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 describe Constraint-based Attribute and Interval Planning (CAIP), a 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, based on temporal intervals and attributes. We then show how the plans are naturally expressed by networks of constraints, and show that the process of planning maps directly to dynamic constraint reasoning. In addition, we define compatibilities, a compact mechanism for describing planning domains. We describe how this framework can incorporate the use of constraint reasoning technology to improve planning. Finally, we describe EUROPA, an implementation of the CAIP framework.

1. What Should a Planner Do?

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 the allowed sequences of activities. 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 mutually constrained in a variety of ways. Finally, resources such as memory and power are often in limited supply on spacecraft, imposing further restrictions on activity sequences.

Until quite recently, researchers studied problems in planning represented in propositional formalisms such as STRIPS (Fikes and Nilsson, 1971) or Planning Domain Definition Language (PDDL) (McDermott, 2000). In STRIPS, the world state is represented by a set of propositions, and operators change the truth values of these propositions. While these formalisms are powerful and have led to numerous contributions in planning, it is difficult to represent problems involving

* Research Institute for Advanced Computer Science

time, resources, mutual exclusion, and concurrency using propositions. 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 propositions include no inherent notion of time, it is difficult to decide what actions in a plan take place simultaneously, even when using a Partial-Order Causal-Link (POCL) planner. Finally, it is difficult to express and meet maintenance goals in a propositional framework.

The restrictive representation of STRIPS operators creates other problems. STRIPS operators alone cannot be used to check for illegal initial states. For example, consider the Blocks World `Move` operator, which repositions a block. A precondition of this operator is that the intended destination block `b` be available, represented by the predicate `Clear(b)`. 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 [1]. 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. 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, which makes for hard-to-read models.

Constraint-based representations offer solutions to many of the problems that arise in frameworks such as STRIPS. The use of variables and constraints provides representational flexibility and reasoning power. For example, variables can represent the start and end times of an activity, and these variables can be constrained in arbitrary ways. The ability to model activities this way is a key component of representing and reasoning about concurrent plans with absolute and relative temporal constraints. More generally, constraints can also be 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.

[1] PDDL handles this by explicitly introducing domain axioms.

An additional advantage of a constraint-based representation is that a wide range of constraint reasoning techniques and results can be brought to bear on planning problems. For example, techniques like consistency enforcement (Joslin, 1996; Jónsson et al., 2000), no-good reasoning during planning (Do and Khambhampati, 2000), satisfiability techniques (Kautz and Selman, 1996), domain independent heuristics (Ghallab and Laruelle, 1994; Haslum and Geffner, 2000) and the use of constraint reasoning systems employing linear programming (Penberthy, 1993) have been used in different planners. By using a constraint-based representation, we continue the trend set with these earlier efforts.

In this paper, we describe 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 threads of activity, and *intervals*, which describe temporally extended activities and states. Our goal in this paper is to describe a paradigm that makes it easier to both specify complex planning domains and create planners that take advantage of constraint reasoning technology.

Our paradigm builds on prior approaches to planning with time and attributes, the Remote Agent Planner (RAP) (Jónsson et al., 2000) and IxTeT (Laborie and Ghallab, 1995; Ghallab and Laruelle, 1994). RAP is derived from an earlier planning system, HSTS (Muscettola, 1994). Both were developed to work on real-world problems involving space operations. Both are attribute-centric, in that plans consist of attributes that take on changing values over time. IxTeT uses a point-based representation of time while RAP uses an interval-based representation; the latter are similar to the Temporally Qualified Assertions (TQAs) first introduced by Penberthy (Penberthy, 1993) and later used by Joslin (Joslin, 1996). IxTeT has sophisticated resource representation and reasoning capabilities built into the planner infrastructure (Ghallab and Laruelle, 1994). Mutual exclusion on IxTeT attributes is handled via a threat mechanism similar to that used in POCL planning, while in RAP attributes require distinct intervals on the same attribute to be totally ordered. Both RAP and IxTeT require that every time instant maps to some interval. Finally, RAP supports disjunctive relationships between activities, while IxTeT does not. The Constraint-based Attribute and Interval framework provides the necessary link between representations like these and more traditional propositional representations of planning domains.

The paper is organized as follows. In §2 we formally define attributes and intervals, which are the fundamental concepts of our framework. We define the grounded logic of domain constraints and plans in terms

of attributes and intervals, which provides a theoretical foundation for the constraint-based approach. In §3 we define the constraint-based approach to planning over intervals and attributes, show that CAIP plans are naturally expressed by networks of constraints, and show how planning maps directly to dynamic constraint reasoning. We present a compact mechanism for describing CAIP planning domains using constraint templates called *compatibilities*. We also show that the resulting framework is both *correct* and *complete*. In §4 we then discuss how various advanced constraint reasoning techniques can be used to take advantage of the CAIP representation. In §5 we briefly describe EUROPA, an implementation of the CAIP framework. In §6 we discuss previous work, and in §7 we conclude and discuss future work.

2. The Attribute and Interval Planning Framework

The motivation for our planning framework comes from the design of complex concurrent systems, such as spacecraft. 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. The rules governing how subsystems act and interact form the basis of the *planning domain constraints*. A *plan* consists of a sequence of intervals for each attribute, such that the planning domain constraints are satisfied. The process of planning is based on reasoning about temporal intervals that make up the sequence of values of attributes. In the remainder of this section, we formally define these key concepts and the basic semantics used in the framework.

2.1. Intervals and Attributes

In order to plan concurrent activities and states with temporal extents, we need to represent the fact 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(rock,lander)` holds between time 10 and time 20.

In order to facilitate reasoning about real systems, we reason explicitly about attributes and their states. Associated with each attribute is the set of values it can possibly take on, which are described using intervals. As an example, consider a simple domain for a planetary rover with a robot arm. Suppose we only care where the rover is, and whether or not the rover is collecting samples with the arm. We model this with a `Location` attribute, which can take on values like `At(lander)` and `Going(lander,hill)`, and an `Arm-State` attribute, which can take on values such as `Collect-Sample(hill)`, `Idle()`, and `Off()`.

Based on this, we formally define an *attribute* as a mapping from time to a set of possible values, each of which is an interval. Sequences of non-overlapping intervals are a convenient representation of this mapping. Problem instances can contain multiple instances of attributes, so intervals also specify the attribute instances whose value they describe. We will represent the attribute instance with an additional parameter. 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 instance to which it applies. To continue our simple rover example, the above-mentioned interval, which can be written as `holds(Location-Instance-1, 10, 20, Going(rock,lander))` specifies that instance `Location-Instance-1` of attribute `Location` takes on the value `Going(rock,lander)` at time 10, and this value holds until time 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 that apply 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 instance to a sequence of intervals. In order to constitute a valid plan, it must also satisfy domain constraints that limit the interactions between activities.

