representing time and temporal plans, much of which has been adopted by later researchers (including ourselves) as the representation for planning. However, no planners based on this formalism were developed, and the framework developed does not include completeness results for planning domains.

Our work builds heavily on two prior approaches to planning with time and attributes. The Remote Agent Planner (RAP) (Jónsson et al., 2000), which is derived from HSTS (Muscettola, 1994), and IxTeT (Laborie and Ghallab, 1995; Ghallab and Laruelle, 1994), are planners that handle time and resources, as well as supporting mutual exclusion through attributes. Both were developed to work on real-world problems involving planetary rover operations. Concurrent plans are produced by defining what values the attributes take on. IxTeT uses a point-based representation of time while RAP uses an interval-based representation. IxTeT has sophisticated resource representation and reasoning capabilities built into the planner infrastructure (Ghallab and Laruelle, 1994). In addition, mutual exclusion on IxTeT attributes is handled via a threat mechanism similar to that used in POCL planning, while the approach in RAP is to explicitly order subgoals on attributes. RAP depends on the `met-by` and `meets` constraints to ensure attributes have a value at all times, while IxTeT uses events at the start and end of activities. Finally, IxTeT cannot support disjunctive relationships between activities. The CAIP framework can be viewed as providing a sound theoretical justification for these planning systems.

DEVISER (Vere, 1983) is a POCL planner that handles time [4]. DEVISER domain descriptions can include absolute temporal constraints and duration constraints; the duration of an activity can be an arbitrary function of any of the parameters of an activity. Goals can be expressed using both absolute temporal constraints and relative temporal constraints; for instance, it is possible to assert that A and B are both true simultaneously. In DEVISER, all additions and deletions occur at the end of the activity. Modeling techniques are described to model activities in which a precondition need not be true throughout the action and to model an activity in which an effect takes place immediately. Both require adding new fluents to the representation, and there is no easy way to introduce related activities that start or end at times arbitrarily related to the activity in question.

ZENO (Penberthy, 1993) and Descartes (Joslin, 1996) are POCL planners that handle time. Both are built on the notion of intervals (Temporally Quantified Assertions (TQAs) in Descartes and in ZENO. Descartes allows arbitrary constraints among the parameters of TQAs. In ZENO, continuous variables are allowed to vary in a piecewise linear manner; this forces the modeling of other constraints as piecewise linear. Neither ZENO nor Descartes support mutual exclusion, and lack theoretical justification for their extensions of STRIPS.

[4] Deviser is actually based on NONLIN and NOAH, which pre-date POCL planning

TGP (Smith and Weld, 1999) is a version of Graphplan that handles a version of STRIPS with time. Activities are assumed to have duration, and can also have absolute temporal constraints on the start and end times. This is coupled with the following extension to the semantics of STRIPS: All preconditions are required to hold before the action begins. All preconditions unaffected by the action are required to hold until the action ends. Effects are required to hold after the action ends. These semantics are similar to those found in DEVISER; TGP is more limited than DEVISER in that activity durations must be part of the model. The CAIP framework is more expressive, in that arbitrary synchronizations between actions can be expressed. TGP also does not support attributes or resources.

5. Conclusions and Future Work

We have presented CAIP, a planning framework that supports features common to real planning problems. CAIP provides primitives that supports modeling domains with real time, concurrency, resources, mutual exclusion, and disjunctions. Intervals representing a temporally extended state provide a basis for constraining the timing and concurrency of activities. Attributes enforce mutual exclusion and support the modeling of resources. The underlying constraint-based representation permits compact representation of these rules, supports disjunctions, and also allows planning technology to leverage off of efficient algorithms for constraint satisfaction problems.

The framework leaves a variety of implementation details unspecified. For example, a wide variety of constraint reasoning algorithms such as arc consistency or bounds consistency can be used to quickly identify and eliminate values of variables that can lead to invalid plans. Special reasoning algorithms can be used for domain specific constraints. In some cases, collections of constraints may be reasoned about simultaneously; ZENO uses an incremental Simplex algorithm to manipulate linear constraints (Penberthy, 1993). Sophisticated resource reasoning algorithms such as edge-finding (Nuijten, 1994) and balance constraints (Laborie, 2001) can also be used. However, these require matching attributes with resources for particular domain models.

