# Identifying executable plans

**Tania Bedrax-Weiss*** **Jeremy D. Frank**
**Ari K. Jónsson†** **Conor McGann***
NASA Ames Research Center, MS 269-2
Moffett Field, CA 94035-1000,
{tania,frank,jonsson,cmcgann}@email.arc.nasa.gov

**Abstract**

Generating plans for execution imposes a different set of requirements on the planning process than those imposed by planning alone. Fully grounded plans will frequently become inconsistent when executed in dynamic environments. *Intelligent execution* permits making decisions when the most up-to-date information is available, ensuring fewer failures. Planning must acknowledge the capabilities of the execution system, both to ensure robust execution in the face of uncertainty, and to relieve the planner of the burden of making premature commitments. To do so, we formalize the notion of *executable plans* in a planning and execution system through the use of a declarative *Plan Identification Function* (PIF). PIFs guide the planner in making only decisions that the execution system cannot handle and also determine when to stop planning. We describe the implementation of PIFsfor a specific temporal, constraint-based planner. One of the motivations for this particular implementation is to support multiple different plan identification functions within the same planner.

## Introduction

Planning has been extensively studied in Artifcial Intelligence, but this has too often been done without considering the execution of the plan. Most approaches are designed to generate fully specified grounded plans. Unfortunately, a fully grounded plan will often contain unfounded assumptions about the outcomes of actions, which cause execution failures in unpredictable environments. Additionally, it is inefficient for the planning process to commit to decisions in advance that are likely to be invalidated during execution. One way to avoid these problems is to have an *intelligent execution system* that is able to "fill in the blanks" given a plan that is not fully grounded. Intelligent execution may range in complexity from simple computations to a process resembling full-blown planning. This means that the level of commitment and information in a plan should depend on the characteristics of the execution environment.

Different execution systems have different capabilities when it comes to executing plans that may not be fully specified. One engine may only be able to handle very specific open decisions, such as which of two identical cameras to use, while another may be able to assign execution times to actions whose order has been determined, but whose start times are left flexible. Where multiple execution engines are available, the same planner may be used to generate plans for all of them. Furthermore, variation in the available computational resources during execution can affect the amount of flexibility the plan execution system can handle. As a consequence, an important part of planning for execution is to allow the planning system to generate plans at the right level of abstraction and commitment given all relevant constraints on the executive in question.

In this paper, we outline a formalization that allows us to easily specify which plans are considered executable by a given execution system. For a given execution engine, a plan is *executable* if the engine can turn the plan into commands that then are sent to the hardware interface. If a given plan is not fully specified, then the execution engine must be able to make the necessary decisions during execution, so that the plan gets correctly executed. To formalize this notion, we will define the notion of Plan Identification Functions (PIFs). These functions are used to characterize execution engines by specifying which plans can be handled and which can not. Consequently, these functions give us the ability to specify the boundary between planning and execution in a flexible manner.

This technique has many advantages in building intelligent agents combing planning and execution systems. The PIFallows a formal description of the separation of duties between the plannner and the executive. This formal framework makes it easy to vary the boundary between planning and execution, and also makes it possible to use different planning and execution tradeoffs within the confines of a single application. As we shall show later in the paper, a declarative language can be used to easily specify the PIFs, which makes the separation of duties part of a completely declarative model of the agent.

### A Simple Example

To examine the issues involved in generating plans for execution, let us consider a simple spacecraft that can slew (i.e, turn to different orientations), take pictures, and download pictures to Earth.

A plan request for this spacecraft might consist of a set of picture requests, and then a request for downloading some

*QSS Group, Inc.
†Research Institute for Advanced Computer Science

or all of these pictures to Earth. The planning process would generate an "executable plan" that achieved those goals. An execution agent would then execute the plan by thrusting to rotate the spacecraft, activating camera components, and transmitting data.

A traditional approach to this problem is based on separating the planning from the execution at some fixed level of abstraction. For example, the planning process would generate slew actions, orientation maintenance actions, picture-taking actions, and download actions. The complete plan, at that level of abstraction, would then be executed by breaking each high-level action down into specific commands that together perform the action. The slew actions, for example, would be broken down into engine warmup, thruster firing, wait, opposite thruster firing, and then stabilization. This would be done according to the parameters specifying the details of each slew action.

A more sophisticated execution system might also be able to determine when to start activities that have some temporal flexibility. Rather than waiting until the next action start time, the execution system could determine that all earlier actions have been completed, that the action in question can start early in its feasibility window and then start that action at that time. Temporal flexibility is not the only type of flexibility an execution system may be able to handle. An execution system may be as simple or as complex as the problem requires.

The PIF provides a general mechanism for specifying the sets of decisions that an execution system can handle. In this case, the PIF would indicate that all temporal decisions can be left to the execution system. It would let the planner know which decisions to make and when it can stop by comparing the sets of outstanding decisions with the set of decisions that the execution system can handle.

It is possible to specify the domain model for the planner and the execution system in the same modeling language. Having the same language makes it easy to verify shared semantics and synchronize model changes. Information flows seemlessly between the execution system to the planner. Furthermore, changes in the planning model are then automatically propagated to the execution model. In addition, it provides flexibility in specifying the boundary between planning and execution systems.