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 false. Simple STRIPS extensions are not

sufficient to describe concurrent domains, for example delayed or transient effects. STRIPS also does not handle disjunctions elegantly. For these reasons, we need a more sophisticated way of expressing domain constraints.

A domain constraint must describe the necessary conditions under which an interval can hold on an attribute in a valid plan. For a simple example, consider our rover. Suppose that the grounded interval `holds(Location, 10, 20, Going(rock,lander))` 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 means that there must be an appropriate interval on the `Arm-State` attribute that ensures the arm is `Off` while the rover is `Going`. 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* O_i in which I legally can appear in a valid plan. Each configuration defines a conjunction of other intervals, J_{ik}, all of which must exist in a valid plan containing the interval I. We refer to a disjunction of possible configurations as a *configuration rule*. Logically, a configuration rule is an implication $I \Rightarrow O_1 \vee O_2 \vee ...O_i$. Notice that this formalism easily provides support for disjunctive preconditions and conditional effects.

We can now formally define 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

In order to complete the framework, we must define planning problem instances and their solutions. 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. It is more natural to require that a planning problem instance be an incomplete plan, and the problem is to turn an incomplete plan candidate into a valid and complete plan. Unsequenced intervals can be part of an incomplete plan; these are assertions that some activity take

place, without a commitment of how or in what order the activities occur. We formalize these definitions below:

DEFINITION 2.2. *A* candidate plan *for a planning domain $\mathcal{D}$ is a mapping from attributes to sequences of intervals and a set of non-sequenced intervals. Given a candidate plan P_C, a* plan extension *is a plan P such that there is a mapping from the intervals in P_C to a subset of the intervals in P, and the mapping maps each interval in P_C to an identical interval. A* valid plan *is a candidate plan such that for each interval I, and for each configuration rule R that matches I, there is a configuration O in R for which all of the intervals in O are also part of the plan.*

The job of a planner is to find an extension of the initial plan such that all of the configuration rules in the domain are satisfied for all intervals. This notion permits a wide variety of goals, including maintenance and achievement goals. It is also possible to use this framework for generating explanations by not specifying the initial state of one or more attributes. With this notion we can now complete the framework by defining a planning problem instance:

DEFINITION 2.3. *A* planning problem instance *is a tuple $(\mathcal{D}, P_C)$, where $\mathcal{D}$ is a planning domain and P_C is a planning problem instance. A* solution *to the instance is a plan P that is a valid extension of P_C.*

3. Constraint-based Interval and Attribute Planning

We have now formally defined a planning framework in terms of predicate instantiations, interval instances for attributes, and grounded configuration rules. While this provides a solid foundation for a planning framework that is significantly more expressive than traditional STRIPS, the number of grounded intervals and configurations in configuration rules may be very large. Thus, it is not a practical framework for solving planning problems. In this section, we turn the formal framework into a practical approach to planning.

3.1. Representing Candidate Plans

We will represent candidate plans as a *network of constraints*. 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. 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 the set of valid combinations of values assigned to the variables in the scope.

In order to employ a constraint-based representation, we generalize the notion of an interval to allow predicate atoms with variables and the use of variables to describe the times and attribute. We will use *italics* to denote variables, and `typewriter` to denote constants and predicate names. An interval now becomes a tuple of the form $\texttt{holds}(a, t_s, t_e, P)$, where a is a variable with the set of possible attribute instances as its domain, and t_s and t_e have the possible time values as their domains. Predicate atoms P take the form $\texttt{p}(x_1, ..., x_k)$, where $\texttt{p}$ 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.

We introduce two extra constants h_s, h_e, to represent the *horizon* of the plan. The obvious requirement that actions in the plan occur between the horizons has an added complication, discussed in detail in §3.3.

As noted above, a candidate plan is a mapping of attributes to sequences of intervals, a set of non-sequenced intervals, and a set of constraints among the variables representing the 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 temporal 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.

Figure 1 shows a fragment of a plan and its CSP representation.

3.2. Compatibilities

Having generalized the representation of candidate plans, we find that constraints can be used to significantly compress the specification of 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, compact expression. In essence, the rule will say that for any interval of the form $\texttt{holds}(\mathit{Location}, s, e, \texttt{Going}(x, y))$, there exists another interval $\texttt{holds}(\mathit{ArmState}, s', e', \texttt{off}())$, such that $s' \leq s$ and $e \leq e'$. This notion of specifying constraint rules is generalized to a construct called a *compatibility*.

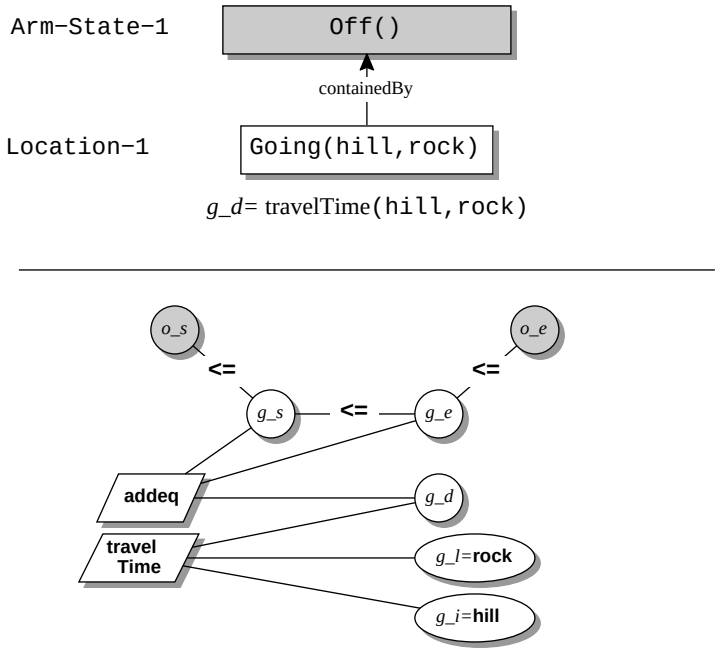

Figure 1. A plan fragment and its representation as a CSP.

3.2.1. *Compatibility Syntax*

The basic structure of compatibilities is similar to that of configuration rules, as they specify that certain intervals must exist in order for a given interval to appear in a valid plan. Configuration rules specify both constraints on how attributes may evolve, as well as cross-attribute constraints. For example, suppose that a rover travels from the `rock` location to the `hill` location. The model must ensure that the rover's location is constrained to be the location `hill` as soon as the traversal is complete. This can easily be turned into a constraint on any interval of the form $\texttt{holds}(\mathit{Location}, s, e, \texttt{Going}(\texttt{rock}, \texttt{hill}))$, by requiring that it immediately be followed by an interval of the form $\texttt{holds}(\mathit{Location}, s', e', \texttt{At}(\texttt{hill}))$. This is accomplished by asserting that $e = s'$. The limitations on cross-attribute restrictions are easily encoded in the same manner. Our previous example asserted that while the rover was travelling, the robotic arm has to be turned off; this example involves an interval on the *Location* attribute asserting that an interval must hold on the *ArmState* attribute.

