ELSEVIER

Artificial Intelligence 76 (1995) 319–375

Artificial Intelligence

A general programming language for unified planning and control

Richard Levinson *

Recom Technologies Inc., NASA Ames Research Center, Mail Stop: 269-2, Moffett Field, CA 94035, USA

Received June 1993; revised March 1994

Abstract

This paper presents a method for embedding predictive search techniques within a general-purpose programming language. We focus on using this language to program the behavior of a real-time control system. Our goal is the ability to write complex programs that can be interpreted by both a real-time controller and an associated planner. The language provides an expressive action representation which captures the procedural complexities of *practical* control programs, yet can still be projected by a search-based planner. To support integration with the real-time controller, the planner can provide useful advice when it is interrupted after an arbitrary amount of computation. The system provides a unified approach since the planner and the controller share identical data structures and algorithms for interpreting a shared action representation. This unified representation facilitates very tight integration between the planner and the controller.

1. Introduction

Developments in the field of real-time control have had an enormous impact on modern society. Process control software has revolutionized such critical industries as medical and scientific instrumentation, automobile production, and air traffic control. However, there are limits to the capabilities of modern control software. Programming can be viewed as the art of specifying all control details in advance. This requirement to encode all behavior in advance inhibits automation in applications where the environmental conditions and the effects of control actions cannot be fully predicted at design time. In this paper, we consider how AI planning techniques can improve the ability of real-time control software to operate in unexpected situations.

* E-mail: levinson@ptolemy.arc.nasa.gov.

SSDI 0004-3702(94)00075-1

In order to operate under novel conditions, control systems may use both feedback and feedforward techniques. Feedback and feedforward are complementary methods of using sensory input to select an appropriate control action. *Feedback* is typically used to select actions based on observations of errors that have already occurred, and it relies on an ability to *monitor* the controlled process. *Feedforward* is used to select actions based on predictions of errors that could happen in the future, and it relies on an ability to *predict* the behavior of the controlled process [10]. Dean and Wellman have noted that planning is a form of feedforward control [10]. Planning serves as a feedforward method for extending a controller's operating range by generating novel control behavior to handle non-routine situations.

1.1. Background definitions

We begin by defining some background terminology, starting with the central concept of *search*. Search is the process of selecting a subset of elements with desirable properties from a superset of choices. Selected elements are evaluated and replaced if they do not have the desired properties. In our system, *planning* means using predictive (look-ahead) search to select effector commands. Thus, planning involves using prediction to select a set of effector commands that achieve a given goal. *Real-time control* means reading sensors and executing effector commands in bounded time (without search). The fundamental difficulty for embedding planning within real-time control software is the fact that the planner's search process cannot guarantee a solution within the bounded time required for real-time control.

The planner often must meet real-time deadlines that vary with the given problem. This class of planning has been called time-dependent planning by Dean and Boddy [9]. They identified a class of algorithms called *anytime algorithms* that are useful for meeting the demands of time-dependent planning [9]. They define anytime algorithms as procedures that are interruptable and ready to provide increasingly useful results at any time [2,9]. Further, they identified heuristic search methods as being likely candidates for use as anytime algorithms [2,9]. Our system combines heuristic search methods with anytime interruptability for algorithms written in a general programming language.

Finally, we define the term *choice point* to mean a step in a program where a single choice must be selected from a set of alternatives before the program can continue. Each of these terms will be discussed in depth throughout the rest of the paper. For now, these preliminary definitions will suffice.

1.2. Research goals

Our primary goal is to test the following:

Top-Level Hypothesis. Planning techniques can extend the operating range of a real-time controller.

The class of planning techniques required to test this hypothesis involves situated, predictive search for real-time, closed-loop control. In other words, we expect our planner to operate while connected to sensors and effectors, with no human interaction. Since the planner must operate in real-time, it must be interruptable and ready to provide useful results at *any time*. We are particularly interested in *autonomous* control, where human intervention is not possible and the system must rely on its own sensors. Our efforts are driven by a NASA application concerned with developing autonomous scientific instruments that can operate without human assistance in remote or hostile environments [28].

Extending a controller's operating range is an important obstacle in the path toward achieving autonomous controllers. As controllers get more complex and less reliant on humans, a given controller may be situated in a wider variety of environments. An *autonomous* controller will eventually face unexpected situations, and therefore requires the capacity to operate under conditions for which it was not entirely pre-programmed. Due to *lack of information* and *finite effort* at design time, a control program may not have been written to account for every possible situation. Currently, controllers are written as deterministic programs. The programmer selects certain situations to handle, and supplies error messages for deviant cases. If deviant situations are not predicted by the programmer and trapped with an error message, the behavior of the controller may simply be undefined and potentially dangerous. A good measure of a system's autonomy is its ability to operate in unusual situations for which it wasn't explicitly programmed. Such behavior is analogous to a student who has been taught exactly how to execute a "textbook" procedure under routine conditions, but must plan how to modify the procedure to handle real-world complexities.

1.3. Limitations of current approaches

In this section we discuss the limitations of current approaches which make it difficult to achieve the above goals. To test our top-level hypothesis, we need a system that integrates predictive search with real-time control programs. Although some initial efforts have begun [13,25,32], few such systems exist today. In considering why this is so, we observe two problems due to a language barrier between the planner and the controller.

Observation 1. Most control programs are written in a general programming language.

Most control system designers know how to program, but they do not know how to use AI planning systems. This is because the designers require the procedural expressiveness of a general language. We would like to provide the vast numbers of control system engineers with access to planning techniques—*in their own environment*. Our goal is to provide a language that allows controller designers to incorporate planning techniques into their control programs relatively seamlessly. They must be able to use standard programming techniques such as hierarchical procedure decomposition, variable assignment, iteration and conditional control. Even when the programmers *do* know how to use planning techniques, they might not be able to use them because of the next observation.

Observation 2. Planning systems usually do not scale up to real-world control applications.

The class of applications we have in mind involves real-time closed-loop control. One important reason that planning systems don't scale up to these sorts of applications is that they *don't* use a general-purpose programming language. Planning systems typically require that control behavior be encoded in an unconventional, procedurally inexpressive representation. The representations are unconventional because they do not use standard programming constructs and they require specialized training and knowledge about AI planning methodology. Planning systems typically use a representation that cannot capture the procedural expressiveness required for real hardware control such as hierarchical and modular procedures or complex iterative and conditional control. However, hierarchical procedure decomposition is essential for encoding practical control behavior because it works like *subgoaling*, which is a method that is widely recognized to reduce search [23]. Planning systems that *do* support hierarchical decomposition typically do not provide results until they generate a complete plan. This decreases their value in deadline constrained situations [9].

One problem that has impeded the development of procedurally expressive planning techniques is the difficulty of analyzing conditional operators and effects [38]. Yet another problem is that the planner typically uses an action representation that is different from that used by an associated controller (which is usually written in a general programming language). This forces the designer to encode and maintain different control procedures for planning and execution. It also makes it difficult to transition smoothly back and forth between planning and execution because the data structures and processing algorithms speak different languages [19]. Another large obstacle to the incorporation of planning systems into real-world, real-time control applications is the need to respond quickly to a changing environment [10, 17, 21]. Most systems that generate and then execute plans operate with the assumption that the external world doesn't change in unexpected ways. All of these factors impede the transfer of planning systems to real-world control applications.

1.4. Overview of our approach

We have developed the PROcedure Planning and Execution Language (PROPEL) to address these problems so that we can more accurately evaluate how planning can help real-time control systems. An alternative expansion for the name PROPEL is: the *PROgrammer's* Planning and Execution Language. This alternative emphasizes our goal to provide planning techniques in a form that is familiar and thus accessible to the general programming community.

We describe a predictive search technique that can be embedded within a dialect of LISP using a software abstraction called the *choice point*. The controller relies on pre-programmed choices for routine situations, but uses predictive search to evaluate choices and generate novel behavior in unusual situations. We use the term *pre-programmed* to mean the degree to which choices are eliminated at design time. The primary reasons for not pre-programming all control behavior are *finite programmer effort* and *programmer*

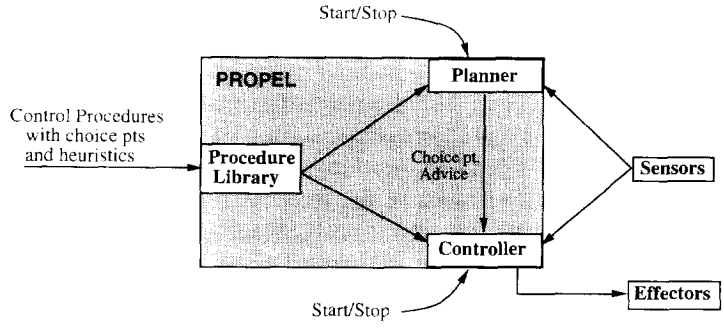

Fig. 1. Functional overview of PROPEL.

uncertainty about action outcomes and a changing environment.

To encode control programs, PROPEL uses a general programming language that is shared by an interruptable planner and an associated real-time controller. The control programs are written in a dialect of LISP that provides nondeterministic constructs, called *choice points*, for subroutine calls and assignment statements. Additionally, the designer can specify a set of default heuristics that will be used by the controller to instantiate the control programs in bounded time without using search.

Fig. 1 shows a sketch of the functional relationships between the three primary components in PROPEL: the procedure library, the interruptable planner and the real-time controller. The procedures in the library can be interpreted by both the planner and the controller, both of which have access to the sensor readings. The effectors on the other hand, are only accessible to the controller. The planner performs look-ahead search on the procedures and advises the controller about which selections to make at choice points. The planner and the controller can each be started and stopped asynchronously by an application-specific executive.

At *design time*, a programmer designs a set of effector control procedures that contain choice points. At *run time*, the planner uses predictive search to simulate and evaluate alternative procedure instantiations. When a successful procedure is instantiated or the planner is interrupted, the planner's choices are collected into rules that advise the controller while it executes the procedure. The controller can also execute the procedures without the advice of the planner through the use of heuristics that guide a default *reactive* instantiation.

1.5. Benefits of this approach

Our approach allows programmers to use predictive search to provide "exception handling" for the controller. In unexpected environmental conditions, the planner's advice can enable the controller's behavior to degrade gracefully, instead of producing a "hard failure" in the form of an error message. When the state of the controlled process gets outside the normal operating range of the controller, the planner can construct an "in-situ" handler by searching through a space of procedure instances for an appropriate combination of effector commands.

Embedding choice points within a general programming language allows us to explore the role of search alongside deterministic, pre-programmed control. This is important because the tradeoffs between pre-programming and predictive search are not well understood. When predictive search and general programming techniques co-exist in a single language we are forced to address the question of which decisions *can* be pre-programmed "off-line", and which ones *need* to be determined using search at run time. This dichotomy is at the root of much of the recent debate between *reactive* and *predictive* systems. All control systems take a position on this issue by pre-programming some parts of their application, but the tradeoff is rarely discussed within the context of a single action representation. PROPEL facilitates analysis of this tradeoff between pre-programming and search by allowing programmers to experiment with both options.

Since a unified action representation for planning and control is rare, many systems represent behavior as either entirely reactive (pre-programmed) or entirely predictive (search-based). *Reactive* systems like PRS [17], GAPPS [21] and Brooks' subsumption architecture [5] rely exclusively on pre-programming, while *predictive* systems like O-plan [8], SIPE [38] and SNLP [30] rely almost exclusively on search. We choose a middle position on this continuum. We recognize that all decisions that *can* be pre-programmed, should be. But we also recognize that controllers can be used in situations outside of their normal operating conditions. In other words, we do not need to build plans from scratch, but we also cannot pre-program all of the controller's behavior. Other systems that occupy this middle ground typically partition behavior into two distinct classes, called *planning actions* and *control actions*, that use different action representations [10, 19, 38]. In contrast, we use an action representation that is shared by both the planner and the controller. This facilitates a much tighter integration of planning and control than is possible when the two components use different languages.

Many significant questions arise regarding the use of predictive search within a general programming language. Although PROPEL certainly does not provide all of the answers, it is intended as tool for exploring and studying the issues. We hope that embedding predictive search within general programs will enable much more widespread use of planning techniques by control system programmers. Until we understand how planning techniques can exist alongside pre-programmed control, the AI planning techniques will remain isolated. Isolated techniques will not be widely used and thus will never be fully tested or evaluated.

1.6. The structure of this paper

The structure of this paper is as follows: Section 2 introduces the core software abstraction concept of our system, called the *choice point*. With that as a base, we introduce an example application in Section 3 that will be used throughout the paper. Section 4 comprises the technical core of the paper, using the running example to illustrate a detailed description of how the system works. Section 5 presents the results of two experiments that illustrate PROPEL's behavior and demonstrate how planning can extend the controller's operating range. Section 6 summarizes pointers to related work,

and Section 7 provides a qualitative evaluation of our system in terms of its assumptions, limitations, contributions, and future work.

2. Introduction to the choice point abstraction

The key concept behind this entire system is a software abstraction, called the *choice point*, that defines a search space within the context of a deterministic procedure. The purpose of this section is to illustrate how the choice point software abstraction defines a search space, and how this affects the semantics of an otherwise normal program. In this section, we will introduce the core effect of the choice point abstraction without addressing issues of hierarchical procedures, sensing and acting, and closed-loop control. Those issues, and a detailed technical description of how the choice point abstraction works, are discussed in Section 4.