For the remainder of this paper, we will assume a single shared model. However, the basic notion of plan identification can also be used in systems that have different languages, as long as the executability criteria can be translated from the execution plan language to that of the planner.

The rest of the paper is organized as follows. We first provide a formal definition of PIFs and characterize some useful properties of PIFs. We then describe an implementation of PIFs in a planning framework called EUROPA. We identify some important implementation details that arise when implementing PIFs in this framework. We then conclude and discuss several open issues.

## Plan Identification

We now turn our attention to formally defining the concepts related to general plan identification. We begin by outlining a general and expressive approach to planning, which supports arbitrary variables, quantitative temporal relations, arbitrary constraints, and expressive activity-state rules. This approach generalizes traditional STRIPS planning, but is significantly more expressive.

### Constraint-based planning

In order to address realistic problems, a planning paradigm must support actions and states with temporal extent, complex relations among action and state arguments, as well as complex model rules about conditions and effects of actions and states. In recent years, different approaches have been proposed for moving away from the classic STRIPS paradigm, and towards more realistic approaches that incorporate explicit representations of time and resources. These approaches fall into a broad category called *Constraint-Based Planning* (CBP) (SFJ00).

The basic idea behind CBP is to use variables to represent all aspects of states and actions, and to use constraints to enforce relations between those variables. The basic element in constraint-based planning is an interval. An *interval* is simply a predicate holding over a period of time. The start and end of the interval and the parameters of the predicate are described by variables. More formally, an *interval* is a tuple, $(p, X, s, e)$, where $p$ is a predicate name, $X$ is a vector of variables defining the arguments to the predicate, and $s$ and $e$ are temporal variables, defining the start and end of the interval.

A *planning domain* is defined by the set of interval types, and a set of configuration rules. A *configuration rule* is a generalization of the notion of preconditions and effects. It consists of a head and a set of consequences. The head of a configuration rule is a pattern for an action or a state. Each of the consequences specifies a state or action, along with a set of constraints. A configuration rule is *applicable* if its head matches some action or state in a plan. To satisfy the rule, all the consequences must also be in the plan, satisfying the associated constraints. The configuration rules are very expressive. Instead of specifying only state values before and after an action, they can specify arbitrary temporal relations between actions and states that must hold in a valid plan.

In our spacecraft example, we might have a configuration rule with a head specifying a `takePicture(x)` interval, and a consequence specifying that if $I$ is such an interval, the plan must also contain a `pointingAt(y)` interval, $J$, such that `x` $=$ `y`, the start of $J$ is at least 10 seconds before the start of $I$, and the end of $J$ is no earlier than the end of $I$.

### Partial plans and completions

In CBP, a *partial plan* consists of a set of intervals and a set of constraints among the variables representing those intervals.

A partial plan $P$ is *valid*, if for every applicable configuration rule, all the intervals and constraints required by those rules are in $P$. A partial plan $P$ is *instantiated*, if each variable has been given a single value. A partial plan $P$ is *con-*

*sistent* if none of the constraints in the plan are violated and *inconsistent* otherwise.

A *planning problem* is simply a partial plan. This notion generalizes the very restrictive STRIPS notion of only specifying an initial state and a set of goals. The notion of a planning problem as a partial plan allows specific actions as goals, supports the specification of maintenance goals, makes it easy to define exogonous events, and much more. Planners can modify plans in two ways. A *restriction* is defined as the binding of a variable or the addition of a constraint. A *relaxation* is defined as the unbinding of a variable or the removal of a constraint. An *extension* of a given partial plan, $P$, is a plan $Q$ such that each interval in $Q$ can be mapped to a compatible interval in $P$, and each constraint in $P$ is in $Q$. Thus, restricting a plan $P$ results in an extension $Q$, and relaxing a plan $Q'$ results in a plan $P'$ such that $Q'$ is an extension of $P'$.

A partial plan, $Q$ is *complete* if every interval is instantiated and the plan is valid. $Q$ is a *completion* of every relaxation of $Q$. We say that a problem instance $P$ has a *solution* if it has a consistent completion $Q$.

The strictest notion of solving a planning problem $P$ is to find a consistent completion of $P$. However, when planning for execution agents, a more general notion becomes more useful. A solution is any extension of $P$ that can be executed by the execution agent. Since the execution agent capabilities vary, the specification of what is an executable plan becomes part of the planning problem statement. We now turn our attention to formalizing that notion.

## Plan identification functions

Consider a partial plan encountered during the course of planning. We would like a declarative description of the set of plans that can be accepted for execution by the execution system. This is the notion of *plan identification functions* (PIFs). The basic idea is to have a mapping that indicates whether or not a partial plan is suitable for a given execution engine or not.

The original notion of a PIF appeared in (JMM+00), but it was slightly different. The original definition combined the notion of consistency with executability, and therefore used three return values, T, F and ?. Although not required, a common aspect of executable plans is that they are consistent. However, it is computationally expensive to determine whether a given partial plan is inconsistent or not. Consequently, it is useful to think of a "consistency identification function" that maps partial plans to T, F, or ?, where the T value indicates that the plan is consistent, F indicates it is inconsistent, and ? indicates that the consistency of the plan is not known. In the original definition of PIFs , the value T indicated that the plan was consistent and executable, the value F meant the plan was inconsistent, and ? was used to indicate that either the consistency or executability of the plan was not known yet.