Compatibilities can also be used to limit the set of intervals that is allowed to be in the plan, by using constraints on the parameters of the intervals. For example, suppose going from the `rock` location to the `hill` location takes 40 time units. This can be represented by a constraint

on any interval of the form $\texttt{holds}(\mathit{Location}, s, e, \texttt{Going}(\texttt{rock}, \texttt{hill}))$, simply by requiring that $e - s = 40$.

Finally, compatibilities can employ *guards* to limit the set of intervals that the compatibility applies to. For example, suppose the constraint that the robot arm must be turned off while the rover is travelling only applies when travelling to the location `rock`. We can use a guard that indicates that the compatibility applies to predicates of the form `Going(`*loc-1*`,hill)`.

Based on these observations, a compatibility has to specify the set of intervals to which it applies, the constraints on valid variable combinations, and the set of valid configurations. A simple compatibility is structured as follows:

Head Interval: $\texttt{holds}(a, t_1, t_2, I(x_1, \ldots, x_k))$
Guards: **list-of** $v_i \in G_i$
Parameter Constraints: **list-of** $C_j(Y_j)$
Disjunction of Configurations: **list-of**
  Configuration rule: **list-of**
    Configuration Interval: $\texttt{holds}(b, t_3, t_4, J_{kr}(z_1, \ldots, z_n))$
    Configuration Constraints: **list-of** $K_m(W_m)$

As previously described in §2.2, compatibilities can be understood as implications; abusing notation slightly, the semantics of compatibilities are:

$$(\texttt{holds}(\texttt{I}) \wedge G_1 ... \wedge G_i) \Rightarrow$$

$$(C_1 ... \wedge C_j \wedge (O_1 ... \vee O_k))$$

where

$$O_s \equiv (\texttt{holds}(\texttt{J}_{\texttt{s1}}) \wedge K_{s1_1} ... \wedge K_{s1_m})$$

$$... \wedge (\texttt{holds}(\texttt{J}_{\texttt{sr}}) \wedge K_{sr_1} ... \wedge K_{sr_n})$$

Again, we utilize variables and constraints to make the representation more compact. Each of the guards specifies a domain G_i for a *head interval variable*, i.e. a variable in the set $\{a, t_1, t_2, x_1 \ldots x_k\}$. The head and the guards specify the set of grounded intervals to which the compatibility applies. Each C_j is a constraint on the head interval variables, 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 with respect to I by the general constraints K_m, each of which has a scope W_m that combines variables from the parameters of I and the parameters of

the interval for J, that is, the set $\{a, b, t_1, t_2, t_3, t_4, x_1 \dots x_k, z_1 \dots z_n\}$. We will formally define the semantics of compatibilities in §3.2.2.

There are two reasons to separate the specification of the constraints restricted to the head interval I from those relating the parameters of I and J_{kr}. The first is that it is a natural separation. The parameter constraints C_j are limited to the parameters of the predicate atom; these constraints restrict the evolution of the attribute instance. The remaining constraints K_m enforce how intervals on different attributes are mutually constrained. The second reason is more technical. It is possible that the conjunction within a configuration rule is empty, indicating that interval I can appear in any configuration. In such cases, using configuration constraints to constrain the parameters in I would require adding unnecessary configuration rule entries.

3.2.2. *Semantics 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 compatibilities and configuration rules are equally expressive. However, the use of compatibilities will invariably lead to an exponentially smaller encoding of the domain constraints.

The semantics of a compatibility are as follows:

DEFINITION 3.1. *Given a planning domain $\mathcal{D}$ and attribute $A \in \mathcal{D}$. If A takes on the value $I(X)$, and all of the guard constraints $v_i \in G_i$ of a compatibility M are satisfied, then the following must hold:*

- *All of the parameter constraints $C_i(Y_i) \in M$ must be satisfied.*
- *There is a configuration rule $O_k \in M$ such that, for each configuration interval $J_{kr}(A) \in O_k$, there must exist a corresponding interval $J'(A')$ in the plan, such that:*
  - *The predicates of $J_{kr}(A)$ and $J'(A')$ match.*
  - *The configuration constraints $K_{kr}(W_{kr})$ of the configuration interval hold on the domains of the variables in the set A'.*

It should be noted that multiple compatibilities may be applicable to a given interval; should that be true, all the compatibilities must hold simultaneously.

Judicious use of guards allow us to create compatibilities that enforce conditional constraints and configuration rules. We will see an example of this in §3.2.5. 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 a number of restrictions on `Going`: The rover must be at the location it begins the `Going` action, and must end up at the location it terminates the `Going` action. The travel time depends on the departure point and the destination; let us assume this time is specified by a function named `travelTime`. Lastly, while the rover is traveling, the arm must be `Off`.

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

Head: `holds`(*Location-1*,s_g, e_g,`Going`(x, y))
Parameter Constraints: $s_g + travelTime(x, y) = e_g$
Disjunction:
  Configuration Rule:
    Configuration Interval: `met_by holds`(*Location-2*,s_{a1}, e_{a1},`At`(a)),
    $a = x$, *Location-1 = Location-2*
    Configuration Interval: `meets holds`(*Location-2*,s_{a2}, e_{a2},`At`(b)),
    $b = y$, *Location-1 = Location-2*
    Configuration Interval: `contains holds`(*ArmState*,s_o, e_o,`Off`)

Figure 2 shows this pictorially.

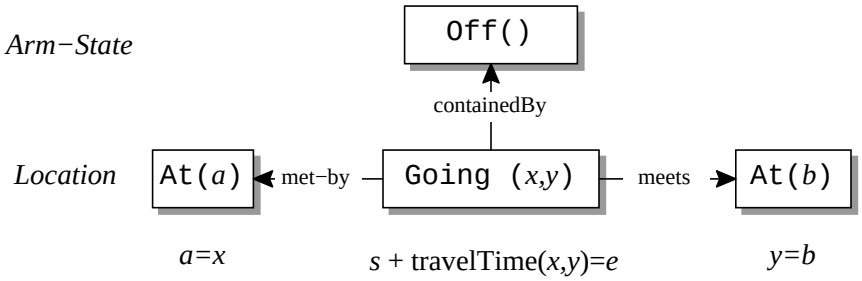

Figure 2. A graphical diagram of a simple compatibility for `Going`(x,y). The parameter constraint is shown beneath the `Going` interval. We omit the constraints on the *Location* parameters for simplicity.