We begin by extending a general programming language to include choice points in the form of nondeterministic assignment statements. These choice points, called *choose-value* statements, take the general form of:

(⟨variable⟩ ← (**choose-value** ⟨choices⟩
:heuristic ⟨preference-function⟩)).

Intuitively, this statement says that the variable on the left of the arrow will be assigned a single value that is heuristically selected from the set of given choices. However, the choices that were not selected represent alternative instantiations of the assignment statement. Thus each choice point defines a *disjunctive set* of alternative assignment values. A program that contains choice points therefore defines a set of disjunctive procedure instances, where each instance results from a unique set of choice point selections.

We have developed a search engine that interprets these procedures. The choice points generate branches in a search tree of nodes that correspond to computational processes. The root node of the tree corresponds to an initial procedure call, and child nodes correspond to disjunctive "continuations" of a parent node. Each *continuation* corresponds to a unique instantiation of the parent process' choice point. Each path through the tree defines a unique procedure instance. A single PROPEL procedure can represent a game playing controller as follows:

```
(Defprocedure play-game (board)
   :Body
   (moves ← nil)
   (Until (game-over? board)
      Do (move ← (choose-value (legal-moves board)
                               :heuristic (best-moves)))
         (board ← (change-board move board))
         (push move moves))
   (print-msg "The solution is: " (reverse moves)))
```

The `play-game` procedure accepts a structure called a `board` as input. Until the board is in some termination state, `play-game` iteratively chooses a legal move and then changes the board by applying the selected move to the current board. The moves are collected and then printed out at the end. If the functions `game-over?`, `legal-moves` and `change-board` are defined appropriately, this controller could play the 8-puzzle.

The core of our system is a search engine that interprets this procedure, generating a search tree for the board game. Search nodes in the tree correspond to computational continuations, with the root node representing the initial procedure call. The search tree is generated by iteratively selecting and expanding nodes. In this context, *expanding a node* corresponds to executing the next instruction of a computational process. Whenever the instruction

(move ← (**choose-value** (legal-moves board) **:heuristic** (best-moves)))

is executed, the current node splits, producing child nodes for each legal move. Each child represents a distinct continuation of its parent, based on the selection of a different choice.

Although the `play-game` procedure encodes the control structure required to generate a single behavior instance, a *complete* search space is implicitly declared by the `choose-value` statement. We use heuristic search control methods to avoid exhaustive exploration of that search space. In the above example, the heuristic `best-moves` is used to rank the legal moves, and a beam-width is used to restrict the branching factor. If the functions `legal-moves` and `change-board` are designed to model the 8-puzzle, the resulting search tree looks like a standard search space for the 8-puzzle [33].

This example illustrates a fundamental benefit of the choice point abstraction for control system programmers. The controller's behavior (iteratively selecting and executing legal moves) is encoded independently from the search engine's behavior (iteratively selecting and expanding nodes). As the designer learns which moves are better, only the heuristics change. There is no need to modify the controller's procedure definition at all. Since the facility for searching and evaluating alternative procedure instances is parcelled out, it is not entangled with the controller's behavior definition. Thus, changing only the heuristic will produce different behaviors, without interfering with the completeness or correctness of the controller's behavior. In addition to being more robust in the face of changing heuristics, the automatic handling of choices saves designers from having to implement their own search engine and backtracking scheme (which would tend to get intertwined with the controller's behavior definition).

3. An example application

In this section, we introduce an example that will be used throughout the rest of the paper. It is a simplified application that has been chosen for illustrative purposes. We have also begun building a "real" PROPEL application that plans and executes experiments using scientific equipment [28]. For our running example, we have selected a "pickup-and-delivery" procedure using the NASA Tileworld simulator [34]. The simulator operates asynchronously with PROPEL, and serves as our controller's sensors and

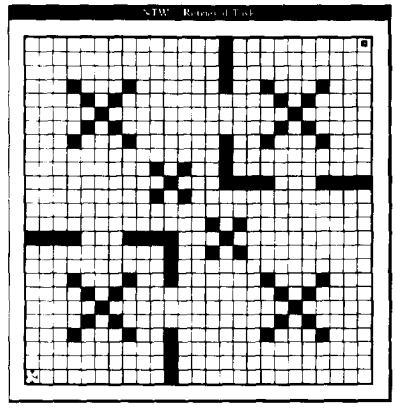

Fig. 2. A typical Tileworld problem.

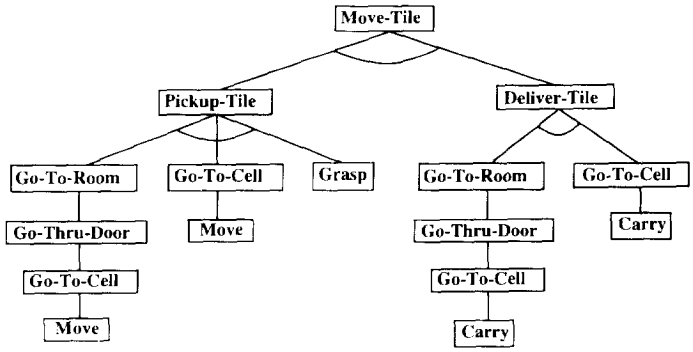

Fig. 3. The procedure hierarchy for our application.

effectors. This example is based on a set of problems originally designed by Bresina [3].

Fig. 2 illustrates a typical Tileworld problem. This is a two-dimensional grid world, containing a single agent that can MOVE north, south, east, or west in discrete steps. The grid world includes two rooms, each with two doors. The area between the rooms is called the hallway. The agent's job is to move a given tile to a new location. In all problem instances used in this paper, the agent starts in the lower left corner, at cell (0 0), and its goal is to pick up the tile in the upper right corner, at cell (24 24), and deliver it to cell (1 1).

Fig. 3 shows that moving a tile consists of first picking the tile up from its current location, and then delivering it to its destination. To pick up a tile, the agent must move to the same room as the tile, move next to it, and grasp it with one of its four grippers. To deliver the tile at its destination, the agent must move into the destination room before moving to the actual delivery location. In the simplest situation, the agent can move straight toward the doors, the tile, and the destination location. However, sometimes the agent faces more difficult situations that involve getting around obstacles.

Fig. 3 illustrates a sketch of the procedure hierarchy for this application. This figure shows that the structure of this problem lends itself to a procedural decomposition

approach because picking up a tile, delivering it, and moving through doors are natural subroutines for achieving the overall goal. Also, some procedures such as GO-THRU-DOOR and GO-TO-CELL can be encoded as modular subroutines that get used by more than one higher-level procedure. PROPEL's hierarchical and modular procedures effectively capture these properties. Fig. 3 only indicates the procedure hierarchy in our application. The next section discusses the use of iteration, conditional and choice point constructs by examining the procedure definitions in more detail.

3.1. Example procedure definitions

We present the following procedure definitions to serve three purposes. First, they illustrate the syntax of our action representation. Secondly, they provide a concrete definition of our example controller's behavior. And finally, they are used in Section 4 to illustrate the technical discussion of how PROPEL works. See Appendix A for PROPEL's full grammar specification, and see Appendix B for the full listing of PROPEL procedures used in this application.

```
(Defglobal *state* (read-sensors))
```

We begin by declaring a global variable called `*state*` to store the sensory state of the environment. This PROPEL variable is not the same as a standard "global" variable in LISP and will be described further in Section 4. The LISP function `read-sensors` identifies the locations of the agent, the tile to be delivered, and the static obstacles. The sensors also determine if the agent is grasping a tile with one of its four grippers. For this example, we assume the agent has global sensing capabilities so that it sees the contents of all 625 grid cells. We discuss the effects of relaxing this assumption in Section 7.

```
(Defprocedure move-tile (tile destination)
   :Body
   (pickup-tile tile)
   (deliver-tile destination))
```

`Move-tile` is the top-level procedure for moving a tile to a given location. When a top-level PROPEL procedure is invoked, it initializes all global variables. So, when `move-tile` is invoked, the `*state*` variable will be initialized by reading the sensors as described above. The `move-tile` procedure calls one subroutine to pick the tile up from its current location, and another subroutine to deliver it to the destination location.

```
(Defprocedure pickup-tile (tile)
   :Body
   (tile-loc ← (get-tile-loc tile *state*))
   (go-to-room (what-room? tile-loc))
   (go-next-to-cell tile-loc)
   (grasp-tile (adjacent-direction (get-agent-loc *state*) tile-loc)))
```

The `pickup-tile` procedure first binds a local variable, `tile-loc`, to the tile's cell coordinates. This is achieved by calling the ordinary LISP function `get-tile-loc` to

perform a simple lookup operation using the global variable `*state*`. The local variable is then used in the next three statements. After going to the room that contains the tile, the agent moves to a cell next to the tile, and then grasps the tile. This procedure illustrates the use of both local and global variables, and the use of ordinary LISP subroutines.

```
(Defprocedure go-to-room (destination-room)
   :Body
   (agent-room ← (what-room? (get-agent-loc *state*)))
   (IF (not (equal agent-room destination-room))
       Then (IF (not (hallway? agent-room ))
                Then (go-thru-door agent-room 'exit))
            (IF (not (hallway? destination-room))
                Then (go-thru-door destination-room 'enter))))
```

`Go-to-room` encodes the behavior for going to a room. If the agent is not already in the destination room, then if it is not in the hallway, it exits the current room. After the agent is in the hall, it will enter the destination room unless the hall is the destination. This procedure illustrates the conditional flexibility provided by our action representation.

```
(Defprocedure go-thru-door (room direction)
   :Body
   (door-loc ← (choose-value (door-locations room)
                  :heuristic (closest-loc (get-agent-loc *state*))))
   (doorstep-loc ← (get-doorstep-location door-loc direction))
   (go-to-cell doorstep-loc)
   (move-dir ← (adjacent-direction doorstep-loc door-loc))
   (IF (grasping-object? *state*)
       Then (carry move-dir)
            (carry move-dir)
       Else (move move-dir)
            (move move-dir)))
```

The `go-thru-door` procedure is used for both entering and exiting a room. Since each room has two doors, the agent must first choose which door to use. The `choose-value` statement in this procedure represents that choice point. After *choosing* a door, the agent locates the cell adjacent to the door, called the *doorstep*. Depending on whether the agent is entering or exiting the room, the doorstep will be on the inside or the outside of the room. After going to the doorstep, the agent takes two steps through the doorway. If the agent is grasping a tile, `go-thru-door` calls the `carry` procedure to get through the doorway, otherwise it calls the `move` procedure. Aside from going through doorways, all movement by the agent, including going to the doorstep, is controlled by the workhorse subroutine `go-to-cell` which is shown below:

```
(Defprocedure go-to-cell (goal-loc)
   :Body
   (Until (equal (get-agent-loc *state*) goal-loc)
      Do (move-dir ←
            (choose-value '(N S E W)
               :heuristic (closest-dir (get-agent-loc *state*) goal-loc)))
         (choose-procedure (take a step in move-dir goal-loc)
            :heuristic (prefer-first-choice))))
```

The `go-to-cell` subroutine is called in order to move the agent to a door, tile, or destination location. It is also where most of the search occurs in the application. The procedure instructs the agent to go to a location by repeatedly choosing a direction and then choosing and executing a subroutine for moving in that direction, until it is in the goal location. There are thus two choice points on each iteration, one for selecting the direction and one for selecting the action. This procedure illustrates a second form of choice point that PROPEL supports: the `choose-procedure` statement. This statement is a nondeterministic *subroutine call*, in contrast with `choose-value`, which is a nondeterministic *assignment statement*. Choose-procedure can be viewed as an instruction to *achieve a subgoal*. In this example, the subgoal is "take a step in move-dir toward goal-loc". Choose-procedure statements will be discussed further in Section 4. Without any obstacles to cause backtracking, the solution for our Tileworld example requires nearly 100 iterations through this loop, passing through nearly 200 choice points. PROPEL prevents endless loops by detecting and pruning redundant (duplicate) search nodes.

3.2. *Features of this example*

The first important property of this example is that it illustrates the procedural expressiveness of PROPEL. The natural decomposition of this problem lends itself to the general programming techniques of hierarchical and modular procedures with iterative and conditional control constructs. Reactive languages such as PRS [17], RAPs [15], and RPL [31] can also represent the hierarchical, conditional and iterative structure of this example, but they encode only deterministic programs that correspond to our default program instances. We contrast this with PROPEL's ability to encode a *planner's* search space of disjunctive procedure instances using choice points.

This example also illustrates how a controller may be faced with situations of increasing difficulty. The controller was initially written only to move a tile within a single room. `Go-to-cell`'s simple Manhattan distance minimization heuristic worked fine in this simple case. Then the controller was to be used in a more complex situation that required moving a tile between different rooms. We used our original hill-climbing procedure from the simple case as a subroutine for this harder case, but we needed to add new procedures for entering and exiting rooms. Finally, the controller is required to move the tiles in the presence of obstacles that trap the hill-climbing heuristic. If the location and size of obstacles is not known until run time, then an efficient plan cannot be pre-programmed.

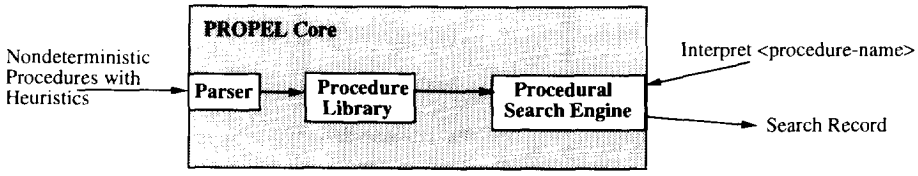

Fig. 4. The PROPEL core: a procedural search engine.