In this paper we generalize the notion of PIFs . A PIF maps partial plans to the values Y and N. A return value of Y indicates a plan is executable and a return value of N indicates a plan is not executable. Keeping the definitions as general as possible, we do not impose any more restrictions on the evaluations of partial plans. For example, we do not require that a PIF must return N if the partial plan is inconsistent. The reason is that some plan execution systems may be able to work in the space of inconsistent plans, to repair a broken plan before execution.

## Characteristics of plan identification functions

The idea behind PIFs is to specify to a planner what is acceptable to a given execution agent. However, many classes of execution agents have elements in common, such as not being able to execute inconsistent plans. We now define some common characteristics that designers of PIFsmay wish to enforce. The first characteristic we define is consistency.

A PIF, $i$, enforces *consistency* if, for any partial plan $P$, such that $i(P) = \mathtt{y}$, $P$ has at least one consistent completion.

Another useful characterization is based how much work needs to be done by an execution engine to find a consistent completion in different circumstances. This is a particularly interesting question if uncertainty during execution is taken into account. The simplest case to handle is where the execution agent can make arbitrary choices to complete the given partial plan:

A PIF, $i$, enforces *solvability* if, for any partial plan $P$, such that $i(P) = \mathtt{y}$, all extensions of $P$ are complete and consistent.

Requiring solvability is often unnecessarily expensive for the planner and needlessly restrictive for the execution agent. A more relaxed notion is that a partial plan requires only a bounded amount of time to solve:

A PIF, $i$, enforces $O(f(n))$ *solvability* if, for any partial plan $P$ satisfying $i(P) = \mathtt{y}$, then in time $O(f(|P|))$ either a consistent completion of $P$ can be found or it can be shown that no consistent completion of $P$ exists.

As noted above, our formalization also covers execution agents that are capable of repairing inconsistent plans. For such agents, the PIF may return y for inconsistent plans. In that case, it is useful to characterize the amount of time required to repair the plan, in order to find a consistent completion of the original problem. If the original problem is $R$, we say that a PIF, $i$, enforces $O(f(n))$ *transformability* if, for any partial plan $P$ satisfying $i(P) = \mathtt{y}$, then in time $O(f(|P|))$ a series of restrictions and relaxations of $P$ resulting in a $Q$, a consistent completion $R$, can be found, or it can be shown that no such completion exists.

## Examples of Plan Identification Functions

The capabilities of the Remote Agent Executive were limited to handling temporal flexibility (MNP+98; JMM+00). Furthermore, the RA Executive had to guarantee a timely response, so the time required to deal with the flexibility had to be bounded. As a result, the PIF used with the Remote Agent Planner accepted only consistent and valid plans where all parameter variables had been assigned specific values, but tolerated unassigned temporal variables forming a dispatchable simple termporal network (JMM+00). The restriction that the resulting temporal network be dispatchable provided a linear bound on how much time it would take the execu-

tion engine to complete a given plan. In other words, the RA Planner used an $O(n)$ solvable PIF.

Recent techniques have extended the ability of execution systems to handle uncertainty in temporal quantities. In particular, (MMV01) presents an algorithm that can determine that a temporal network with uncertainty can be executed without failure, and (MM01) presents an algorithm for executing such networks in polynomial time. Execution agents based on these techniques would lead to PIFs are $O(f(|P|))$ solvable, where $f$ is a polynomial.

### From Plan Identification to Flaws

We have formally defined the PIF as a function from a plan and a model to an answer of either `Y` or `N`. This is sufficient for defining the capabilities of the execution agent and restrict the planner to returning only plans that are executable by that agent. However, when the planner gets the answer `N`, it is often difficult to determine what is keeping that plan from being executable. To address this issue, we look at a generalization of plan identification that can provide more information to the planner.

Consider an execution agent that is only capable of handling temporal flexibility. The corresponding PIF will return `N` for any plans that have unbound non-temporal variables. But more information is available; the PIF could indicate that the set of non-temporal variables $V$ are what stands in the way of the plan being executable. To support this extension, we extend the definition of a PIF to provide such an indication when possible.

To do this, we need to find a way to describe what prevents a plan from being executable. Let us first look at traditional planning, where the goal is to find a valid and consistent completion. For any plan that is not complete, not valid or inconsistent, it is possible to identify the cause. For example, if the plan is not valid, there must exist an applicable configuration rule such that one of its consequences is not enforced. This has led to the notion of *plan flaws*, or simply *flaws*.

Formally, a *flaw* in constraint-based planning is one of the following:

- A violated constraint
- An unbound variable
- An unenforced consequence of an applicable configuration rule

For a given plan $P$, let us denote the set of flaws by $\mathcal{F}(P)$.

Let us return to our example of an execution agent that only handles flexiblility in temporal variables. Let $P$ be a partial plan being considered for execution by this agent. If the set of flaws, $\mathcal{F}(P)$, has any flaws other than unbound temporal variables, the plan is not executable. But, instead of simply returning `N` to the planner, the execution agent function could indicate which flaws need to be addressed to make the plan executable. These are all of the non-temporal variable flaws. On the other hand, if the set of flaws for this plan, $\mathcal{F}(P)$, only has unbound temporal variable flaws, then the PIF returns `Y`. This leads us to viewing the PIF as a *flaw filter* that filters out any flaws that can be handled by the exeuction engine, and thus need not be handled by the planner. This relation is shown in Figure 1.