3.2.4. *Example: A Disjunctive Compatibility*
Let us now consider an example of a compatibility specifying optional configurations for an interval. 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 or followed by `At` or `Turning`. This requires deciding whether or not to turn before traveling to the

next location. The following compatibility shows the set of possible configurations for `Going`, indicating that this action can be preceeded by `At` or `Turning`, and similarly can be followed by `At` or `Turning`. Note, however, that the robot arm must always be off; this part of the compatibility is repeated in every configuration.

Head: `holds`(*Location-1*,s_g, e_g,`Going`(x, y))
Parameter Constraints: $s_g + travelTime(x, y) = e_g$
Disjunction:
  Configuration Rule:
    Configuration Interval: `contains holds`(ArmState,s_o, e_o,`Off`)
    Configuration Interval: `met_by holds`(*Location-2*,s_a, e_a,`At`(q)),
    Location-1 = Location-2
  Configuration Rule:
    Configuration Interval: `contains holds`(ArmState,s_o, e_o,`Off`)
    Configuration Interval: `met_by holds`(*Location-2*,s_a, e_a,`Turning`(q)),
    Location-1 = Location-2
  Configuration Rule:
    Configuration Interval: `contains holds`(ArmState,s_o, e_o,`Off`)
    Configuration Interval: `meets holds`(*Location-2*,s_a, e_a,`At`(q)),
    Location-1 = Location-2
  Configuration Rule:
    Configuration Interval: `contains holds`(ArmState,s_o, e_o,`Off`)
    Configuration Interval: `meets holds`(*Location-2*,s_a, e_a,`Turning`(q)),
    Location-1 = Location-2

3.2.5. *Example: Using Guards for Disjunctive Compatibilities*
In the previous example, we retained the explicit disjunction syntax from configuration rules. However, we can represent disjunctions by using explicit variables and guards. We introduce new parameters to the predicates whose domain is the set of disjunctive choices [2]. A set of guarded compatibilities on the same interval can specify the disjunction of plan configurations. The `Going` predicate is 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 compatibility can possibly be satisfied at a time, only one configuration can apply. Furthermore, we eliminate the repetitive specification of the limitations on the `Off` state by specifying the compatibilty with no guard, so it applies to every `Going` interval. The compatibilities are shown here below, and Figure 3 shows the structure

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

of the compatibilities graphically. Note that the compatibility syntax is simplified, because we have omitted the disjunctions and replaced them with variables.

Head: `holds`(*Location-1*,s_g, e_g,`Going`(x, y, p, f))
Parameter Constraints: $s_g + travelTime(x, y) = e_g$
Configuration Rule:
Configuration Interval: `contains` `holds`(ArmState,s_o, e_o,`Off`)

Head: `holds`(*Location-1*,s_g, e_g,`Going`(x, y, p, f))
Guards: $p =$`at-bef-go`
Configuration Rule:
Configuration Interval: `met_by` `holds`(*Location-2*,s_a, e_a,`At`(q)),
$q = x$, *Location-1 = Location-2*

Head: `holds`(*Location-1*,s_g, e_g,`Going`(x, y, p, f))
Guards: $p =$`turn-bef-go`
Configuration Rule:
Configuration Interval: `met_by` `holds`(*Location-2*,s_t, e_t,`Turning`(q)),
$q = x$, *Location-1 = Location-2*

Head: `holds`(*Location-1*,s_g, e_g,`Going`(x, y, p, f))
Guard: $f =$`at-aft-go`
Configuration Rule:
Configuration Interval: `meets` `holds`(*Location-2*,s_a, e_a,`At`(q)),
$q = y$, *Location-1 = Location-2*

Head: `holds`(*Location-1*,s_g, e_g,`Going`(x, y, p, f))
Guards: $f =$`turn-aft-go`
Configuration Rule:
Configuration Interval: `meets` `holds`(*Location-2*,s_t, e_t,`Turning`(q)),
$q = x$, *Location-1 = Location-2*

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. The variables in the guards can be constrained in the same manner as all other variables, and we can now propagate to and from the variables in the guards, possibly limiting the legal configurations by reducing the legal variable domains of these variables. Secondly, the application and semantics of compatibilities are simplified.

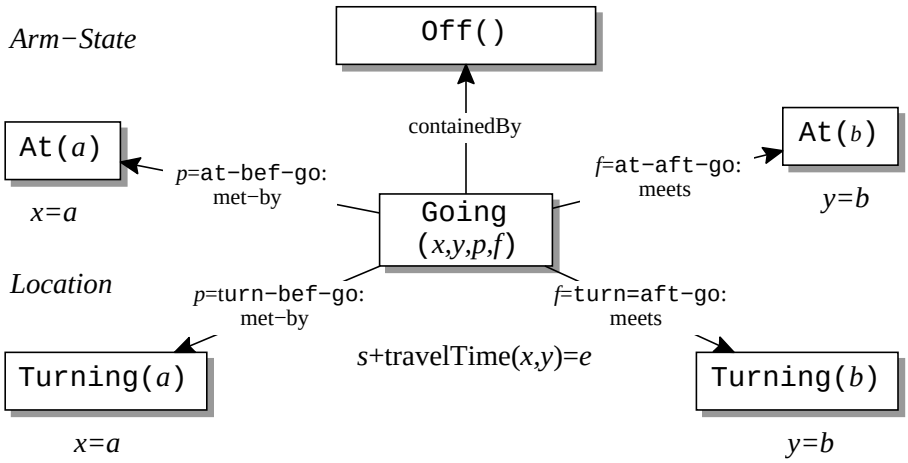

Figure 3. A graphical diagram of a disjunctive compatibility for `Going`(x, y). The constraints are shown beneath the `Going` interval. The conditional configuration rules are shown by placing the guards and temporal relations on the arcs linking the intervals that exist in the plan if the guard is satisfied. Attribute equivalences are omitted.

3.3. Building Plans

Recall that in our constraint-based representation, actions and states are described using intervals where timepoints, parameters and other information is represented by constrained variables. We now extend the definition of candidate plans given in §2.3: a *candidate plan* consists of a mapping of attributes to sequences of intervals, a set of non-sequenced intervals, and a set of additional constraints on variables in the given intervals. The notions of valid plan and plan extension are extended in an analogous manner. This is a very expressive approach to specifying planning problems, as it allows us to compactly specify sets of possible interval instantiations.

We now turn our attention to the basis of the planning process, by describing how such candidate plans can be modified to turn them into valid plans. First, we show how to modify plans in order to search the space of possible plans. Next, we extend the notion of complete plans to *sufficient plans*. Finally, we discuss how compatibilities are satisfied in the planning process.