In the CAIP framework, an interval on an attribute can force other intervals to exist on other attributes. Another option is to constrain the intervals that other attributes may take on. In essence, this would permit the expression of negation constraints on attributes. HSTS permitted the posting of constraints limiting the possible intervals that could occur on an attribute within a period of time (Muscettola, 1994).

Modeling experience with the CAIP framework will indicate whether such expressive power is needed by the framework, and how best to incorporate it.

References

Allen, J.: 1991, 'Planning as Temporal Reasoning'. In: *Proceedings of the Second Conference on Knowledge Representation.*

Allen, J. and J. Koomen: 1983, 'Planning using a Temporal World Model'. In: *Proceedings of the Eighth International Joint Conference on Artificial Intelligence.*

Dechter, R., I. Meiri, and J. Pearl: 1991, 'Temporal Constraint Networks'. *Artificial Intelligence* **49**, 61–94.

Do, M. B. and S. Khambhampati: 2000, 'Solving Planning-Graph by Compiling It Into CSP'. In: *Proceedings of the Fifth International Conference on Artificial Intelligence Planning and Scheduling.* pp. 82–91.

Fikes, R. E. and N. J. Nilsson: 1971, 'STRIPS: A new approach to the application of theorem proving to problem solving'. *Artificial Intelligence* **2**((3-4)).

Ghallab, M. and H. Laruelle: 1994, 'Representation and Control in IxTeT, a Temporal Planner'. In: *Proceedings of the 2d Conference on Artificial Intelligence Planning And Scheduling.* pp. 61–67.

Haslum, P. and H. Geffner: 2000, 'Admissible Heuristics for Optimal Planning'. In: *Proceedings of the Fifth International Conference on Artificial Intelligence Planning and Scheduling.* pp. 140 –149.

Jónsson, A.: 1997, 'Procedural Reasoning in Constraint Satisfaction'. Ph.D. thesis, Stanford University Computer Science Department.

Jónsson, A. K., P. H. Morris, N. Muscettola, K. Rajan, and B. Smith: 2000, 'Planning in Interplanetary Space: Theory and Practice'. In: *Proceedings of the Fifth International Conference on Artificial Intelligence Planning and Scheduling.*

Joslin, D.: 1996, 'Passive and Active Decision Postponement in Plan Generation'. Ph.D. thesis, Carnegie Mellon University Computer Science Department.

Laborie, P.: 2001, 'Algorithms for Propagating Resource Constraints in AI Planning and Scheduling: Existing Approaches and New Results'. In: *Proceedings of the 6th European Conference on Planning.*

Laborie, P. and M. Ghallab: 1995, 'Planning with Sharable Resource Constraints'. In: *Proceedings of the International Joint Conference on Artificial Intelligence.* pp. 1643 – 1649.

Marriott, K. and P. Stuckey: 1998, *Programming with Constraints: An Introduction.* The MIT Press.

Muscettola, N.: 1994, 'HSTS: Integrated Planning and Scheduling'. In: M. Zweben and M. Fox (eds.): *Intelligent Scheduling.* Morgan Kaufman, pp. 169–212.

Nuijten, W.: 1994, 'Time and Resource Constrained Project Scheduling: A Constraint Satisfaction Approach'. Ph.D. thesis, Eindhoven University of Technology Department of Mathematics and Computer Science Department.

Penberthy, S.: 1993, 'Planning with Continuous Change'. Ph.D. thesis, University of Washington Department of Computer Science and Engineering.

Smith, D. E. and D. S. Weld: 1999, 'Temporal Planning with Mutual Exclusion Reasoning'. In: *IJCAI.* pp. 326–337.

Vere, S.: 1983, 'Planning in Time: Windows and Durations for Activities and Goals'. *Pattern Matching and Machine Intelligence* **5**, 246–267.