This example also allows us to explore the tradeoffs between search and pre-programmed approaches towards extending a controller's operating range. The designer can have the controller print error messages when it runs into obstacles, or they can build in a form of reactive behavior that can physically backtrack out of deadends. A third option is to use predictive search at run time to generate an appropriate plan. PROPEL allows us to explore and evaluate each of these options. In Section 5 we present experiments that demonstrate some of these options.

4. How PROPEL works

In this section, we present a technical description of how our system operates by presenting PROPEL's primary data structures and processing algorithms. We use the above Tileworld example to illustrate the technical concepts throughout the discussion.

We present PROPEL as a sequence of three design layers. The first layer is the core of the system. It consists of a library of procedures that contain choice points, and a procedure interpreter that operates as a search engine. The second design layer includes extensions to the core that are required when the procedures use physical sensors and effectors. In this context, the search engine operates as both a *planner* and a *controller*. The third design layer involves the development of application-specific *executives* that manage the relationship between the planner and the controller to achieve closed-loop planning and control.

4.1. The core: a procedural search engine

The core of our system is a *procedural search engine* that implements the choice point abstraction. The search engine operates on arbitrarily complex procedures instead of specialized, procedurally inexpressive representations like IF–THEN rules or STRIPS operators [14]. We are not using any sensors or effectors at this first design layer, so we do not distinguish the concepts of planning, sensing, or control from strictly computational behavior.

Fig. 4 illustrates the components in this core layer. First, the designer enters a set of application-specific control procedures that contain choice points and heuristics. The procedures are immediately parsed into a form required by the search engine. When the search engine is instructed to interpret a specific procedure, it generates a search tree of procedure instances. Each leaf in the tree corresponds to a distinct procedure instance

defined by the path of choices from the root to the leaf. A search record that describes the search tree is produced as the output of the search engine.

4.1.1. Nondeterministic procedures

A PROPEL application begins with a set of *nondeterministic procedures* that are represented in a dialect of LISP that includes two forms of choice points. In particular, `choose-value` statements correspond to nondeterministic assignment statements, and `choose-procedure` statements correspond to nondeterministic subroutine calls. Both statements represent a step in the program where a single choice must be selected from a set of alternatives before the program can continue. The `go-to-cell` procedure below illustrates both types of choice points. Each choose-statement includes a heuristic function that sorts the choices according to local heuristics. PROPEL also uses global heuristics which are discussed when we describe the search engine later in this section.

```
(Defprocedure go-to-cell (goal-loc)
   :Body
   (Until (equal (get-agent-loc *state*) goal-loc)
      Do (move-dir ←
            (choose-value '(N S E W)
               :heuristic (closest-dir (get-agent-loc *state*) goal-loc)))
         (choose-procedure (take a step in move-dir goal-loc)
            :heuristic (prefer-first-choice))))
```

Although not present in the `go-to-cell` procedure shown above, procedures may optionally be associated with a *goal pattern*. Choose-procedure statements use the goal patterns to match a set of potential subroutine procedures. Thus, the choose-procedure statement operates like a subgoal statement in a backward chaining system [33] that uses a goal pattern to identify matching procedures. Each procedure may also be associated with an optional set of *preconditions* that are tested whenever the procedure is invoked (either explicitly or by matching a choose-procedure goal). See PROPEL's full grammar specification in Appendix A for more detail.

The choose statements are called *nondeterministic* because the outcome of their execution is not uniquely defined [20]. PROPEL is written in LISP, which is itself a deterministic program. However, PROPEL procedures are nondeterministic above the software abstraction level of the choice point, because each choose statement has as many outcomes (continuations) as it has choices.

This integration of search into a general-purpose programming language is also useful for non-planning applications such as fault diagnosis and other forms of hypothesis generation. However, we will focus exclusively on planning applications in this paper.

4.1.2. The parser

The parser converts the nondeterministic procedures into *flat* code that does not contain any high-level iteration or conditional constructs. For example, the above `go-to-cell` procedure is converted into the form shown below. The **Until** loop was *flattened* into the conditional `GOTO` at step 0 and the unconditional `GOTO` at step 3.

```
(Defprocedure GO-TO-CELL (GOAL-LOC)
 :Body
  0:  (IF (EQUAL (GET-AGENT-LOC *STATE*) GOAL-LOC) GOTO 4)
  1:  (MOVE-DIR <-
       (CHOOSE-VALUE '(N S E W)
        :HEURISTIC (CLOSEST-DIR (GET-AGENT-LOC *STATE*)
                                GOAL-LOC)))
  2:  (CHOOSE-PROCEDURE (TAKE A STEP IN MOVE-DIR GOAL-LOC)
                        :HEURISTIC (PREFER-FIRST-CHOICE))
  3:  (GOTO 0))
```

Translating loops and complex conditionals into simple GOTO statements facilitates backtracking and the ability to interrupt the search process at any time. The flat code can be more easily split into continuations at choice points such as those at steps 1 and 2 in the procedure below. The flat code also allows the interpreter to execute one instruction at a time so that it can be interrupted after any instruction. Otherwise it could be difficult deciding where to interrupt a complex conditional or iterative construct. The parsed procedures are stored in the *procedure library*, to be accessed by the *procedural search engine*.

4.1.3. The procedural search engine

The search engine generates a space of disjunctive procedure instances by interpreting the parsed procedures. This process produces an or-tree [33], where nodes correspond to computational processes and arcs correspond to choice point selections. When the search engine interprets a choice point statement, the node splits into children that represent alternative continuations of the parent node. Since nodes correspond to computational continuations, the search process becomes an effort to schedule competing continuations. The leaf nodes are called *open* because only they are eligible for scheduling. Interior nodes are called *split*, and they cannot be scheduled for expansion.

The search tree is stored in a structure called the *search record* which is the primary output of the search engine. This record contains a pointer to the *root node*, a list of *open nodes* (leaves), a list of *success nodes*, a list of *failed nodes*, and a list of *split nodes*. The PROPEL data structures are defined as:

Search-Record → ((**root** ⟨node⟩) (**open** ⟨node⟩*)
(**success** ⟨node⟩*) (**failed** ⟨node⟩*) (**split** ⟨node⟩*))

Node → (⟨name⟩ ⟨control-stack⟩ ⟨global-vars⟩ ⟨parent-node⟩(⟨child-node⟩*))
Control-stack → (⟨frame⟩+)
Frame → (**frame** ⟨procedure-name⟩ ⟨program-counter⟩ ⟨bindings⟩)
Program-counter → ⟨integer⟩
Global-vars → ⟨bindings⟩
Bindings → ((⟨symbol⟩ . ⟨value⟩)*)

Each node in the search space corresponds to a computational process with a control stack. Each stack consists of a list of frames. Each frame contains a procedure-name, a

program counter and a set of local variable bindings. A stack frame's program counter indicates the next instruction to be executed. An example of a node's control stack can be seen below:

```
((FRAME :PROCEDURE-NAME GO-TO-CELL :PROGRAM-COUNTER 2
        :BINDINGS ((*VALUE* . W) (MOVE-DIR 'S)
                   (AGENT-LOC '(0 2)) (GOAL-LOC '(0 1))))
 (FRAME :PROCEDURE-NAME DELIVER-TILE :PROGRAM-COUNTER 3
        :BINDINGS ((AGENT-DESTINATION '(0 1))
                   (AGENT-DIR 'W) (DESTINATION '(1 1))))
 (FRAME :PROCEDURE-NAME MOVE-TILE :PROGRAM-COUNTER 2
        :BINDINGS ((TILE '*) (DESTINATION '(1 1)))))
```

This control stack indicates that the top-level procedure, `move-tile`, was invoked to move the tile named "*" to cell (1 1). `Move-tile` called subroutine `deliver-tile`, which called subroutine `go-to-cell`, which is ready to execute step 2. According to the bindings for that top frame, the agent is at location (0 2), it has just moved south, and is about to move west to the goal location (0 1). The special binding for `*value*` indicates the most recent choice point selection. This is discussed further in the next section.

In addition to a control stack, each node also contains a set of *global variables* that are accessible to all procedures on the stack. Note that these variables are only global for a given node, and are thus not the same as standard "global" variables in LISP. When a *root node* is created, the global variables are initialized by evaluating their associated initialization forms. In our example, the `*state*` variable presented in Section 3.1 is such a global variable.

The search algorithm

The search engine functions as a scheduler that uses heuristics to bias the amount of CPU time allotted to competing continuations. Local and global heuristics are both used to bias the search. The search algorithm that schedules the CPU time for search nodes is presented below.

```
(defun node-scheduler (search-record)
   (loop until (terminate? search-record)
      do (loop for node in (best-nodes search-record)
            do (expand-node node search-record))))
```

The `node-scheduler` algorithm iteratively selects a subset of nodes for expansion until the termination criterion is satisfied. *Expanding* a node causes that node's next instruction to be executed, and thus corresponds to giving that node a quantum of CPU time. The function `best-nodes` selects a preferred subset of nodes according to a combination of global and local heuristics. The default definition of `best-nodes` provides depth-first search by simply returning the first open node. However, the functions `best-nodes` and `terminate?` are typically programmed by the designer to use application-specific knowledge heuristics. For our Tileworld example, these functions are defined as fol-

lows:

```
(defun terminate? (search-record)
   (or (success-nodes search-record)
       (null (open-nodes search-record))))

(defun best-nodes (search-record)
   (list (first (sort (open-nodes search-record) 'rank<))))
```

The function `terminate?` returns TRUE when a solution has been found or there are no more open nodes to expand. The function `best-nodes` serves as a *global* heuristic, and can use any application-specific method to select a subset of the open nodes. For this example, we defined `best-nodes` to sort the open nodes in increasing rank and then return the node with the lowest ranking. The rank is determined by a local heuristic function when each node is created. This is discussed in more detail later in this section. Most search strategies such as depth-first search or A* [33] can be implemented by defining `best-nodes` appropriately. If we define `terminate?` to use a search depth limit, and we define `best-nodes` to perform A* search, the result would be similar to RTA* [24], but in the context of PROPEL's action representation and search space.

Expanding nodes

Most of the `node-scheduler`'s work is performed by the `expand-node` function. Using the process scheduling metaphor, expanding a node is analogous to giving a process some CPU time. `Expand-node` executes the next instruction of the top frame on a given node's control stack. Instructions are executed by consulting a table of *handlers* for each type of PROPEL expression. The handler for assignment statements pushes new bindings on the current frame's binding list, and the handler for subroutine calls pushes a new frame onto the stack. The program counter is incremented after executing each instruction, except for `goto` statements which cause the program counter to jump to a nonsequential step number. Conditional `goto` statements are handled by updating the program counter based on testing the condition. Whenever the program counter is equal to the length of the procedure, the procedure is considered complete, and the frame is popped off of the stack. The search engine can be interrupted after each cycle of node expansion.

The functional programming feature of "returned values" is achieved by pushing the value of a procedure's last expression onto the bindings of the next frame using the special binding `*value*` as in `'(*value* . 3)`. An example of this was seen in the sample control stack presented earlier in this section. Whenever a node is expanded, it is first checked for a returned value, and if found, the value is syntactically substituted for the subroutine call in the current instruction. After replacing the procedure invocation with its returned value, the instruction is handled as a simple expression.

Choice point processing

The handler for expanding `choose-value` and `choose-procedure` statements is the central component of the search engine because it is responsible for generating the

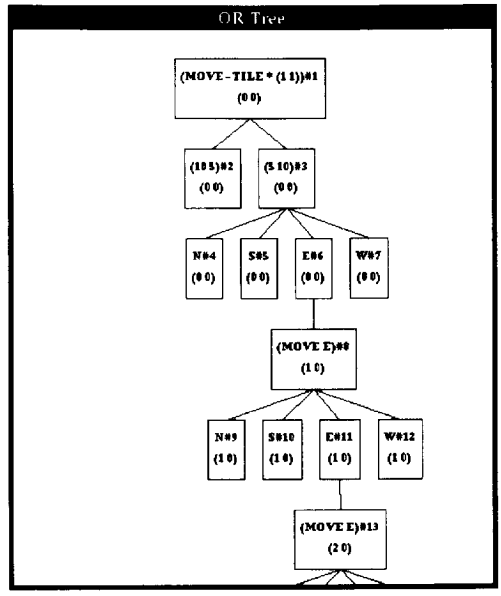

Fig. 5. The search tree for our example.

search space. Expanding a node whose next instruction is a choose statement causes that node to *split* into a set of new nodes which serve as unique continuations for each choice in the choice point. The new nodes are linked as children to the current node, creating branches in the underlying search tree. The new nodes may be annotated by an optional local heuristic function before they are pushed onto the open node list. After splitting, the parent node is removed from the open nodes list and moved to the split nodes list of the search record. An example of the search tree generated for our running example can be seen in Fig. 5.

The root node in Fig. 5 corresponds to a top-level procedure invocation of the form: `(move-tile '* '(1 1))`. This invokes the procedures that were presented in Section 3.1. The first choice point is encountered by the procedure `go-thru-door`, which must choose which of two doors to use as an exit; either the door at (10 5) or the door at (5 10). This causes the root node to split into continuations for each of those choices. Each node is named after the unique choice associated with that node's continuation. For this example, we use best-first search with a beam-width of one. Thus, a single node is selected for expansion by the node scheduler. That node, labeled "(5 10)#3", is repeatedly expanded by the node scheduler. When that node executes the instruction `(go-to-cell '(5 10))`, a frame for the `go-to-cell` procedure is created and pushed onto the node's control stack.