3.3.1. *Modifying Plans*

We now define the operations that modify plans, and can be used to search for a valid complete plan. The first group of modifications either adds new decisions to the plan or reduces the number of options

remaining for existing decisions, and thus potentially reduce the set of valid plan extensions, and so are called *restrictions*:

- Satisfying a compatibility interval requirement. First, the existence of the interval must be satisfied. This can be accomplished in two different ways:
  - A *new* 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 temporal constraints enforcing the total ordering of distinct intervals on the attribute will be violated.
  - Constraints are added to force an *existing* interval to satisfy the compatibility element in question. Note that this requires that the existing interval matches the interval specified and that the constraints added do not cause inconsistency.

  The required parameter constraints and configuration constraints linking the two intervals are then added to the plan.
- An unsequenced interval can be inserted on an attribute. The choices are defined by the domain of the attribute variable, and the intervals already in place on the compatible attributes.
- 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 inverses of these operations are *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.

3.3.2. *Sufficient Plans*

The traditional definition of a complete valid plan requires that all of the constraints are satisfied, all intervals have been sequenced, and that the plan has been fully specified to the end of time. However, this is an unnecessarily restrictive definition. In many applications of planning, the entity executing a plan can "fill in" unspecified parts of the plan. A common example of this is the plan horizon, mentioned in §3.1. When

building a plan for execution, the goal is typically to build the plan up to a certain timepoint, the plan horizon. This allows the planner to ignore activities that fall outside the horizon. As another example, the Remote Agent (Jónsson et al., 2000), was capable of instantiating unbound temporal variables during execution. Thus, the planner did not fully ground every temporal variable. To allow for such flexibility, we will refer to plans that satisfy a given set of completeness requirements as *sufficient plans*.

We now elaborate on the definition of the *extension* of a candidate plan. Given a candidate plan P_C, an extension of P_C is a plan P such that:

- There is a mapping from the intervals in P_C to a subset of the intervals in P that maps each interval I in P_C to an interval J in P that has the same predicate and such that the domain of a variable in J is a subset of the domain of the corresponding variable in I.

- P satisfies all the constraints given in P_C.

Clearly, the set of plan extensions 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. We can now refine the solution of a planning problem instance to be a *sufficient and valid extension* of the initial plan candidate P_C.

Three things distinguish candidate plans from solutions. One is that there may be unbound variables, which represent choices in the exact specification of intervals. Another is that there may be unsequenced intervals in the candidate plan, which have to be scheduled onto the appropriate attribute. The third and last is that some applicable compatibilities may not yet satisfied, in the sense that there are intervals that are required to be in the plan by some compatibility, but are not yet sequenced. Recall that these distinctions only apply to variables and intervals that the sufficiency condition does not otherwise exclude.

For variables, the choices to be made are straightforward. If multiple values remain in the domain of a variable, one must be selected as the value assigned to the variable. Obviously, the value chosen must satisfy all constraints in the plan. For unscheduled intervals, the set of possible decisions is also clear, with one slight wrinkle. In addition to being placed before or after other intervals, unsequenced intervals can be equated with intervals that have the same predicate. This corresponds to satisfying a goal in traditional planning by using an existing state in the plan. The variables of the merged interval are equated,

and must satisfy all configuration and parameter constraints related to both intervals. It remains only to describe how the compatibilities are satisfied.

3.3.3. *Satisfying Compatibilities*

Handling unsatisfied compatibilities is the trickiest part of building plans. There are two main reasons for this. The first is that satisfying a compatibility may involve multiple decisions. In the CAIP framework, this is handled by conceptually splitting the compatibility enforcement into a set of separate decisions. The second issue is that the decisions enforcing the satisfaction of a compatibility must continue to hold as planning proceeds.

Before we describe our approach to handling unsatisfied compatibilities, let us look at the analogous issue in classical POCL planning. In concurrent plans, the preconditions required by any given action must be achieved by a set of earlier actions, such that no other actions can delete a preconditions before the action requiring it begins. The second issue here above, then, becomes a question of how to ensure that achieved preconditions remain enforced until the action is ready to execute. In POCL planning, this is done by explicitly adding the safety conditions and causal links as part of the decision-making process (McAllester and Rosenblitt, 1991).

Turning our attention back to the CAIP framework, recall that each compatibility defines a set of parameter constraints and a set of configuration rules. Since the framework is based on an underlying constraint network, enforcing the parameter constraints is straightforward; they are simply added to the constraint network when the interval is created. When the compatibility is no longer applicable, the constraints are removed. The handling of the configuration rules is somewhat more involved, but also builds on the constraint-based representation used in CAIP.

Let us consider enforcing a configuration rule on an interval I that is part of a plan. Suppose the configuration rule on I requires the existence of an interval J, along with some constraints K_i. The constraints K_i are created along with J, and are added to the constraint network. Consequently, any future decisions made during planning must satisfy these constraints along with the other constraints in the plan. The planner can satisfy the need for the interval J by either equating J with some existing interval, or by sequencing J between a pair of ordered intervals. In both cases, new constraints are added to the plan, and again, any future decisions made during planning must satisfy these constraints. This posting of constraints is a natural extension of the notion of causal links and ordering constraints in POCL planning.

3.4. Planning Framework Correctness and Completeness

Before we can formalize our correctness criterion, we note that it is not possible to build an *arbitrary* sufficient and valid extension of a given candidate plan, using only the operations we defined here above. Consider a domain with one attribute I, for which there are two intervals A and B. Now consider a candidate plan with one fully grounded interval A. Assume further that the sufficiency criteria eliminates all intervals and variables that would otherwise be required by the compatibilities for A. This candidate plan is a sufficient and valid plan. However, a plan where another interval, B, has been added after the original interval is also sufficient and valid, but cannot be generated only by adding intervals required by compatibilities and not excluded by the sufficiency criteria.

Our correctness and completeness proof must take this issue into account. This is done by requiring only that the plan modification operations be able to construct an "intermediate" extension that can be further modified by adding the necessary intervals and constraints to construct any sufficient valid extension of the initial plan.

THEOREM 3.1. *Let $\mathcal{D}$ be a domain, and let P_C be a finite length plan candidate. Let Q be any finite length sufficient and valid extension of P_C. Then P_C can be transformed into a sufficient and valid extension P of P_C by a sequence of plan modifications as described in §3.3.1, such that P can be transformed into Q by adding a (possibly empty) set of intervals and constraints to P.*

We will outline the formal proof here, by showing that there exists a sequence of plan modifications that transforms P_C into a valid and sufficient plan P, such that applicable compatibilities are satisfied in the same way as in Q, and necessary variable assignments and interval sequencing decisions are made the same way as in Q. We initialize $P = P_C$ and apply the following operations repeatedly:

- if some applicable compatibility is not enforced for an interval
  - find a set of intervals in Q that satisfy the compatibility in question
  - if matching intervals already exist in P, add any constraints in Q that are not in P
  - otherwise, add a new interval to P, sequencing it in the same way it appears on attributes in Q
- if an unsequenced interval in P is sequenced in Q, then sequence it in the same way as in Q

– if an interval in P has a variable whose value is a superset of the assignment in Q, make the variable assignment

Halt this process when P is valid and sufficient.