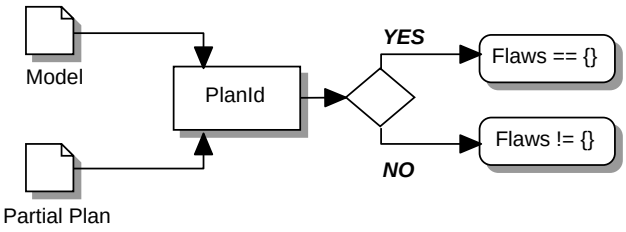


Figure 1: The Plan Identification Function as a Flaw Filter

We can now redefine a PIF as a function that maps a set of flaws to a subset of these flaws. In other words, if $i$ is a PIF, then $i(\mathcal{F}(P)) \subseteq \mathcal{F}(P)$, and a partial plan $P$ is *executable* if $i(\mathcal{F}(P)) = \emptyset$.

## An Implementation of Plan Identification

We now turn our attention to describing an example system that implements a general architcture for using PIFs. The system, called EUROPA (Extensible Universal Remote Operations Planning Architecture), is an instantiation of the the *Constraint-based Attribute and Interval Planning* framework (FJ03) (CAIP). In this section, we first give an overview of EUROPA, then describe how PIFs are implemented as flaw filters. Further details on EUROPA implementation can be found in (FJ03); in this section, we focus on those aspects that are most relevant to PIFs.

### EUROPA Overview

CAIP is an extension of CBP. Like the basic constraint-based planning paradigm, intervals represent actions with durations and states with temporal extent. The key addition in CAIP is the notion of an *attribute*. An attribute represents some system, subsystem or other aspect of the domain for which planning is being done. An attribute can only take on one value at a given time, so attributes enforce a mutual exclusion relation among intervals that are assigned to the same attribute. In addition, each interval must be placed on an attribute. This requirement enforces mutual exclusion among all intervals.

Consider the following simple model of the spacecraft domain. The first half of the model specifies the intervals that can appear on each attribute, and the second half specifies the configuration rules. We use the simple temporal relations of Allen's Algebra to specify constraints between the timepoints of required intervals. We assume that parameters of different intervals with the same variable name require the parameters to take on the same value.

EUROPA enforces configuration rules by means of the following *plan invariant*: whenever a plan modification results in a change to the set of plan completions, the intervals in the plan are updated. In the case of relaxations, some intervals that were part of the plan may no longer be justified, and if so, the intervals and all associated variables and constraints are removed. In the case of restrictions, new intervals, variables and constraints may be needed in the plan,

```
Attitude:{pointAt(object),
turnTo(object)}
Camera:{off(), ready(), takePic(object)}
Take-Picture(B) → met-by ready()
Take-Picture(B)  →  contained-by
pointAt(B)
ready() → met-by off()
pointAt(B) → met-by turnTo(B)
```

Figure 2: A simple model of the spacecraft domain.

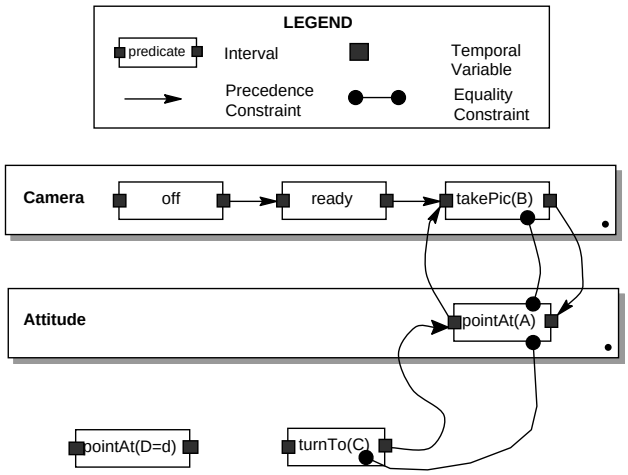


Figure 3: A simple partial plan for the model described in Figure 2

and if so, they are added. If new intervals are added, they remain as free intervals until the planner decides which attribute to sequence it in.

Figure 3 shows a plan fragment based on the simple model. The `Camera` attribute is initially turned off, then it is ready, and then it is taking a picture. While the `Camera` is taking a picture of the object, the `Attitude` is pointing at the object. Notice that there are two free intervals, one with predicate `turnTo` and one with predicate `pointAt`. The free interval `turnTo` was generated by the plan invariant, while the free interval `pointAt` was part of the initial problem instance.

Consider the interval `pointAt(`$A$`)` which is inserted on the `Attitude` attribute. In this case, the rule `pointAt(`$B$`)` → `met-by turnTo(`$B$`)` means that if a `pointAt(`$B$`)` interval exists in a plan, then a `turnTo(`$B$`)` must preceed the `pointAt(`$B$`)`. The presence of the `pointAt(`$A$`)` interval forces the addition of the free `turnTo(`$C$`)` interval due to the plan invariant. Similarly, if the `pointAt(`$A$`)` interval is removed from the `Attitude` attribute, then the free interval `TurnTo(`$C$`)` is no longer justified, and is removed from the plan.