As discussed in Section 4.1.1, the `go-to-cell` procedure is where most of the search occurs in this application. It iteratively chooses a direction and then an action. When `go-to-cell` executes the statement

(move-dir ← (**choose-value** '(N S E W) **:heuristic** (closest-dir ...))),

node (5 10)#3 splits into continuations for each possible move direction. The split parent is then removed from the open node list. Our heuristic suggests moving east, so node E#6 is selected as the continuation to be expanded next. After selecting EAST as the move *direction*, `go-to-cell` selects an *action* by executing the statement

(**choose-procedure** (take a step in move-dir goal-loc)).

The goal pattern for this choose statement matches both the `move` and the `carry` procedures. When the agent is not holding anything, the preconditions for `move` will be satisfied, otherwise the preconditions for `carry` will be satisfied. Since the preconditions for these procedures are mutually exclusive, only one child node is created. In Fig. 5, the agent is not grasping a tile, so `move` is selected, and the node labeled (MOVE E)#8 is created. After moving east, the agent's simulated location is at (1 0). `Go-to-cell` repeats this process of selecting a direction and an action, resulting in more split nodes, until the agent's simulated location equals (5 10).

When a node splits, continuations are created by copying the control stack and global variables from the splitting node to each child node. Then, each child's stack is modified to reflect that continuation's unique choice point selection. This is achieved by pushing a different choice onto the local bindings of each continuation's top stack frame. The choice is stored as a value for the special variable `*value*`, and the program counter is not incremented. The next time this node is expanded, the choice will be substituted for the choice point using the same syntactic substitution method as described above for handling *returned values*. For example, each continuation from the choice point `(dir ← (choose-value '(n s e w)))` will have a unique choice substituted for the term on the right hand side of the arrow. After this substitution is made, the instruction will be treated as a simple assignment statement that does not contain a choice point. This can be viewed as a form of program transformation, where the choice point represents a class of values that are syntactically transformed into instances.

Local heuristics

The user can associate an application-specific local heuristic function with each choice point. The heuristics evaluate the choices and annotate the nodes based on local optimization criteria. These annotations can then be accessed by the `node-scheduler`'s `best-nodes` function to determine the nodes' global ranking. The local heuristics are optional, and the manner in which choices are annotated and used by `best-nodes` is entirely defined by the application designer.

Using a local heuristic is important in our application because of our problem reduction framework. We do not want to minimize a global distance function (like A* [33]) because we must first achieve the subgoals of getting to the doors, which conflicts with minimizing the global distance for the whole problem. The relationship between heuristic search and subgoal processing is discussed by Korf in [23].

The `closest-dir` heuristic shown below is used in our application to rank potential move directions in the `go-to-cell` procedure. Local heuristic functions accept explicit arguments determined by the programmer, and are also passed the list of new child nodes

to be evaluated and annotated with rankings. Our example heuristics use an application-specific global variable called `*rank*` to annotate each node. `Closest-dir` ranks the planner's choices as the best with a value of 0. However, we will defer the discussion of planner choices until we have defined the planner and the controller in Section 4.2. Without the planner's advice, directions that decrease the Manhattan distance between the agent location and the goal location receive a ranking of 1. Directions that increase that Manhattan distance receive a ranking of 2. Each node is annotated by storing its ranking on its global variable `*rank*`. This ordering is then accessed by the `best-nodes` function which applies global heuristics.

```
(defun closest-dir (agent-loc goal-loc choice-nodes
                                    &optional planner-choice)
   (let* ((agent-x (first agent-loc))
          (agent-y (second agent-loc))
          (goal-x (first goal-loc))
          (goal-y (second goal-loc)))
      (loop for choice-node in choice-nodes
         do (setq choice-value (get-choice-value choice-node))
            (set-global choice-node '*rank*
               (if (equal choice-value planner-choice) 0
                   (case choice-value
                         (w (if (> agent-x goal-x) 1 2))
                         (e (if (< agent-x goal-x) 1 2))
                         (s (if (> agent-y goal-y) 1 2))
                         (n (if (< agent-y goal-y) 1 2)))))))))
```

A local heuristic like this is an appropriate approach when sequential choice points do not interact. The local heuristics facilitate the development of non-monolithic objective functions. Instead of writing a single heuristic function that covers all choice points in the search space, the local heuristics allow the designer to supply a lot of smaller heuristic functions. This may provide localization benefits such as decreased coding complexity.

Success and failure nodes

An empty control stack indicates that the initial top-level procedure has been completed. Thus, when the last frame is popped off of a node's control stack, that node is considered a *success node*, and it is moved from the open nodes list to the success nodes list of the search record. Explicit FAIL statements can be used to prune bad procedure instances by removing them from the open nodes list. A procedure's preconditions provide an implicit FAIL statement. If any precondition cannot be satisfied, the current node is considered a *failure node* and it is moved from the open nodes list to the failure nodes list of the search record. This is how *backtracking* occurs: removing an open node will allow a new continuation to be scheduled. Node failures also occur when no matching procedures with satisfied preconditions can be found for a `choose-procedure` statement, or if a `choose-value` statement contains an empty set of choices. Also, a new

node is considered *redundant*, and removed from the open node list, if it has exactly the same control stack as a node that already exists.

Summary of PROPEL*'s search control techniques*

PROPEL uses four methods to control the exponential explosion of its search space. The most important factor in controlling search within PROPEL is the use of *procedure schemas*. Choice points only appear in isolated pockets within a PROPEL procedure definition. Thus, most of the controller's behavior is deterministic and no search is involved at all. The size of the search space is therefore only a function of the number of choice points, not the size or complexity of the program. The second most important search control technique in PROPEL is the use of both *local and global heuristics*. The value of heuristics in controlling plan search is described by Korf in [23]. The third form of search control is the use of *subgoaling*. One of our two forms of choice point, the `choose-procedure` statement, corresponds to a subgoal achievement statement. As noted in [23], the use of subgoals can drastically reduce the size of a search space. *Node failures* represent the final technique for controlling search in PROPEL. When a node fails due to the explicit use of a FAIL statement, or an empty set of choices, or unsatisfied preconditions, it is pruned from the search space.

4.2. Planning and control

This section describes the extensions required when PROPEL procedures use physical sensors and effectors. Only within the context of sensors and effectors can we distinguish between using the procedural search engine for planning and using it for control. First, remember that in our system *planning* means using predictive search to select effector commands. It requires mechanisms for simulating the effects of physical actions. *Control* means reading sensors and executing effector commands in bounded time (without search). When no planning time is available, the controller will execute a heuristically chosen default plan. However, if planning time *is* available, the planner can consider and evaluate other choices in order to advise the controller accordingly. The role of the planner is to pre-program the controller at run time to operate in unusual situations. At this second design layer we are still talking about *open-loop* planning and control, where a human interacts with the planner and the controller. *Closed-loop* planning and control systems will be the subject of Section 4.3.

Fig. 6 illustrates the PROPEL architecture after extending the core to handle sensors and effectors. The primary changes include using two copies of the procedural search engine, one for the planner and one for the controller. At *design time*, the user enters a set of application-specific control procedures. At *run time*, the user instructs the planner to project a given procedure for a given period of time. The planner performs its search based on run time sensor information that describes the current external state. The search process continues until either a solution is found, or the time limit is expired, or the search space is exhausted. The user then instructs the controller to execute the procedure using choice point advice that is extracted from the planner's search record.

We now describe each of these components in more detail, starting with the *sensors* that are used to receive input from the external environment. The raw sensor readings

pass through an application-specific *sensor interpreter* component that is outside the scope of this paper. The sensor interpretation can be as simple or as complex as required for the application. Its function is to convert the sensory input into whatever form is required by the control procedures. The results of sensor interpretation are stored in variables within the control procedures. As described in Section 3.1, our example application uses an application-specific global variable called `*state*` to record sensory information in the form of predicates such as `(agent-location '(2 5))` and `(temperature 25)`.

4.2.1. The planner

The planner is implemented using the procedural search engine described in Section 4.1.3, after extending it to *simulate sensing and effecting*. The planner is invoked by a calling the function `Plan!`, defined below, with either a procedure invocation such as `(move-tile '* '(1 1))`, or a previously existing search record as an argument. The search record option will be used to pass execution failure information from the controller to the planner. This is described further in Section 4.3 where closed-loop planning and control is discussed. If a time limit is provided, the planner will terminate after that amount of time has elapsed. The function `Plan!` performs the same node-scheduling activity that was discussed in Section 4.1.3. After setting up the search record, `Plan!` iteratively selects and expands a subset of continuations until the termination criterion is satisfied. The programmer customizes the functions `planner-best-nodes` and `planner-terminate?` to suit their application.

```
(defun Plan! (&key invocation search-record time-limit)
   (if invocation
       (setq search-record (init-search-record invocation)))
   (if time-limit (set-deadline search-record time-limit))

   (loop until (planner-terminate? search-record)
      do (loop for node in (planner-best-nodes search-record)
            do (expand-node node search-record 'plan)))

   (cond((search-record-success search-record)
         (print-msg ''Search Success!''))
        ((search-record-open search-record)
         (print-msg ''Time Limit Expired!''))
        (t (print-msg ''Search Failure: No Open Nodes!'')))
   search-record)
```

Simulating actions

We do not want the planner to cause changes in the external environment. Since effector commands produce changes in the environment, we cannot actually execute them during the planner's search process. Therefore effector commands must be simulated. To facilitate this, procedures can be provided with a *simulation description* that will only be interpreted by the planner. If a simulation description is provided for a procedure, the planner will always interpret the simulation instead of the *body*. If no

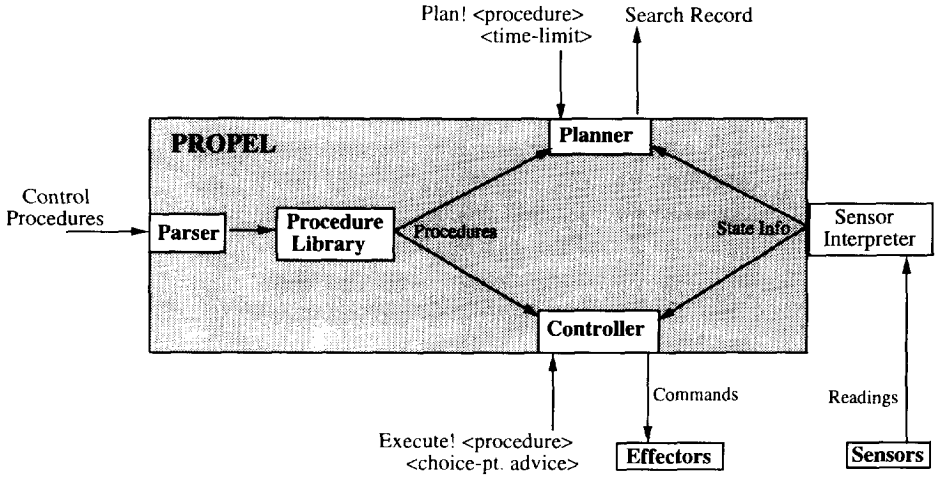

Fig. 6. Using sensors and effectors.

simulation description is provided, the planner interprets the body. On the other hand, the controller will always execute the body, which will activate the effectors. The `move` procedure shown below has a simulation description which defines how to simulate the `move-agent` effector command.

```
(Defprocedure move (?dir)
   :Goal
      (take a step in ?dir ?goal-loc)
   :Preconditions
      (not (grasping-object? *state*))
      (agent-loc ← (get-agent-loc *state*))
      (target-loc ← (adjacent-cell agent-loc ?dir))
      (cell-empty? target-loc *state*)
      (in-bounds? target-loc)
   :Body
      (move-agent ?dir)
      (wait (expected-action-duration 'move))
      (*state* ← (read-sensors))
   :Simulation
      (*state* ← (remove-fact '(at agent-loc) *state*))
      (*state* ← (add-fact '(at target-loc) *state*)))
```

To simulate moving in our example application, we explicitly add and delete facts that simulate the physical effects of the `move-agent` effector command. During planning, the simulation uses the LISP functions `remove-fact` and `add-fact` to maintain the global variable `*state*`. These statements function like the classical add and delete lists of STRIPS fame [14]. However, any application-specific method can be used for

modeling simulations. The add and delete list approach presented here is simply the method we've chosen for this example application. Since the simulation can contain nested loops, conditionals, arbitrary LISP function calls, and even choice points, we can simulate highly conditional operators and effects. It has traditionally been difficult to model highly conditional effects for planners that use analytical techniques such as TWEAK's Modal Truth Criterion [7].

Failure nodes occur when the preconditions fail for a deterministic subroutine call, or if there are no subroutines with enabled preconditions to match a choose-procedure call. In our example, this occurs when an obstacle causes a failure of the `(cell-empty? target-loc *state*)` precondition of the `move` procedure. When this happens, the node is simply removed from the list of open nodes. This is how *backtracking* occurs: removing an open node will allow a new continuation to be scheduled.

The primary product of the planner is a sequence of choices that correspond to the arcs along a path in the search tree from the root node to a leaf node. These choices are stored as IF ⟨*condition*⟩ THEN ⟨*action*⟩ rules, and used by the controller's local heuristic functions when it executes a choice point statement. The rules say "IF ⟨the control stack looks like x⟩ THEN ⟨select choice y⟩". We call these rules Situated Control Rules (SCRs) after the terminology of Drummond [11]. The production and use of SCRs is described further in Section 4.2.2, where we discuss the integration between the planner and controller.