It is relatively straightforward to see that the process halts with a valid and sufficient plan P. First, we note that the set of compatibilities that must be satisfied in Q is finite, since Q has a finite number of intervals, and each interval only gives rise to a finite set of applicable compatibilities. Second, we note that during the process of transforming P_C into P, the set of compatibilities is monotonically increasing, while being bounded by the set of compatibilities applicable to Q. This is because the only intervals added to P are those that appear in Q. Finally, we note that the set of variables to be bound, and the set of unsequenced intervals to be sequenced, are also finite. Since each step in the process addresses a compatibility, an unbound variable, or an unsequenced interval, only a finite number of steps can be taken. The validity and sufficiency of Q guarantees that the process can halt with a valid and sufficient plan.

4. 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. In this section, we focus on constraint reasoning, deferring discussion of the application of constraint satisfaction heuristics and planning to later work.

There are many constraint reasoning techniques that can be used to make the planning effort more effective. The only requirement we impose is that the constraint reasoning methods must preserve the set of valid plan extensions. Any constraint reasoning technique that only performs sound reasoning, i.e. only eliminates values or value combinations that provably cannot participate in a solution, preserves the set of valid plans. Examples of sound reasoning include temporal constraint propagation (Dechter et al., 1991), arc consistency maintenance, higher-level consistency enforcement such as k-consistency, and the correct procedure application as described in (Jónsson and Frank, 2000). In the following sections we discuss some aspects of constraint reasoning when planning with the CAIP framework.

4.1. Domain dependent constraints

When modeling real systems, often constraints will not be expressed as relations, but as functions. In many cases, complex machinery will be required to compute these functions. One way to handle these constraints is to use a *procedural constraints* framework (Jónsson, 1997; Jónsson and Frank, 2000). In this framework, each constraint is embodied as a procedure. This allows arbitrary functions to be encapsulated as relations among the variables of the function. An additional benefit of this is an efficient and compact representation of the constraints, as each procedure can take advantage of the most efficient 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.

4.2. Reasoning about Collections of Constraints

Several methods exist to handle a homogeneous set of constraints using efficient techniques. For examples, cliques of equivalence constraints can be enforced much more efficiently than enforcing them piecemeal in a general reasoning framework; this approach is used by the ZENO system (Penberthy, 1993). The set of all simple temporal constraints can also be made arc-consistency efficiently by using the algorithms described in (Dechter et al., 1991). However, this is not the only way to handle temporal constraints; they can also be treated as arithmetic constraints and handled by Gaussian elimination, as is done by ZENO (Penberthy, 1993).

4.3. Constraints on Attributes

Attributes turn out to be a useful framework for discussing constraint reasoning techniques. Attributes can be viewed as unit-capacity resources, and therefore techniques such as edge-finding (Nuijten, 1994) can be employed among the intervals on a single attribute instance. However, since intervals may be equated, edge-finding would have to account for this possibility when computing its bounds. Similarly, edge-finding would only be allowed to reason about intervals that are known to take place on a particular attribute instance. In order to make full use of edge finding, it would have to be fully incremental to reduce the cost of recomputation.

Some domains employ attributes to represent multi-capacity resources by employing a combination of parameter choices and arith-

metic constraints. For example, we can model a renewable resource using two intervals: `Has` takes one parameter a and asserts that a resource has amount a resource remaining. `Change` takes 3 parameters: i, the initial amount of the resource, d the change in the resource, and f the final amount of the resource. We then use the following compatibilities to model the evolution of the resource due to `Change`:

Head: `holds`(*Resource-Amount-1*,s_g, e_g,`Change`(i, d, f))
Parameter Constraints: $f = i - d$
Configuration Rule:
`met_by holds`(*Resource-Amount-2*,s_a, e_a,`Has`(x)),
$i = x$,*Resource-Amount-2 = Resource-Amount-1*
`meets holds`(*Resource-Amount-3*,s_b, e_b,`Has`(y)),
$f = y$,*Resource-Amount-3 = Resource-Amount-1*

Head: `holds`(*Resource-Amount-1*,s_g, e_g,`Has`(x))
Parameter Constraints: $f = i - d$
Configuration Rule:
`met_by holds`(*Resource-Amount-2*,s_a, e_a,`Change`(a, b, c)),
$c = x$,*Resource-Amount-2 = Resource-Amount-1*
`meets holds`(*Resource-Amount-3*,s_b, e_b,`Change`(d, e, f)),
$d = x$,*Resource-Amount-3 = Resource-Amount-1*

Currently, these compatibilities may be mixed with others that impact the same interval. Isolating these compatibilities would permit the use of techniques such as the balance constraint (Laborie, 2001) to both determine the state of resource consumption, and add other constraints to the plan to ensure that adequate resources are available.

Finally, constraint reasoning can be used to efficiently detect temporal inconsistencies related to inserting unsequenced intervals into the plan, by exploiting the structure of attributes. Consider the following example, as shown in Figure 4.3. Let `P` be a free (unsequenced) interval that was added to satisfy a "before" compatibility for interval `Q`. Let `S` be another interval, added to satisfy a "before" compatibility for interval `R`, which is temporally ordered after `Q`. Inserting `P` on or after `S` results in a directed cycle of temporal constraints that makes the candidate plan invalid.

While this constraint violation would be caught by the Simple Temporal Network (STN) algorithm of (Dechter et al., 1991), it may be found faster by using a special-purpose method for detecting cycles between attributes. There are some subtleties involving the possible combinations of temporal constraints that can lead to such cycles. However, the complexity of these checks depends only on the number of intervals on the attributes, and is simpler than the cycle detection

done in full STN algorithms. Suppose there are m intervals on the same attribute as Q and further that each of these is related by compatibilities to at most j other intervals. Then the complexity[3] of checking for temporal cycles is $O(mj)$. In many useful cases, this is considerably smaller than the complexity of enforcing consistency on the full STN.

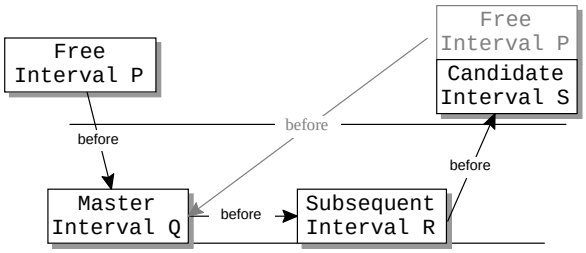

Figure 4. Identifying inconsistent orderings for intervals by analyzing pairs of attribute instances. Note that in this figure, a directed edge exists between the interval timepoints labeled with the temporal relation which induces that constraint.