In EUROPA, the parameter equivalence is handled by creating a new variable for the required `turnTo` interval and posting an equivalence constraint between the parameters. In this example, the parameter $C$ of the `turnTo` predicates has been equated with the parameter $A$ of the sequenced

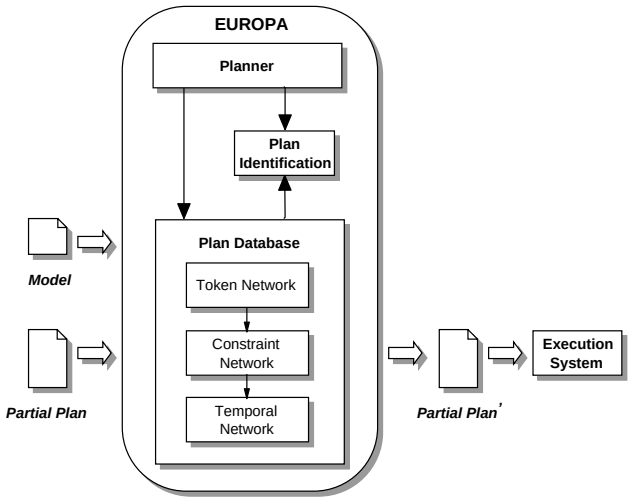


Figure 4: Planning For Execution With EUROPA

`pointAt()` interval. Finally, we note that $C$ has not been bound to any particular value, while parameter $D$ of the other `turnTo` has been bound to value $d$.

A partial plan in EUROPA consists of a mapping of attributes to sequences of intervals, a set of free intervals, and a set of constraints on variables in the given intervals. We assume for simplicity that the set of flaws of a partial plan is comprised of free intervals and unbound variables [1]. A plan identification function, then, takes the set of free intervals and unbound variables in the plan database and returns a subset of these in response to a query from the planner.

## Plan Identification in EUROPA

Figure 4 shows the overall architecture of EUROPA in the context of planning for execution. The system is composed of the following modules: a *planner*, a *plan database*, and a *plan identification* module. Planning begins when the plan dabase is initialized with a partial plan and a domain model. During planning, a planner can query the plan database through the plan identification module for flaws in the initial partial plan. Flaws filtered by the PIF are assumed to be handled by the execution system. If no flaws remain and the plan is consistent, the planner concludes that a plan has been found. If flaws remain, however, the planner resolves the remaining flaws by updating the plan database. The plan database uses a constraint network to manage the consistency of variables and constaints and uses a temporal network to maintain consistency between temporal variables and the temporal relationships imposed by the configuration rules. The planning process continues to alternate, asking the plan identification module for flaws, and updating the plan database until a plan is found that satisfies the model configuration rules and the PIF.

In EUROPA, the PlanId function is implemented as a filtering operation on the set of flaws in the plan database. To

[1] Note that this set of flaws is only useful for planners that search in feasible space, but EUROPA can support other flaws as well.

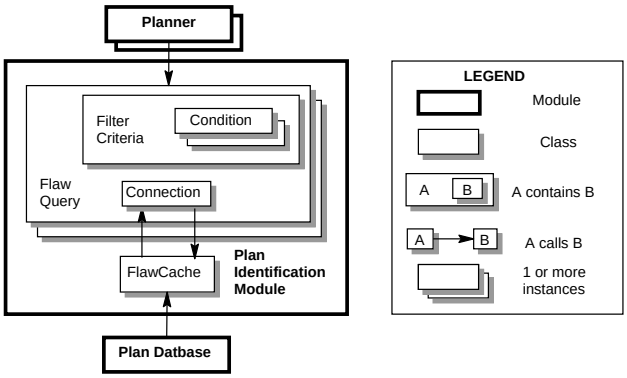


Figure 5: Class Diagram of the PlanId Framework

support this, the system must provide capabilities to:

1. obtain access to the set of flaws in the plan database;
2. define a filter expressing criteria for including or excluding a flaw;
3. obtain a set of filtered flaws by applying such a filter.

These capabilities are accomplished by providing:

1. a flaw storage mechanism, referred to as the `FlawCache`, which keeps the set of flaws in the plan database synchronized with changes made through explicit commitments by the planner or derived through inference.
2. a highly customizable filtering structure which allows predefined conditions and/or new custom conditions to be seamlessly integrated in a single filter.
3. a flaw querying facility which handles all access to the `FlawCache` and applies filtering criteria defined by the planner.

The remainder of this section describes in more detail the framework developed to achieve this in an efficient and customizable manner.

## Framework Class Diagram

Figure 5 presents the internal details of the *PlanId* module referenced in Figure 4. The `PlanDatabase` generates events indicating changes to intervals and variables when the plan invariant is invoked. These events are received by the `FlawCache` and used to maintain the set of all flaws in the system, i.e. all free intervals and unbound variables. Events indicating a restriction may cause a flaw to be removed from the `FlawCache` e.g. inserting a free interval or assigning a value to an unbound variable. Events indicating a relaxation may cause a flaw to be inserted into the `FlawCache` e.g. relaxing to domain of a variable or freeing an inserted interval.