Plan representation

Our *plan representation* is tuple of the form ⟨Procedure-name, SCRs⟩. Thus a *plan* is defined as a top-level procedure call and a set of choice selection rules for choice points that may be encountered during the execution of the top-level procedure and its subroutines. The SCRs set may be empty, or it may only provide coverage for some choice points.

Anytime planning properties

As discussed in the Section 1.1, Dean and Boddy have identified a class of algorithms called *anytime algorithms* that are useful for meeting the demands of time-dependent planning [9]. They define anytime algorithms to be procedures that are interruptable and ready to provide increasingly useful results at any time [2,9]. Further, they identified heuristic search methods as being likely candidates for use as anytime algorithms. PROPEL combines heuristic search with anytime *interruptability* for algorithms written in a general programming language. The planner's partial results can be used when it is interrupted at any time. Since the procedures are simulated by executing each instruction in chronological order, all partial results start at the beginning of the plan. Thus, the planner can be interrupted after any instruction and the partial planning results will provide coverage from the *beginning* of the plan, where the controller needs it first.

Since the search engine can be interrupted after each node expansion, the grain size of interruptability is the amount of time required to expand a node. Most node expansions take only a small fraction of a second to evaluate conditions, or push a new frame onto

the control stack, or update the program counter, or process a choice point. However, an instruction may also be an arbitrary LISP function. Thus, lengthy computations should be written as PROPEL procedures rather than LISP in order to achieve a required grainsize of real-time interruptability.

An important property of anytime algorithms is that their results are expected to improve as computation time increases. This is not a guaranteed property of PROPEL applications because the planner is not inherently "anytime". However, PROPEL's inherent *interruptability* and general programming constructs facilitate the study and development of anytime algorithms. Any particular program written in this language may or may not provide incremental improvement depending on interactions between the heuristics and the environment. The value of partial plans produced by PROPEL will be similar to that produced by RTA* [24], or Reaction-First Search [13] depending on the heuristics that are used. Understanding how to generate monotonically improving partial plans will require much further study.

Instead of using a time limit as a termination test, anytime algorithms are typically described as being halted by some asynchronous process. We have used the time limit approach here to simplify the presentation by avoiding the need to present a method of asynchronous communication. The planner presented in this paper accepts a time limit which is used only in the termination test, as shown in Appendix B. We have simply defined the `planner-terminate?` function to terminate when the deadline expires. The time limit is not used as a basis for procedure selection as in the case of "contract" algorithms [35], where different decision procedures would be selected depending on the amount of available computation time. We have also implemented a version of `planner-terminate?` that tests the status of an asynchronous flag. Whether the planner uses a time limit to terminate or it receives an asynchronous halt message, the `node-scheduler` is interrupted at an arbitrary point in time.

4.2.2. *The controller*

The controller always executes procedures in bounded-time because it never uses search. The controller is implemented using a procedural search engine identical to that of the planner, but its `best-nodes` and `terminate?` functions are modified to prevent backtracking. Thus the controller always uses best-first search with a beam-width of one, and any *node failure* will cause the controller to terminate instead of automatically backtracking.

To prevent backtracking, we modify the controller's choice point processing as follows: Although the controller only uses the single (best) open node, it goes through the same node splitting behavior as the planner. However, when a choice point is expanded by the *controller*, the previously open nodes are moved to a special list in the search record called *closed nodes*, before storing the new nodes as the *only* open nodes. This prevents the controller from automatically backtracking after node failures. The closed nodes can be resurrected for later use by the planner in cases of execution failures.

```
(defun Execute! (&key invocation search-record scrs)
   (if invocation
       (setq search-record (init-search-record invocation)))
   (if scrs (set-scrs search-record scrs))

   (loop until (controller-terminate? search-record)
      do (expand-node (controller-best-nodes search-record)
                      search-record 'execute))

   (if (search-record-success search-record)
       (print-msg ''Execution Complete!'')
       (print-msg ''Execution Failure!''))
   search-record)
```

Our controller is invoked by a call to the function `Execute!`, which is shown above. It accepts an invocation such as `(move-tile '* '(1 1))`, or a search record as an argument, just like the planner. The controller also accepts choice point advice in the form of SCRs. If SCRs are provided, they will be used by the local heuristic functions to annotate the continuations produced when splitting a choice point.

The function `Execute!` shares a nearly identical structure with the planner, `Plan!`, defined in the previous section. It too operates as a node scheduler, using its own versions of the termination test and best-nodes function. As with the planner, the `controller-terminate?` and `controller-best-nodes` functions are application-dependent. Although the planner and controller *can* use different versions of the `best-nodes` function, it is not always necessary as long as they both use a beam-width of one. For our Tileworld application, the definition of `controller-best-nodes` is identical to that of `planner-best-nodes` (see Appendix B). However, the `controller-terminate?` function *must* be different from `planner-terminate?` so that the controller terminates after any node failure.

Since the controller is connected to the real world, it must read physical sensors and execute effector commands instead of simulating them. This occurs by executing the *body* of a procedure such as `move` which was presented in the previous section. The controller always executes the procedure's body, unlike the planner which uses a procedure's simulation when it is provided. In our example, the body of the `move` procedure shown in the previous section first issues an effector command that moves the agent. It then waits for the expected duration of the action before updating the global variable `*state*` by reading the sensors. In our application, all actions have a 0.5 second duration. This example shows that the programmer must explicitly update the sensor readings.

An interesting side benefit of using the same search engine for planning and control is that the redundant node checker used during search can effectively detect execution loops whenever a node is created with a control stack identical to an existing node. In some applications, it is appropriate to treat redundant execution nodes as failures, while in other applications it is not. Thus this feature can be switched on or off by the designer.

Integrating advice from the planner

The controller accepts optional choice point advice from the planner in the form of Situated Control Rules (SCRs) [11] that map a given control-stack to a choice point selection. When the controller encounters a choice point, it looks for an SCR with a left hand side that matches the current control stack. If a match is found, the SCR's right hand side is passed to the local heuristic function as described in Section 4.1.3.

An example SCR can be seen below. This SCR advises the controller to select the choice (MOVE 'E) when the current control stack matches the left-hand side of this rule. We can see from the flattened version of `go-to-cell` in Section 4.1.2 that step 2 is a choose-procedure statement. That statement is the choice point where this SCR will be used.

IF the stack equals:

```
(FRAME :PROCEDURE-NAME  GO-TO-CELL :PROGRAM-COUNTER 2
       :BINDINGS ((MOVE-DIR 'E) (AGENT-LOC '(0 0))
                  (GOAL-LOC '(5 9))))
(FRAME :PROCEDURE-NAME GO-THRU-DOOR :PROGRAM-COUNTER 2
       :BINDINGS ((DOORSTEP-LOC '(5 9)) (DOOR-LOC '(5 10))
                  (DIRECTION 'EXIT) (ROOM 'ROOM-A)))
(FRAME :PROCEDURE-NAME GO-TO-ROOM :PROGRAM-COUNTER 5
       :BINDINGS((AGENT-ROOM 'ROOM-A)
                 (DESTINATION-ROOM 'ROOM-B)))
(FRAME :PROCEDURE-NAME PICKUP-TILE :PROGRAM-COUNTER 1
       :BINDINGS ((TILE-LOC '(24 24)) (TILE '*)))
(FRAME :PROCEDURE-NAME MOVE-TILE :PROGRAM-COUNTER 1
       :BINDINGS ((TILE '*) (DESTINATION '(1 1))))
```

THEN take action (MOVE 'E)

SCRs are collected for a given search record by calling the `collect-scrs` function, shown below, with a search-record as its argument. This function starts with the first *success* node if one exists. If the planner is halted before a success node has been found, then SCR collection begins with the *best* open node. The control stack and selected choice for each node on the path from the leaf to the root is then collected.

```
(defun collect-scrs (search-record)
   (let ((node (or (first (success-nodes search-record))
                   (first (planner-best-nodes search-record))))
         parent scrs)
        (loop while (setq parent (get-node-parent node))
           do (push (get-scr node) scrs)
              (setq node parent))
        scrs))
```

Concurrent procedures

Since effectors usually operate in parallel, practical control systems often require the use of *concurrent* control programs. The management of concurrent control processes can be very difficult due to complex timing constraints and interactions. Although PROPEL does not provide any general solution to this problem, it does support the use of *concurrent procedures*. This is achieved by modifying the search node data structure so that each node contains one or more *threads*, each of which contains a separate control stack. Now, whenever a node is expanded, the next instruction in each of its threads is executed. This requires that we extend PROPEL's grammar with three new expressions. The *run-process* statement will start a concurrent procedure by creating a new thread for the current node. The *wait-until* statement will prevent a thread from being expanded until a given condition becomes true. The *sleep* statement is like wait-until except it waits for a period of time to elapse. Concurrent procedures also facilitate the planner's *simulation* of continuous and servo processes.

4.3. Closed-loop planning and control

The techniques described in the previous sections were application independent. However, defining a relationship between planning and execution is largely *application-specific*. For example, physical backtracking may or may not be possible, and deadlines will vary. In this section, we describe how to build an autonomous link between the planner and the controller that were presented in Section 4.2.

For *autonomous* systems, human control of when to plan and when to act is not possible. Thus we need an *executive* that determines how to distribute the total available time between planning and execution. Issues that must be handled by the executive include deadline management, execution failures, and a changing environment.

Fig. 7 shows a generic architecture for an executive that coordinates the planner and the controller described in Section 4.2. The user enters a procedure name and a deadline. The executive then determines how much time to spend planning using a method provided by the designer. We will return to this issue later. The executive invokes the planner described in Section 4.2.1, called `Plan!`, for the allotted time. When the planner terminates, it returns its search record to the executive. The executive collects SCRs from the planner's search record and then invokes the controller using the `Execute!` function described in Section 4.2.2, passing in the SCRs. When the controller terminates, it returns its own search record to the executive. In addition to SCRs, a *concurrent* executive could extract plan duration predictions from the evolving search record in order to help it decide when to stop planning and start acting.

The executive detects *execution failures*, such as failed preconditions, by inspecting the failed nodes list of the search record returned by the controller. When an execution failure occurs, the executive can pass a copy of the controller's search record to the planner. This record describes the tree of nodes generated during execution. Although the search tree was generated by the controller, the planner can backtrack through the tree to generate a set of SCRs that can be used by the controller to recover from the failure. This tight integration between planning and control is made possible because the two components share the same data structures and interpreter.

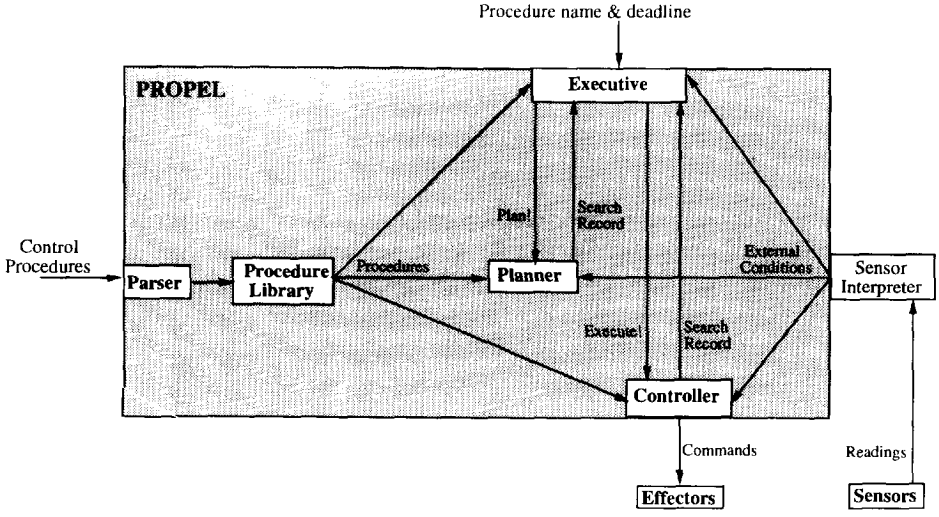

Fig. 7. An autonomous executive.

The relationship between the planner and the controller will depend on many application factors. For instance, physical backtracking may or may not be possible in any given application. Thus, the executive must be programmed by the designer. A desire to experiment with different executive models for complex closed-loop control programs was a primary motivation behind the design of PROPEL.

Depending on the application, several different styles of executive can be developed. This simplest is one-shot planning followed by execution. A more advanced approach is to interleave planning and execution. Finally, the most sophisticated approach is to have the planner and controller run concurrently. For our Tileworld example, we have performed experiments with both the one-shot and the interleaved approaches. The results from these experiments are presented in Section 5. First we describe how the executives work.

4.3.1. One-shot planning then execution

In the one-shot planning executive, the agent plans for some initial period of time before executing. If the planner has enough time to generate a complete plan, the SCRs are collected and passed to the controller. The SCRs will guide the controller around deadends that would trap its heuristic default behavior. Even if the planner does not have time to develop a complete plan, it still may have figured out how to avoid some early deadends. Thus, the *partial plan* is collected as SCRs and passed to the controller. The quality of the partial plan depends on the heuristics used. The following function implements this simplest executive using the planner and controller described in Section 4.2.

```
(defun plan-then-execute (invocation planning-time-limit)
   (let ((planning-record
            (Plan! :invocation invocation
                   :time-limit planning-time-limit)))
         (Execute! :invocation invocation
                   :scrs (collect-scrs planning-record))))
```

The plan-then-execute executive accepts a PROPEL procedure invocation and a planning time limit. It calls the planner, Plan!, which searches until the time limit expires and then returns its search record. The executive then calls the controller, Execute!, passing in the SCRs that were collected from the planner's search record. This executive is very simple, but it worked well for our example application, as shown in the experiment results section.