4.4. No-Good Reasoning

No-good reasoning has proven to be a powerful method in reasoning about CSPs, and has been used previously in addressing planning problems (Do and Khambhampati, 2000). No-good reasoning may improve the performance of search algorithms designed to employ the CAIP representation. However, because dynamic constraint reasoning is required to support CAIP, variables (and values, depending on the representation) can disappear. No-goods mentioning variables that have been eliminated no longer apply, and can be removed from the no-good store. While performance improvement is possible using no-goods, the fact that no-goods must be eliminated means that care must be taken to avoid inefficiency due to no-good maintenance.

5. EUROPA: A CAIP Implementation

We have implemented a system called EUROPA using the CAIP framework. In this section, we describe some of the design features and implementation decisions embodied in EUROPA. We will focus on two

[3] The number of intervals m is actually the number of total intervals, not the number of interval equivalence classes formed by equating pairs of intervals. The number of compatibility-based interval relations can be inferred from the model. Careful management of data structures may reduce m and j by accounting for equivalences.

aspects of EUROPA; constraint-reasoning techniques employed in the implementation, and how compatibilities are enforced during planning.

EUROPA has been designed and implemented as a software module that maintains a representation of a candidate plan. The module provides interfaces that allow a client to modify the plan, and query the plan concerning its validity, consistency status, open decisions, and so on. The EUROPA systems has been applied to several domains, including multiple satellite scheduling, planetary rover operations, automated flight planning, and DS-1.

5.1. Constraint and Attribute Reasoning

The EUROPA system uses several different constraint reasoning methods. The procedural constraint reasoning framework of (Jónsson and Frank, 2000), the simple temporal network propagation algorithms of (Dechter et al., 1991) and maintenance and propagation of equivalence classes form the principal reasoning mechanism. Constraints in the procedural framework enforce either arc-consistency or, in the case of arithmetic constraints, bounds consistency. The temporal relations are simplified and made explicit in EUROPA compatibilities by limiting the configuration constraints $K_m(W_m)$ to one of Allen's interval algebra(Allen and Koomen, 1983), with metric interval distance bounds. The reason for this is that the resulting constraints form an STN, which can be efficiently reasoned about, as mentioned previously. Configuration constraints may also include equivalence constraints between parameters of the head interval and parameters of the configuration interval; the syntax supports this by allowing the same variable name in the head interval and configuration interval specifications. Finally, configuration constraints include unary constraints on the domains of the configuration intervals. The syntax of a EUROPA compatibility is

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

where $z_1, \ldots, z_n$ is either a variable in the set $x_1, \ldots, x_k$ or a domain specification, and where τ is a temporal constraint[4].

[4] The EUROPA syntax permits disjunctions to be expressed using an AND-OR tree, where the OR is associated with a parameter variable, in addition to the guarded compatibility structure introduced in this paper.

We treat the duration variables as "non-temporal" variables, and link the temporal constraints to the rest of the constraint network via the interval duration constraint $e - s = d$. We also post constraints that "equate" temporal variables to non-temporal variables; these constraints propagate the bounding interval to the STN. Since arbitrary constraints may lead to disjoint intervals, we use this trick to "protect" the STN from the disjunctions.

EUROPA does not explicity represent the domain of possibilities for interval insertion decisions. Rather, when an interval is to be inserted, the domain is constructed by consulting the attribute. Temporal cycle checking techniques like those described in §4.3 are used as a "forward checking" phase when retrieving the candidates for interval insertion. An alternative way of handling interval insertions is discussed in some detail in (Frank et al., 2000).

5.2. Compatibility Enforcement

In the EUROPA framework, the decisions associated with satisfying the compatibilities for an interval are mapped into decisions concerning how to sequence unsequenced intervals. This is done by maintaining the *plan invariant* that every sequenced interval is supported by the existence of intervals that satisfy applicable constraints.

Let P be a candidate plan, and let $P(X)$ be some sequenced interval that matches the antecedent of a compatibility. Let $Q_i(Y_i)$ be the set of intervals specified by that compatibility, along with temporal constraints $\tau_i(P(X), Q_i(Y_i))$, relating $P(X)$ and $Q_i(Y_i)$. Let C be the constraints on variables in X, Y_i, also given by the compatibility. To enforce the EUROPA plan invariant, all intervals $Q_i(Y_i)$ and constraints C and τ_i are added to the candidate plan when $P(X)$ is added to an attribute instance. Similarly, when $P(X)$ is removed from an attribute instance, all of its required intervals $Q_i(Y_i)$ are removed from the candidate plan, and the constraints C and τ_i are retracted. By noting whether the change is a restriction (further refining the plan by adding constraints and intervals, or by assigning values to variables), or a relaxation, the modifications necessary can be done incrementally:

– 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 in an attribute sequence, leading to applicability of a compatibility. Another example is that a variable's domain may now match a compatibility guard, leading to the applicability of a configuration rule. If, however, no new compatibilities apply, then no intervals or constraints are added to the candidate plan.

– 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.

For an example of how this is done, let us return to the complex rover model. Suppose we have a plan with `holds`(*Location-1*, s, e, `Going`(x, y, p, f)), with *Location-1* = `Location-Instance-1`, and the assignment p =`turn-bef-go` is made. Since a variable assignment was made, we only check the guards relevant to variable p. The guard p =`turn-bef-go` is satisfied, and therefore a non-sequenced interval `holds`(*Location-2*, s', e', `Turning(a)` is added to the plan, along with the constraints $a = x$,*Location-1* = *Location-2*, 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 5.

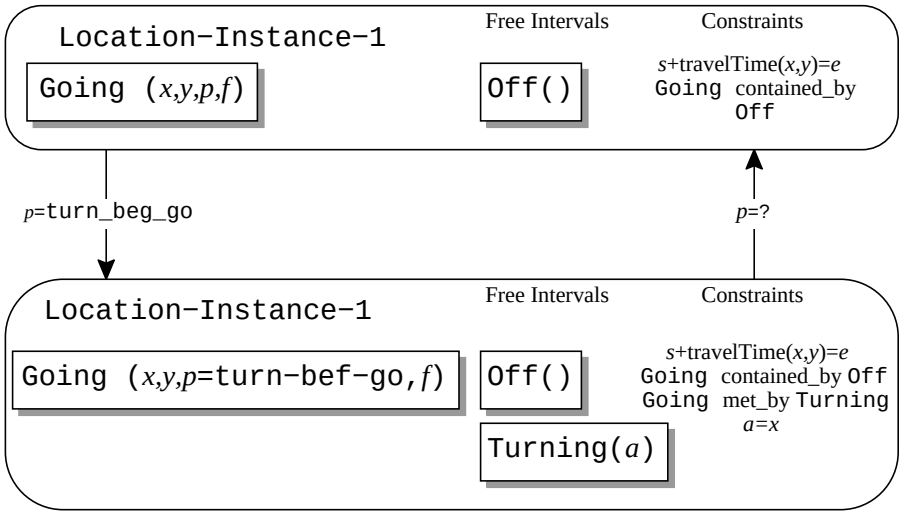

Figure 5. Illustration of the plan invariant at work. When the assignment p =turn-bef-go is made, free (unsequenced) intervals and constraints are added to the plan. When it is retracted, the configuration guard no longer holds, and the `Turning` interval and its constraints are removed from the plan.