A planner creates a `FlawQuery` at the beginning of the planning process. It is by means of a `FlawQuery` that a planner obtains the relevant subset of flaws as indicated by a filter. Planner-specified filters are defined in a `FilterCriteria` object, which is just a collection of `Conditions`. Each `FlawQuery` has exactly one `FilterCriteria` instance, provided to it during construction. `Condition` objects provide the customization necessary for planners to filter out flaws they wish to ignore. For a `Flaw` in the `FlawCache` to be returned by a `FlawQuery`, *all* `Conditions` must be satisfied.

In order to gain access to the set of flaws and the set of flaw changes, each `FlawQuery` establishes a `Connection` with the `FlawCache`. A `Connection` provides access to *all* flaws in the `FlawCache`. A `Connection` also provides a location to store information on changes in the `FlawCache` since the the `FlawQuery` was last queried. Notifications of changes in the contents of the `FlawCache`, i.e. flaws inserted or removed, are pushed to each connection from the `FlawCache` as the latter is synchronized with the `PlanDatabase`.

This architecture provides a number of useful features. First, the FlawCache can support many connections at once, enabling it to provide flaws to many planners. Second, a wide variety of simple conditions are provided, enabling a very large number of different PIFs to be expressible. Third, it is very straightforward to develop additional conditions making the approach very extendible. Finally, emphasis on lazy evaluation and event-based synchronization leads to efficient implementation.

```
while(done==false)
   if (isConsistent())
     filteredFlaws=getFlawsFromQuery()
     if (filteredFlaws.isEmpty()==false)
       nextFlaw = choose(filteredFlaws)
       resolve(nextFlaw)
     else done=true
   else... // rest of the algorithm omitted
end while
```

Figure 6: Planning with Flaw Queries.

Step 1:
FlawCache=$\{A, B, C$,`pointAt(`$D = d$`),turnTo(`$C$`)`$\}$
FilteredFlaws:$\{A, B, C$,`pointAt(`$D = d$`)`$\}$
nextFlaw: `pointAt(`$D = d$`)`
Step 2:
FlawCache=$\{A, B, C, E$,`turnTo(`$C$`),turnTo(`$E$`)`$\}$
FilteredFlaws:$\{A, B, C, E\}$
nextFlaw: $E$
Step 3:
FlawCache=$\{A, B, C\}$`turnTo(`$E = d$`),turnTo(`$C$`)`$\}$
FilteredFlaws:$\{A, B, C\}$
nextFlaw: $A$

Figure 7: Evolution of the flaws for the partial plan in Figure 3.

To see how the flaw filtering works, consider the sample partial plan shown in Figure 3. There are five flaws:

the variables $A$, $B$ and $C$, the `turnTo(`$C$`)` interval and the `pointAt(`$D = d$`)` interval; the FlawCache has these five flaws. Now suppose that the PIF filters out intervals with predicate `turnTo`. Then the set of filtered flaws consists of the three variables and the `pointAt(`$D = d$`)` interval.

The basic loop of a planner is similar to the fragment presented in Figure 6. At each step, the planner requests the filtered flaws. Once the flaws are retrieved, the planner uses some criteria to select a flaw, then uses another criteria to resolve the flaws. Application of the plan invariant and propagation of variable changes in the constraint network result in updates to the FlawCache. Subsequent queries to the FlawQuery will return a new set of flaws that accounts for these updates and the filtering of these flaws by the PIF.

To see this process in action, let us consider a few steps of planning given the partial plan and PIF that we have described. This process is shown in Figure 7. Let us assume that *choose* selects flaws according to some arbitrary order. Also suppose that *resolve* performs an insertion for free intervals or a variable assignment for unbound variables. After inserting `pointAt(`$D = d$`)` we see that the plan invariant ensures the creation of a `turnTo(`$E$`)` interval. The FlawQuery, however, indicates that the set of filtered flaws at step 2 only includes the variables $A, B, C, E$. At the next step, choose() returns flaw $E$; there is only one possible value, $d$, and thus the plan invariant doesn't lead to the creation of any new variables or intervals.

Such a simple filter could be achieved with a single condition which would check the predicate name of each interval flaw and exclude it if it matched the name `turnTo`.

## EUROPA Plan Identification Function Capabilities

EUROPA's PIF framework supports the following conditions, among others:

- Interval predicate filtering - filters all intervals of a particular predicate.
- Interval variable filtering - filters selected variable of all intervals with a particular predicate.
- Attribute filtering - filters all intervals and all variables of all intervals from a particular attribute.
- Temporal filtering - filters intervals according to a variety of temporal specifications. One example is a filter for intervals guaranteed not to happen within a temporal extent (a horizon filter).

In practice, different applications will impose different requirements on plan executives. The PIF framework allows considerable latitude in defining the capabilities of execution systems, and thus enables the planning technology to be more widely useful. However, it also provides considerable flexibility within a single application. Engineers can design different PIFs and analyze the resulting performance of the integrated planning and execution system, and choose the PIF that works best.