4.3.2. Interleaved planning and execution

This second executive illustrates an *interleaved* planning and execution approach. With this approach, the agent begins executing before any planning is performed. When the agent gets trapped in a deadend, the executive calls the planner with a short time limit to generate an escape route.

This example illustrates how we exploit the unified planner and controller to handle execution failures. When an obstacle causes the move procedure's preconditions to fail, the controller terminates and returns its search record to the executive. The controller's search record is then passed to the planner, providing a history of choice points that were encountered during execution. The planner backtracks through the search tree that was originally generated by the controller, and then it generates new SCRs. After planning, the controller resumes execution with its original search record, but using the new SCRs. A simple interleaving executive that uses the same planner and controller as the one-shot executive is shown below.

```
(defun plan-on-demand (invocation)
   (let ((execution-record (Execute! :invocation invocation))
         impasse-record scrs)
        (loop until (success-nodes execution-record)
           do (setq impasse-record
                    (make-impasse-record execution-record))
              (Plan! :search-record impasse-record)
              (setq scrs (collect-scrs impasse-record))
              (physically-backtrack execution-record scrs)
              (Execute! :search-record execution-record
                        :scrs scrs))))
```

The plan-on-demand executive accepts a procedure invocation as an argument. It begins by immediately calling the controller, Execute!, to execute the procedure. When the controller terminates, it passes the search record back to the executive. If the search record has no success node, then a failure must have occurred. To plan an escape, the executive makes a copy of the execution record, called the impasse-record, for use

by the planner. The choices that were not selected by the controller are resurrected as potential backtracking nodes by moving them from the closed nodes list to the open nodes list of the `impasse-record`. The executive then passes the copied record to the planner. In this example, the `impasse-record` that is passed to the planner is assigned a short deadline so that the planner does not plan all the way to a solution. Instead, it plans for a few seconds and then returns control to the executive. The executive then collects the SCRs from the short planning process. The executive must then physically backtrack to a cell where the new SCRs provide coverage before resuming execution. The physical backtracking is accomplished by calling the one-shot `plan-then-execute` executive described in the previous section. After physically backtracking, the executive resumes execution by re-invoking the controller using the same search record with which it started. This time however, the controller uses the new SCRs to guide its decisions, and it starts with a node associated with the agent's location after physically backtracking. We have run some experiments with both the one-shot and the interleaved executives, and the results are presented in the Section 5.

4.3.3. Domain-independent executive functions

Although executives must perform many functions that are application-specific, some common functionality is required regardless of the application domain. Most executives must perform the following self-regulating functions: (1) decide how much time to spend planning, (2) inhibit irrelevant, ineffective, or interfering reactions, and (3) detect and correct errors that occur during plan execution. Our approach to these domain-independent executive functions is briefly described below.

In order to meet real-time deadlines, we are developing a method for deciding when to start executing an incomplete plan. Each procedure will be associated with an estimated execution duration based on prior experience. The executive uses the duration estimates to determine how much time must be set aside for execution. Any available time beyond the expected execution duration can be used by the planner. As plans evolve, the execution duration estimates will become more accurate. Thus, the executive will use evolving plans to update its execution duration estimates.

The need to achieve real-time deadlines may force a system to be very reactive. However, it could also be too reactive. Counteracting this reactive tendency is the need to *inhibit irrelevant*, *ineffective*, or *interfering reactions*. An executive must maintain this balance between reactive and predictive behavior. These executive functions are based on neuropsychological theories of human frontal lobe function, and are described further in [26,27]. The ability to replace inappropriate default reactions with deliberate plans provides the flexibility required for humans (and machines) to operate in novel situations.

In order to detect and correct execution errors, the executive requires a mechanism for incorporating asynchronous sensor reports into the planning process. To achieve this, we are developing a technique called *dynamic dependencies*. The executive will use dependencies to identify plan assumptions, and then it will monitor the external status of those assumptions. If an important plan assumption becomes false, the `planner-best-nodes` function will re-evaluate its selections to prefer nodes that rely on valid assumptions. The

most difficult issue is performing dependency analysis on our procedurally expressive action representation. Our basic approach is to combine Kambhampati and Hendler's work on plan modification [22] with Zabih et al.'s work on dependency analysis for general programming constructs [39].

We are also interested in another style of executive called a *program patcher*. An autonomous system must function intelligently even when faced with a *gap* (i.e. a choice point) in its control program. At choice points, more details must be selected before a *fully-specified* command can be executed by the controller. In such cases it is necessary to synthesize a control program *patch* by combining lower-level control options at run time. With this executive, the planner is called whenever the controller hits a choice point. This differs from the interleaved planning executive described above because that executive requires an execution failure to occur before planning is triggered. Here, the planner is called before the controller makes a default choice. In other words, the planner is called preemptively as an error handler before the controller takes a potentially harmful action. This is similar to the way Soar uses impasses to trigger planning [25].

5. Experiment results

We have performed two experiments aimed at testing our hypothesis that planning can extend the operating range of a real-time controller. A full empirical study of the tradeoffs between between search and pre-programming, and the tradeoffs between alternative executive models is beyond the scope of this paper. Thus, the experiments are intended primarily to illustrate PROPEL's behavior and performance. Both experiments are based on the pickup-and-delivery example used throughout this paper. The experiments require the agent to deliver the tile as fast as possible in five problems of increasing difficulty. The five problems faced by our agent in both of the experiments are shown in Figs. 8 and 9.

Problem 1 is the routine situation for which the controller was designed. There are no obstacles so the agent can follow its heuristic default behavior to pick up and deliver the tile. This default behavior is indicated by the numbered steps in the figure for problem 1. That default path shows how the controller's heuristics prefer to move in directions that minimize Manhattan distances. When the agent must move both horizontally and vertically, the horizontal movements are selected first by the heuristic. In the remaining problems, the agent will encounter increasingly large obstacles that block movement along the default path.

As described so far, the agent's procedures do not allow it to get around any obstacles without planning. In other words, only problem 1 is within the operating range of the controller, and the agent will simply get stuck in deadends in problems 2–5. However, PROPEL's expressiveness allows us to represent *reactive* behavior, so we have added two new procedures, called `move-around-obstacle` and `carry-around-obstacle`. These procedures allow the agent to move around obstacles, escaping deadends by following a wall without using search. This reactive approach allows the controller to operate in situations with obstacles, but the resulting behavior involves a lot of physical

backtracking and is thus inefficient. The agent can use SCR plan advice when available, but without SCRs it can escape from deadends by reactively following walls. The reactive procedures for moving around obstacles are listed in Appendix B.

We want only the controller to use the wall-following procedures. For the planner, the obstacles should trigger computational backtracking rather than the physical backtracking produced by wall-following. So adding the reactive behavior makes it necessary to modify the `planner-best-nodes` function so that it never selects the wall-following procedures.

The plots in Fig. 10 show how long it takes the planner and the wall-following controller to solve each of the five problems independently. The vertical axis is elapsed real time in seconds, and the horizontal axis represents the problem numbers. The plot on the left shows how long the planner takes to generate a complete plan for each problem. The plot on the right shows how long it takes for the controller to physically pick up and deliver the tile using its reactive wall-following procedure alone, without any planning. Problem 5 is unsolvable by the pre-programmed wall-following controller because it gets caught in a loop due to a problem of limited look ahead. This is discussed further in the next section.

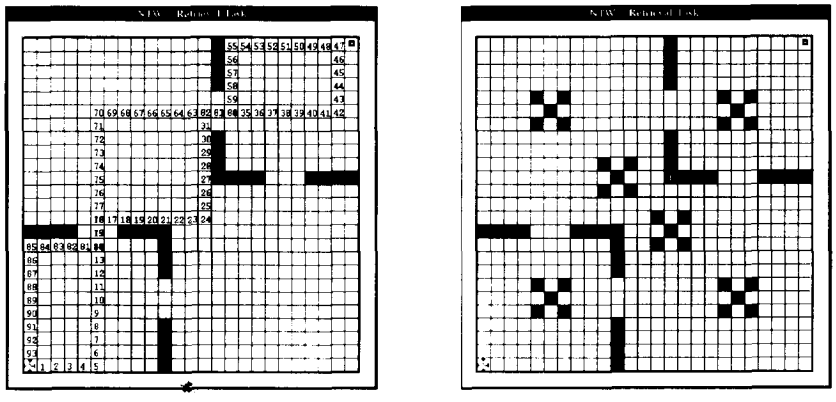

Fig. 8. Problems 1 and 2.

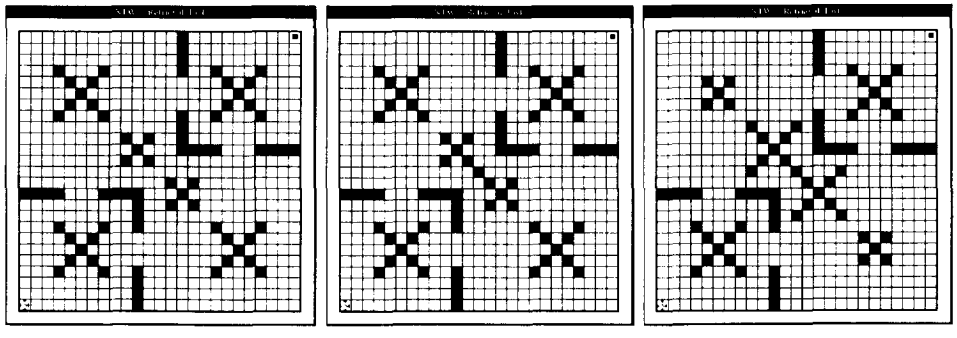

Fig. 9. Problems 3, 4 and 5.

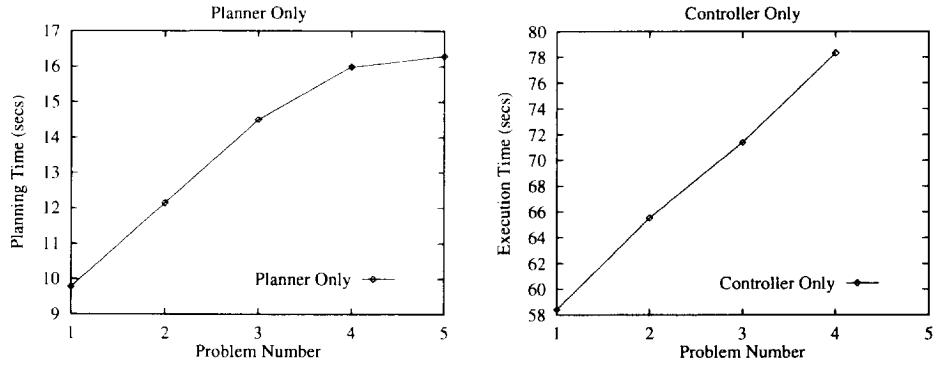

Fig. 10. Problem difficulty for the planner and controller.

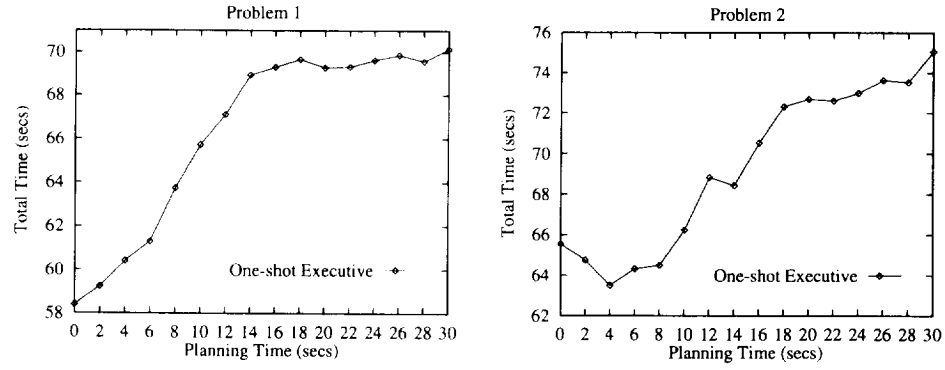

Fig. 11. Total time versus planning time (problems 1 and 2).

5.1. Experiment 1

This first experiment measures the total time taken by the agent to plan and execute the tile delivery. Thus, *total time* = *planning time* + *execution time*. Our hypothesis is that total time will decrease as planning time increases. For each problem, the total time was measured while varying the planning time from 0 to 30 seconds in increments of 2 seconds.

In this experiment, the agent uses the one-shot `plan-then-execute` executive described in Section 4.3.1. With that executive, the agent plans for some initial period of time. Then, when planning time runs out, SCRs are collected for the current best node, and passed to the controller. The controller then begins execution using those SCRs as choice point advice. When the controller runs out of SCRs, it completes execution by falling back on its pre-programmed behavior which uses wall-following procedures to escape from deadends.

Fig. 11 shows the results for problems 1 and 2. Problem 1 is simple enough that planning time only degrades the controller's performance. The total time flattens out at around 18 seconds of planning time because a complete plan is always found by that

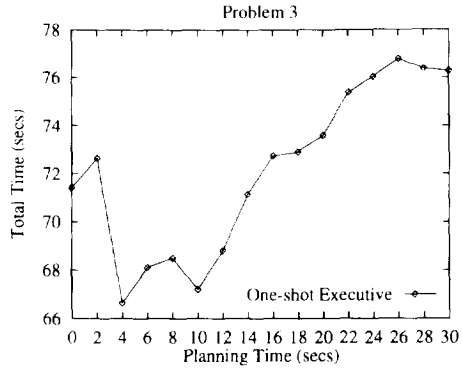

Fig. 12. Total time versus planning time (problem 3).