Now let us consider a slightly different example, where an interval `holds(Location-instance-1,` s`,` e`,` `Going`(i, j, k, l)`)` has been added to satisfy a compatibility, such that it can be equated with an existing interval `holds(`*Location-instance-1*, s, e, `Going`(x, y, p, f)`)`. Suppose also that we already have made the assignment p =`turn-bef-go`.

When we equate the new interval with the existing interval, no additional compatibilities become applicable, so no new intervals are added to the plan, as shown in Figure 6. Similarly, if we remove the interval, no existing intervals are removed from the plan.

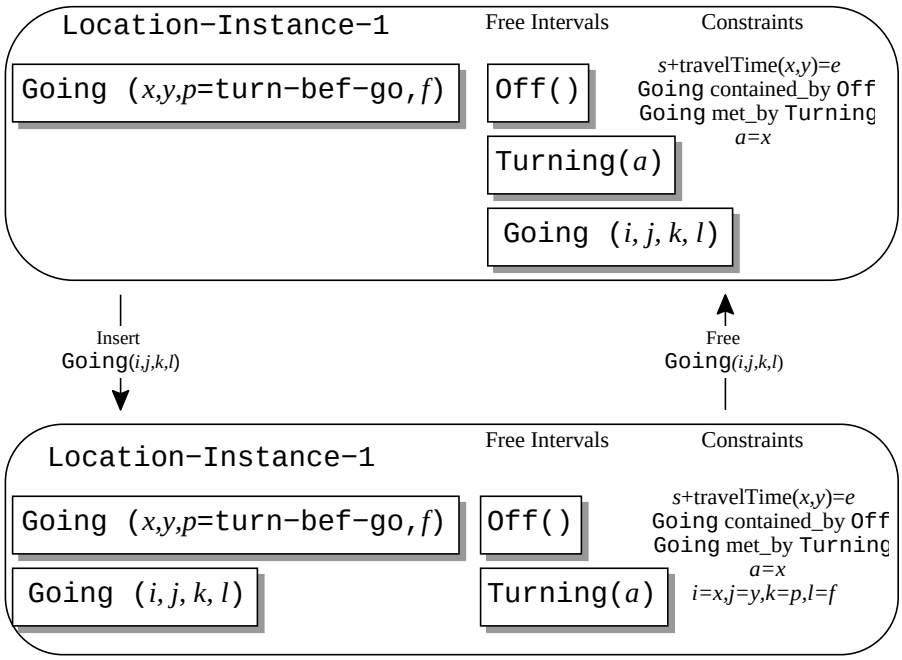

Figure 6. Illustration of the plan invariant at work. In this case, the plan invariant does not change the plan when the interval `holds(Location-instance-1,` s`,` e`, Going(`i, j, k, l`))` is added to or removed from the plan. The parameter equivalences are added or removed by the plan modification itself, not the plan invariant.

The approach taken to compatibility enforcement in EUROPA allows a form of least-commitment planning (Joslin, 1996) to be employed by planners; it allows parameters of intervals to be assigned before the intervals are sequenced. This comes at a cost; non-sequenced intervals must be maintained and managed, even if they are outside the horizon or end up being equated with other intervals. Enforcing the existence of Y can be done in different ways. In temporal POCL frameworks, the need for Y is handled by generating a decision that determines which of existing Y's in the plan are to be used, or whether a new interval is to be generated. The appropriate causal links are then inserted. This is similar to the approach used in EUROPA.

6. 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.

DEVISER (Vere, 1983) is a POCL planner that handles time [5]. 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 addition and deletion effects 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, called Temporally Quantified Assertions (TQAs). 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.

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 an 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; preconditions affected by the action are considered undefined during the action. 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

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

synchronizations between actions can be expressed. TGP also does not support attributes or resources.

The AIPS 2002 Planning Competition featured PDDL 2.1 (Fox and Long, 2002), a version of PDDL which includes facilities for representing temporal planning domains. PDDL 2.1 permits the spefication of planning problems in which actions take time, and allows the specification of "temporally annotated conditions and effects". Unlike the CAIP framework, conditions and effects can only be asserted at the beginning of an action, the end of an action, or during the action. Effects during the action are invariants over the action's duration. Actions are permitted to have conditional effects.

7. Conclusions and Future Work

We have presented constraint-based attribute and interval planning (CAIP), a planning framework that supports features common to real planning problems. CAIP provides primitives that support 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 has already been implemented in three different systems, including one that successfully controlled a spacecraft (Jónsson et al., 2000). The latest implementation, the EUROPA system, will further extend the application of this technology to real-world planning domains, as well as further expand the capabilities of the CAIP paradigm. Among these future challenges are real-time planning systems that will serve as smart execution systems, and mixed-initiative planning tools for helping human users build and verify complex plans.

The framework leaves considerable flexibility in the choice of constraint reasoning techniques to employ when building planners. For example, a wide variety of 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.

8. Acknowledgments

The authors would like to thank the anonymous reviewers for their comments. This work was supported by NASA Ames Research Center and the NASA Intelligent Systems Program.

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)).

Fox, M. and D. Long: 2002, 'PDDL 2.1: An Extension to PDDL for Expressing Temporal Planning Domains'. Technical Report 2/02, University of Durham Computer Science Department.

Frank, J., A. K. Jónsson, and P. H. Morris: 2000, 'On Reformulating Planning as Dynamic Constraint Satisfaction'. In: B. Choueiry and T. Walsh (eds.): *4th International Symposium on Abstraction, Reformulation and Approximation*.

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. and J. Frank: 2000, 'A Framework for Dynamic Constraint Reasoning Using Procedural Constraints'. In: *Euopean Conference on Artificial Intelligence*.

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.

Kautz, H. and B. Selman: 1996, 'Pushing the Envelope: Planning, Propositional Logic, and Stochastic Search'. In: *Proceedings of the 13th National Conference on Artificial Intelligence*. pp. 1194–1200.

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.

McAllester, D. and D. Rosenblitt: 1991, 'Systematic Non-linear Planning'. In: *Proceedings of the Ninth National Conference on Artificial Intelligence*.

McDermott, D.: 2000, 'The 1998 AI Planning Systems Competition'. *AI Magazine* **21**(2).

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.