An execution system will typically only care about the plan developing inside the current execution window. If this execution system is implemented as a planner, a PIF could be used to focus the planning effort on that execution window only using the horizon filter. Such a PIF would contain a horizon condition that would specify, for each free interval and each unbound variable, whether it falls within the horizon or not. A free interval falls within the horizon if its start time and end time variable domains include the horizon timepoints. An unbound variable will fall inside the horizon if it belongs to one of the intervals that falls within the horizon.

The time at which an event actually occurs is usually different from the planned time. This difference can sometimes prove costly since it may cause some assumptions that were made in the planning stage to fail. In EUROPA, the temporal network is implemented as a Simple Temporal Network (DMP91). Simple Temporal Networks guarantee that if the network is consistent, an appropriate set of bindings of the temporal variables can be found in polynomial time. Thus, if the temporal network is consistent, no further commitments on time have to be made during planing. This is assuming that the executive is intelligent enough to be able to find this appropriate set. A PIF provides the means to define this flexibility using a condition that filters temporal variable flaws. If no temporal variable flaws are resolved by the planner, these decisions will remain unbound (though constrained by the temporal network) until execution time.

Overcommitment at planning time may prove costly in other ways. In cases where a plan consists of abstract and concrete tasks, the detailed task expansion of the abstract tasks may depend highly on when these tasks are executed. In such cases, it is better to let the execution system map abstract tasks into concrete tasks during execution. This frees the planner from generating concrete tasks, and allows the executive to choose the concrete tasks that best fit the actual execution. A EUROPA model can force abstract and concrete tasks to be inserted on different attrnibutes, and the PIF can filter flaws depending on whether they are allowed to be placed on "abstract attributes" or not. The EUROPA PIF mechanism associates variable flaws with the attributes that their parent intervals belong to, and so only unbound variables associated with "abstract attributes" will be returned as flaws.

There are instances of planning for execution when some planning decisions inside one execution cycle may determine what will happen in a future execution cycle. These commitments are sometimes unnecessary, especially in uncontrolled execution environments. These planning decisions may manifest themselves as particular predicate logic statements or as arguments to predicate logic statements. A PIF povides the means to delay commitment on these predicates or variables through a condition that filters flaws based on these predicates or variables.

## Complexity Analysis

In the simplest implementation, one could omit the `FlawCache` and `Connection` infrastructure. Resolving a query would be accomplished by iterating over all intervals and variables in the plan database and for each, applying the filter to test for inclusion or exclusion. This would result in a worst-case time-complexity given by $(N_v + N_i) * N_c * C_c$

where $N_v$ is the number of variables, $N_i$ is the number of intervals, $N_c$ is the number of conditions in the filter, and $C_c$ is the average cost of evaluating a condition. [2]

Since the points of greatest cost are in the evaluation of conditions, we seek to reduce the execution of condition tests. This is accomplished in a numnber of ways:

1. The last set of filtered flaws are cached in each `FlawQuery`.
2. The current set of flaws in the plan database are cached in the `FlawCache`.
3. Each cache is maintained through notifications of changes.
4. Conditions may be ordered to fail fast, based on the characteristcis of each problem.
5. The FlawQuery is updated only when the planner consults it for the latest set of flaws. Thus, the queries are only run on the set of flaws that were added since the last query.

The resulting worst-case cost of a query is approximated by: $N_+ * N_c * C_c$ where $N_+$ is the number of flaws inserted into the flaw cache since the last query.[3] The approximation omits the cost of caching events during synchronization of the FlawCache and the PlanDatabase. This is reasonable since the costs of caching are much less than the cost of evaluating the conditions over all insertions. Notice that we do not need to worry about flaws that are removed from the cache, since they aren't returned to the planner in any case.

## Related Work

A wide variety of agent architectures have been designed to support both planning and execution. We will not describe all aspects of these systems here; instead, we focus how these systems characterize the interaction between planning and execution.

Many integrated planning and plan execution frameworks define a fixed boundary between their components. These systems also use different modeling languages, in some cases with different semantics, and thus have potential problems with model synchronization. Finally, these systems do not have a crisp declarative characterization of the boundary between the components. Examples of integrated planning and plan execution systems in this category are O-Plan (TDK94), 3T (BFG+97), and Propice-Plan (DI99).

Cypress (WMLW95) is a planning and plan execution framework designed for a variety of applications, including military operations. Cypress is a loosely coupled integration of the SIPE planner, the PRS reactive execution system, and Gister-CL system for reasoning under uncertainty. Cypress enables human intervention during planning and plan execution. Cypress uses the ACT representation to model both planning and execution. The boundary between SIPE and PRS is flexible, as PRS can invoke SIPE to handle run-time plan failures. However, there is no facility in Cypress to describe the boundary between the planner and plan execution in a declarative way.

The Remote Agent (RA) (MNP+98; JMM+00) is an agent architecture for spacecraft control that was used in a 2-day experiment of an autonomous probe. The RA consisted of a planner, a plan execution system, and a mode identification and reconfiguration system. The RA planner built plans that were temporally flexible so that the plan execution system could decide on-the-fly which tasks to start and end (MMT98). This represented a significant advance at the time; however, other applications using the RA could not use any other divide between planning and execution. Furthermore, the three components of the system used different modeling languages with different semantics, requiring considerable effort to ensure model synchronization.