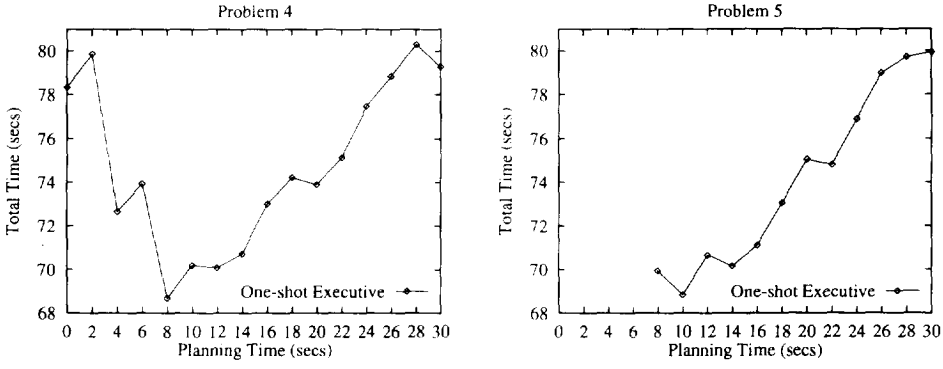

Fig. 13. Total time versus planning time (problems 4 and 5).

time. This causes the planner to terminate well in advance of its 30 second deadline. Problem 2 is only a little harder due to some small obstacles that do not take the wall-follower long to get around.

Fig. 12 shows that the planner starts to provide some value on problem 3. The deadends are large enough that without any planning the controller requires 13 seconds longer than when no obstacles were present. After 4 seconds of planning, the planner advises the controller how to avoid the first deadend. This reduced the required execution time by 9 seconds, yielding a net savings of 5 seconds of total time. Any more planning only increases total time compared to that optimal point.

Fig. 13 shows the results for problems 4 and 5. In problem 4, the agent's default path through the center cell is completely blocked. After four seconds of planning, the planner can advise the controller around the obstacle inside the first room. Eight seconds of planning reduces the execution time by 18 seconds. Thus, 8 seconds of planning yields a 10 second net savings in total time over the time required by the controller alone. Using only a partial plan, the agent takes less total time than if it had constructed a complete plan or if it had done no planning at all.

In problem 5, the deadend trap in the center is large enough that without any planning, the controller cannot solve the problem. Thus there are no data points until the planner uses about 8 seconds to plan around the deadend trap. The size of the deadend in problem 5 causes the wall-following procedure to get stuck in a loop since it relies only on one-step look-ahead to determine when it is out of the trap. This happens as follows: The agent is trying to move east into the cell at (13 11), but it is blocked. It then follows the left wall until it can move in the blocked direction (east) unless moving east would undo one of the detour steps. This worked well in the first four problems. However, in problem 5, the detour loop is large enough that the agent can move east without undoing a detour step while it is still trapped. If the reactive procedure looked ahead two steps, it would realize it had not escaped the deadend yet. This is a good illustration of how finite programmer effort and foresight can limit the operating range of a pre-programmed reactive controller. However, any reactive procedure must have limited look-ahead in order to meet real-time constraints. See the wall-following procedures in Appendix B for the details of the limited look-ahead behavior.

5.2. *Experiment 2*

Our goal for this second experiment was to compare the performances of the interleaved planning executive, the one-shot planning executive and the pre-programmed wall-following controller. Since a full study of the tradeoffs between the different controllers is beyond the scope of this paper, this experiment, like the first one, is intended primarily to illustrate the behavior of the three controllers presented in this paper.

When using the interleaved planning executive described in Section 4.3.2, we remove the wall-following procedures from the agent's procedure library. Thus, whenever the agent runs into an obstacle, an *execution failure* occurs, and the planner is called instead of immediately following a wall. With the interleaved executive, every time the controller ran into an obstacle, the planner was called for 5 seconds. The one-shot planning executive was allotted 10 seconds of planning time for each problem. Each executive would perform differently if we varied its allotted planning time.

Fig. 14 compares the performances of the one-shot planning executive, the interleaved planning executive and the pre-programmed wall-following controller. The figure shows that the one-shot executive did the best on average, maintaining a steady performance of approximately 66 seconds (total time). We also see that the wall-following controller outperformed the interleaved executive on problems 2–4, but it could not solve problem 5 at all. The interleaved executive's inefficiency is due to its physical backtracking. The agent must backtrack to a cell for which the planner generated SCRs, but due to our heuristics, SCRs will only be generated for plans that move directly toward the goal. Thus, the agent sometimes physically backtracks further with this executive than it does with the controller's wall-following routine. There is also a certain amount of overhead associated with swapping between planning and control, which adds to the inefficiency of the interleaved executive. However, the interleaved executive still extends the controller's operating range because it can get the controller through problem 5, which the wall-following program could not solve at all.

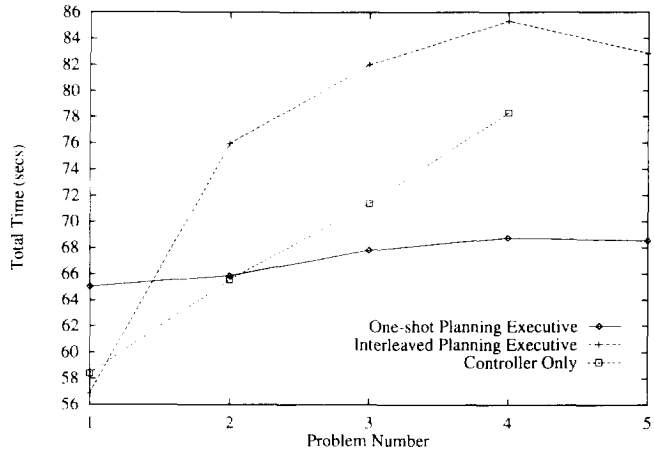

Fig. 14. Comparison of one-shot, interleaved, and no planning.

One advantage of the interleaved approach is that it uses only as much planning time as necessary. The interleaved executive never called the planner for problem 1, called the planner twice for problem 2, and called it three times for problems 4 and 5. Consequently, the interleaved executive outperformed the one-shot executive on problem 1. It also outperformed the wall-following controller for problem 1 because it never called the planner and it did not spend time testing the preconditions of the wall-following procedures.

5.3. *Analysis*

The above experiments support our hypothesis that planning can extend the operating range of a pre-programmed controller. By pruning pre-programmed (default) actions that led to deadends, planning allowed the controller to operate more effectively in situations that trapped the limited look-ahead of the reactive controller. In some cases, the sum of the planning time plus execution time was less than the time that was spent executing with no planning. This has been noted by McDermott as an important property for integrated planning and control systems [32].

While the experiments show that increased planning time can yield a decrease in total time, they also show that too much planning can increase the total time. This could effectively *narrow* the controller's operating range; an effect we'd like to avoid. Thus a good executive will need a fine understanding of just how much planning time is optimal.

The profiles of our agent's performance shown in Figs. 11–14 are particular to our application. The exact shape of the plots depends on interactions between the controller's default behavior, the planner's search heuristics, and the distribution of local minima (deadends) within the search space. Therefore we cannot make any general claims that all PROPEL applications will perform like ours. The above experiments were designed primarily to illustrate PROPEL's behavior for this paper. However, PROPEL's procedural

expressiveness, the interruptability of its planner, and its unification with the controller are general-purpose tools that were designed for exploring and evaluating more general theories.

6. Related work

PROPEL is related to a variety of previous research which can be roughly classified as *procedural search* and *situated planning* systems. In this section, we compare our system to examples from each of these categories.

6.1. Procedural search

The core of our system is the procedural search engine that generates a search space for a set of nondeterministic LISP-like procedures. The most related system to our procedural search engine is Siskin and McAllester's Screamer system [37], which is a dialect of LISP. Screamer includes nondeterministic assignment statements like `choose-value`, but there is no `choose-procedure` functionality. Screamer, a descendant of Chapman's Dependency-Directed LISP [7,39], is very similar to our core, the procedural search engine. However, a major difference is PROPEL's use of heuristic search compared to Screamer's reliance on chronological backtracking. Also, their system has no special facilities designed to support situated planning like the integration of PROPEL's interruptable planner, real-time controller and SCRs. Another difference is that Screamer is a true extension of LISP and is compilable, while PROPEL is an extension to a subset of LISP, and it is interpreted.

Another similar system that performs search with general program procedures is McDermott's Reactive Plan Language (RPL) [1,31,32]. RPL is a descendant of Firby's RAPs system [15], which is primarily a language for writing reactive programs that can physically react to unexpected events and situations without using predictive search. RPL provides many reactive control constructs that are not present in PROPEL, but RPL programs are deterministic because they do not explicitly model choice points for selecting subroutines and variable bindings. The RPL projector generates alternative timelines, but it is not really a planner since it does not modify any action representation. However, the RPL projector is part of a situated transformational planning system called XFRM [32] which is discussed further in the section on related situated planners.

The Procedural Reasoning System (PRS) [17], like PROPEL, was motivated by the desire to represent complex procedures rather than procedurally inexpressive STRIPS-like actions. However, PRS' action representation requires that all behavior be encoded as graphical networks of condition–action rules. Thus, PRS does not allow control system programmers to use familiar programming constructs for iterative and conditional control or local variable assignment. Additionally, using our definitions of planning as predictive search and execution as pre-programmed control, PRS is strictly an executor—not a planner [21]. PRS always executes its procedures, and thus is a purely *reactive* system. PRS has no special facilities for searching and backtracking through a space of procedure *simulations*. It also has nothing like our SCRs for integrating predictive search with

real-time control. Other reactive systems like GAPPS [21] also fit this description. These systems are pre-programmed real-time controllers that focus on providing nearly instantaneous action responses for any given sensory state.

PROPEL's heuristic search through a space of procedure instances is similar to the method of "skeletal plan instantiation" used by Friedland's MOLGEN system [16], with our procedures corresponding to MOLGEN's skeletal plans. Although our representation is more procedurally expressive, both systems encode partially specified procedures. In contrast with MOLGEN, our system performs closed-loop control by actually executing the procedures it selects. One assumption shared by both MOLGEN and PROPEL is that you do not need to plan from scratch because procedure schemas can be designed in advance. This is also similar to a key assumption behind the Case-Based Reasoning work of Hammond [18].

Wilkins' SIPE system [38] is well known as a practical planner; a quality we would like to achieve. SIPE is able to search through procedures in the sense of many other hierarchical, non-linear planners such as O-plan [8] and HTN [22]. These systems decompose high-level operators into more primitive ones by using a form of backward chaining [33]. Although these systems capture notions of procedural decomposition and modularity, they typically rely on unconventional, highly-specialized action representations that make it difficult to represent arbitrarily complex actions. Additionally, the planner's language is typically different from the general-purpose programming language used to implement the real-time controller. This language barrier between the planner and the controller has traditionally made it difficult to replan after execution failures. Additionally, the backward chaining nature of these systems has made it difficult to achieve anytime interruptability properties.

Finally, we should mention that the basic idea of combining search with a general programming language is addressed by PROLOG [6]. PROLOG is a general programming language designed to perform search. However, the syntax is still very unconventional because it is foreign to most control system programmers. Although most procedural constructs can be built out of PROLOG's backward chaining mechanism, it is not as straightforward as a conventional language. PROPEL also differs from PROLOG in its support for situated planning (i.e. incremental search while connected to sensors and effectors). In contrast with PROLOG, PROPEL was specifically designed to provide integrated planning and control using SCRs. We can also contrast PROPEL's heuristic search facilities with PROLOG's tendency toward chronological backtracking strategies.

6.2. *Situated planning*

The relation between the planning and control components in PROPEL is modeled after the Entropy Reduction Engine (ERE) [4, 12]. We chose the ERE approach because it has the benefit that the controller operates independently from the planner so that real-time control is not dependent on the more expensive search behavior of the planner. Both PROPEL and ERE use a planner that passes Situated Control Rules (SCRs) [11] as advice to an independent controller. Using SCRs, both systems can guarantee that the controller selects actions in bounded time since it is not dependent on the search process. ERE and PROPEL differ primarily in their action representations and their search spaces.

In particular, PROPEL uses procedures and searches through a space of computational continuations, while ERE uses a more traditional state-space search approach. Also, the PROPEL planner serves the roles of both the reductor and projector components in the ERE system. Although both systems use SCRs, the rules themselves are different due to the different search spaces. The left-hand side of PROPEL's SCRs refer to control stacks, while ERE's SCRs refer to state-space conditions.

McDermott's XFRM system [31,32] is similar to PROPEL in its ability to plan with an action representation that is procedurally expressive enough to be used as a real-time control programming language. XFRM uses RPL (described above) to represent behavior. In XFRM, RPL procedures can be simulated by a projector and then modified by a planner before being physically executed by a real-time controller. Like PROPEL, XFRM represents a unified planning and control architecture because the same RPL interpreter is used by both the planner and the controller. However, XFRM and PROPEL represent complementary approaches to unified planning and control. PROPEL's *controller* was designed to use the *planner's* procedure interpreter, while XFRM's *planner* was designed to use the *controller's* procedure interpreter. Thus XFRM and PROPEL started at opposite ends of the predictive/reactive spectrum, but are moving towards each other. Another difference is that XFRM's planner searches through a space of program transformations, while PROPEL's planner searches through a space of computational continuations. Also, XFRM's projector is not the same as it's planner. The projector is a module that generates timelines by simulating RPL procedures. The planner's critics then evaluate the timelines and fix bugs by transforming the plan. In PROPEL however, planning and projecting are basically the same node scheduling process. Another distinction is PROPEL's direct support for interruptable planning, contrasted with XFRM, which does not provide plan results until the plan has been completely projected.