IDEA (MDF+02; DLM03) is an agent architecture designed to overcome shortcomings in the RA approach to agent modeling. IDEA provides a simple virtual machine that supports plan execution, consisting of a model, plan database, plan runner, and reactive planner. The job of the reactive planner component of an IDEA agent is to ensure that a "locally executable" plan is returned. Thus, a crucial task is to define the scope of the Reactive Planner's job. The PIF is a natural way to focus on those parts of the model that must be addressed by the Reactive Planner. IDEA also supports many planners operating on the same plan database, and thus the same model. PIFs are a natural way to define the scope of these various planners in order to ensure that planners do not step on each others' toes. IDEA also supports multi-agent architectures using inter-agent communication. The original notion of IDEA is to separate models for each agent; these models are intended to be written in the same language and share components. Partial plans serve as the medium by which planners communicate with the executive, as well as the medium by which IDEA agents communicate with each other. However, the PIF can (in principle) be used to simply divide up the model amongst the agents in a similar manner to the way it divides up models amongst planners; the crucial problem to solve is dividing plan databases efficiently among the IDEA agents.

## Conclusions and Future Work

We have described plan identification functions as a way of circumscribing the planning problem that *must* be solved in order to create an executable plan. PIFs have the advantage of enabling a single model to characterize both the planning problem and the plan execution problem. They also enable easy characterization of the boundary between planning and plan execution, even in cases where different models for planning and execution are used. They also provide considerable flexibility, as they allow the boundary between planning and execution to be adjusted. We have described the implementation of the PIF framework of EUROPA, and shown how it can be used to implement many PIFs for different type of plan execution systems.

We have implicitly assumed that a single model of system behavior can be written, so that PIFs can be used to separate

[2] In practice only some of the conditions will be executed since we discard the flaw after the first condition fails.

[3] $N_+ << (N_i + N_v)$ since there are relatively few flaw insertions resulting from each planner commitment.

the part of system behavior that pertains to the execution system. The IDEA project (MDF$^+$02) is pursuing this notion, but it remains to be seen how the concepts extend to more sophisticated planning and control architectures.

We have described one way of using PIFs to divide a model amongst many planners. This approach does not address important architectural issues of mutli-agent access to a shared plan representation. It also doesn't address the issue of how to structure the plan representations used by planners and executives. The efficient implementation of PIFs may be impacted by this architecture.

Note that while the plan that is passed to the executive may define a set of plan completions, there is no reason to assume that the executive chooses one of these completions, and in fact no way to characterize the actions of the executive in a declarative way.

PIFs can be used for more than just separating planning from the execution system. One can also imagine partitioning the planning problem into many different problems using a collection of PIF functions.

## References


R. Bonasso, R. Firby, E. Gat, D. Kortenkamp, D. Miller, and M. Slack. Experienences with an architecture for intelligent, reactive agents. *Journal of Experimental and Theoretical Artificial Ingelligence*, 9(2), 1997.

O. Despouys and F. Ingrand. Propice-plan: Towards a unified framework for planning and execution. In *Proceedings of the 5th European Conference on Planning*, 1999.

M. Dias, S. Lemai, and N. Muscettola. A real-time rover executive based on model-based reactive planning. In *Proceedings of the International Conference on Robotics and Automation*, 2003.

R. Dechter, I. Meiri, and J. Pearl. Temporal constraint networks. *Artificial Intelligence*, 49:61–94, 1991.

J. Frank and A. Jónsson. Constraint based attribute and interval planning. *Journal of Constraints*, To Appear, 2003.

A. Jónsson, P. Morris, N. Muscettola, K. Rajan, and B. Smith. Planning in interplanetary space: Theory and practice. In *Proceedings of the Fifth International Conference on Artificial Intelligence Planning and Scheduling*, 2000.

N. Muscettola, G. Dorais, C. Fry, R. Levinson, and C. Plaunt. Idea: Planning at the core of autonomous reactive agents. In *Proceedings of the 3d International NASA Workshop Planning and Scheduling for Space*, 2002.

N. Muscettola and P. Morris. Execution of temporal plans with uncertainty. In *Proceedings of the* $17^{th}$ *National Conference on Artificial Intelligence*, 2001.

P. Morris, N. Muscettola, and I. Tsamardinos. Reformulating temporal plans for efficient execution. In *Proceedings of the* $15^{th}$ *National Conference on Artificial Intelligence*, 1998.

N. Muscettola, P. Morris, and T. Vidal. Dynamic control of plans with temporal uncertainty. In *Proceedings of the* $17^{th}$ *International Joint Conference on Artificial Intelligence*, 2001.

N. Muscettola, P. Nayak, B. Pell, , and B. Williams. Remote agent: To boldly go where no ai system has gone before. *Artificial Intelligence*, 103(1-2), 1998.

D. Smith, J. Frank, and A. J/'onsson. Bridging the gap between planning and scheduling. *Knowledge Engineering Review*, 15(1), 2000.

A. Tate, B. Drabble, and R. Kirby. O-plan2: An open architecture for command, planning and control. *Intelligent Scheduling*, 1994.

D. E. Wilkins, K. L. Myers, J. D. Lowrance, and L. P. Wesley. Planning and reacting in uncertain and dynamic environments. *Journal of Experimental and Theoretical Artificial Ingelligence*, 7(1), 1995.