As described in the introduction, Dean and Boddy's work on anytime planning algorithms [2,9] forms a theoretical framework for evaluating and using anytime algorithms. They describe the basic properties of anytime algorithms, and focus on the use of performance profiles to perform deliberation scheduling. The graphs shown for experiment 1 are very similar to Dean and Boddy's performance profiles in [2]. We contrast their focus on decision theoretic deliberation scheduling [2] with our own focus on the development of a procedurally expressive search engine. PROPEL's expressive, nondeterministic, and interruptable procedures may allow many new behaviors to be encoded as anytime algorithms.

Bresina has reported on related work in [3] that combines problem reduction with an SCR-based controller. That work describes an impasse-triggered planner that is very similar to the interleaved planning executive described in Section 4.3.2. The focus of that work involves finding a minimal set of SCRs that guide the reactor around *critical choice points* only. The SCRs can also be used by a learning component that modifies the problem reduction rules to avoid future impasses. We contrast that focus with our own focus on integrating predictive search into a general programming language.

Laird and Rosenbloom's Robo-Soar [25] also contains an impassed-triggered planner. Their architecture consists of a production system that is connected to sensors and effectors. When more than one production rule matches the given sensory state, the

agent is said to have reached an *impasse*. In routine situations, a unique production rule is found that instructs the controller to take a unique action. In non-routine situations however, multiple enabled production rules provide a *choice* of possible actions. This is similar to our notion of a choice point. Soar attempts to solve an impasse as a "subgoal" by triggering a recursive call to the production system. When a subgoal is solved, they use a technique called *chunking* to modify the production rule set so that a unique production rule will match the same situation next time, so that an impasse will not occur. A major difference between PROPEL and Soar is our use of a general programming language to encode behavior in contrast with Soar's production rule representation. We believe that our expressiveness will allow us to represent more complex control programs. However, Soar's ability to learn generalized production rules may be an advantage over the matching cost incurred by PROPEL's use of specific SCRs.

Lyons and Hendriks [29] describe an architecture that is also based on a model where an incremental planner provides advice to an associated real-time controller. They have focused on developing a formal theory about how the planner can incrementally modify the controller's behavior towards some theoretical *ideal* behavior. As with Soar, a major difference between their system and PROPEL is the action representation. Since they have a theoretical emphasis, they use a representation that does not encode control behavior using standard programming constructs for iterative and complex conditional control, variable assignment and procedural decomposition.

We can also compare PROPEL's integrated planning and control approach with a system that combines the SIPE planning system [38] with the PRS control system [17]. In this combined system [38], the SIPE planner produces PRS procedures that are used by the controller. Although the planner produces PRS code, it uses SIPE's action representation and SIPE's action interpreter during the planning process. In other words, the planner and the controller use different languages to encode behavior. Thus this approach does not represent a unified approach to planning and control. PROPEL's unified approach facilitates smoother transitions from the controller to the planner in the face of execution failures because the execution program control stack can be processed by the planner to evaluate error recovery options.

Saffiotti et al. present "A multivalued logic approach to integrating planning and control" elsewhere in this same journal issue [36]. They show how multivalued logic can be used to define behavior schemas that are similar to potential field and fuzzy-rule controllers. They present formal theorems and proofs for blending together independent behavior schemas based on desirability functions that assign a numeric value to each potential action from a given state. They show how the behavior schemas can be integrated with both PRS-style deliberation and goal-regression style "pre-planning". They view plans as a collection of reactions, however, rather than as a behavior sequence such as a PROPEL procedure. They do not specifically discuss the use of run time predictive search which is at the core of PROPEL. It may actually be possible to incorporate some of their features into PROPEL by viewing their desirability functions as a formal definition for PROPEL's local heuristics. The goal patterns in our `choose-procedure` statements would correspond to the names of their behavior schemas, and our local choice point heuristics would be like their desirability functions. This may allow PROPEL's global

heuristic function to use their theorems for blending the local choice point heuristics. It would be interesting to pursue such a blend of these two approaches.

7. Qualitative evaluation

In this section we evaluate PROPEL in terms of its assumptions, limitations, and contributions. We begin by discussing the assumptions that affect PROPEL's utility for any given application, and then we discuss PROPEL's current limitations, suggesting directions for future work. We close by summarizing PROPEL's original contributions.

7.1. Assumptions

PROPEL's utility in any given application relies on some assumptions about that application which we will now describe. First, we assume that the control system designer can supply the various forms of *application-specific knowledge* that are summarized below.

Procedures and choice points

The control system designer must specify a set of application-specific procedures that define the controller's behavior. This is required even for entirely pre-programmed controllers. In a PROPEL application however, they must also specify a set of choice points within those control procedures. The ability to enumerate choices in advance is an assumption that is present in all planning systems since they all enumerate their operators. The difference for PROPEL is that the choices are embedded within a general programming language.

Our approach assumes that procedures don't need to be entirely synthesized from scratch, but they cannot be completely pre-programmed either. This is a reasonable assumption because the existence of many pre-programmed controllers represent a vast pool of control procedures. The boundaries of their operating ranges represent program gaps that could be instantiated at run time by a planner. A simple approach to identifying potential choice points in a standard control program is to identify its error conditions. Thus we feel that all control programs can be viewed as plan schemas that are not fully instantiated for unusual conditions.

Heuristic and termination functions

The designer must provide application-specific definitions for the `best-nodes` and `terminate?` functions. The global heuristic knowledge encoded within `best-nodes` is used to control the planner's search process, and also to provide default choice point advice for the controller. Optionally, the designer can associate a *local* heuristic function with each choice point. We believe that the ability to provide heuristics is a reasonable assumption because heuristics have been widely available and successfully used for many expert system applications, and they were trivial for our example application.

Action simulations

To prevent the planner from effecting changes in the environment, the designer must supply an application-specific method for *simulating* the effects of physical effector commands. In our example, we have shown a very simple STRIPS-based approach toward simulation. However, the designer can use any simulation technique they wish. Most planning systems hold the assumption that effector commands can be simulated. This assumption should not be any less valid for PROPEL which can simulate complex procedures using general programming constructs. Only the low-level procedures that execute effector commands require simulation descriptions. For our Tileworld example, only the `move`, `carry` and `grasp` procedures required simulation descriptions.

Executive models

To achieve closed-loop planning and control, the designer must develop an application-specific executive. Alternative executive models may depend on issues such as the cost and feasibility of physical backtracking and the severity of deadlines. Several executive models were described in Section 4.3.

Other assumptions

In addition to the above forms of application-specific knowledge, we assume that the designer will provide an application-specific set of sensors and effectors, and a sensor interpretation component that converts the sensory input into whatever form is required by the control procedures. We also assume that a useful amount of planning time is available. This is reasonable for many applications where required response times range from seconds to hours. In order for our planner to be truly *anytime*, we assume that the quality of the planner's partial plans will increase as planning time increases. All of these assumptions held for our Tileworld example, and we believe many other applications have these properties, including our NASA application. Currently, we also assume that the external world remains static during the search process, and that the agent has global perception. These two assumptions generally do not hold in real-world applications, so we will address them again in the section on current limitations and future work.

7.2. Limitations and future work

In this section, we discuss some of the current limitations of PROPEL which suggest directions for future work.

Reactivity issues

To truly be reactive, both the planner and controller must be able to operate in a *dynamic world* with *limited sensing*. In particular, two major issues that must be addressed are exogenous events and variant outcomes. *Exogenous events* are environmental changes, like destructive winds, that originate outside of the controller. *Variant outcomes* occur when the effects of a single action cannot be uniquely defined because of mechanical errors such as wheels that drift or a slippery gripper. Since we have yet to tackle

these issues, they remain as critical obstacles toward our goal of fully unified planning and control.

We will first discuss our approach toward extending the *planner* to operate in a dynamic world with limited sensing. Variant action outcomes could be modeled by PROPEL through the use of simulation descriptions that include choice points. Thus, the simulation would split into alternative continuations that correspond to variant action outcomes such as moving straight or veering left. We just need to recognize that the controller cannot actually select a particular outcome, so no SCRs would be created for those choice points. Exogenous events could be modeled by PROPEL using concurrent procedures. For example, an asynchronous exogenous event such as wind can be simulated by a concurrent procedure that models the effects of a periodic wind. In either case, variant outcomes or exogenous events, the planner could project high probability contingencies. The planner would then produce SCRs to advise the controller about choices that are contingent on exogenous events or variant outcomes. Thus, the controller would react to the events using the contingency SCRs in a manner similar to the Traverse and Robustify technique of Drummond and Bresina [12].

It is currently possible to write a PROPEL procedure that updates its sensors during planning by explicitly reading the sensors. This provides a method for planning in a changing world, but it requires the programmer to anticipate and pre-program explicit sensor update instructions. We are therefore working on a more general solution. As discussed in Section 4.3.3, are developing a method called *dynamic dependencies* that will allow the planner to incorporate asynchronous sensor reports into its search process, and to replan when significant changes in the external world are detected. Another issue for reactive systems is that of limited sensing. Although we think PROPEL can be extended to remove our current assumption of global sensing, we have not pursued this issue very far. The basic approach is to limit the search horizon to correspond with the perceptual horizon. We also hope to explore the issue of planning to gain information by implementing our NASA application's sensor interpretor component in PROPEL.

We now discuss how the *controller* could be extended to operate in a dynamic world. First, we believe that standard feedback algorithms that are used to make existing controllers reactive could be encoded in PROPEL. However, rather than placing the entire burden on the programmer, PROPEL could provide some additional support. This area is where McDermott's RPL is stronger than PROPEL. RPL [32] provides many ways to pre-program contingency control behavior, without planning. For instance, RPL's "with-policy" construct provides the ability to say "maintain condition c while performing procedure p". This instructs the controller that if condition c ever becomes false while performing procedure p, it should interrupt procedure p and re-establish condition c. For example, if the agent drops the tile while moving across the hall, it would immediately stop moving, pick up the dropped tile, and then continue moving. With a bit of work, we could provide a similar mechanism that would push a recovery procedure onto the control stack when a protected condition is violated. We could label some preconditions as "protected", and the controller could then react to protected condition violations using a form of an "interrupt handler". If a protected precondition then becomes false while executing the procedure, a recovery procedure would be pushed onto the control stack. When the recovery procedure reestablished the protected condition, it would be popped

off the control stack, and the original procedure could be resumed. However, if the recovery procedure changed the world in any significant way, the agent would need to physically backtrack to a resumption point.

As described earlier, we also need to develop a method for estimating evolving plan durations that can be used by an executive to determine when to stop planning and start execution. We have only done preliminary work on this but it is essential for developing an executive that meets deadlines.

Backtracking issues

The most immediate problem we experienced while developing our example application was related to backtracking. Backtracking in PROPEL is controlled by the heuristic node scheduling function `best-nodes`. When a node in the current expansion beam fails, or is no longer among the "best", it will be replaced by the next best node. In our example, the local heuristics partition the choices into two classes: those that decrease Manhattan distance by moving the agent toward a local goal (e.g., a door), and those that increase Manhattan distance by moving the agent away from the local goal. We call the first class, the *reaction space* because the controller will reactively always move toward the goal (unless it is following a wall). The `planner-best-nodes` function always searches the reaction space first.

All of the example problems presented in this paper contained a solution within this reaction space. However, when no solution is found in the reaction space, then backtracking becomes quite inefficient. Since the reaction space is not partitioned by subproblems, the planner will backtrack through the entire reaction space before trying a solution where it must temporarily increase the Manhattan distance to the goal. For instance, if the agent gets trapped at the very end of the delivery, the planner will backtrack through the entire reaction space which includes rethinking the very first steps toward picking up the tile. In such cases, we would like to modify the search bias so that it stays within the reaction space only for the current subproblem (i.e., the procedure currently on the top of the control stack). Since the backtracking in PROPEL is controlled by application-specific heuristics, a large part of the solution may be application-specific. Bresina has described a learning approach to this problem in [3].

Other interesting options for addressing our backtracking limitations include dependency-directed backtracking, and using critics as a form of heuristic backtracking. In plan transformation systems such as XFRM, a *critic* is often used to identify "bugs" in the plan. Although it is beyond the scope of our current work, similar critics could be implemented in PROPEL as a form of global heuristic that would identify relevant backtracking points. The `planner-best-nodes` function could look at paths in the search tree and determine that the best plan so far has a bug in it due to a choice made in the middle of the path. Functioning like a "critic", `planner-best-nodes` would then backtrack to that choice point in order to fix the bug.

Abstract procedures

The final area for future work we will discuss is the category of abstract procedures. We would like to annotate choice points with an "abstraction level" so that the planner could develop a skeletal plan by instantiating high-level or "critical" choice points first.

This would allow a time-constrained planner to focus on important choices first, leaving lower-level details to be heuristically selected by the controller if planning time runs out. This requires a technique for simulating high-level actions. In PROPEL, the user can provide a simulation for any procedure – it does not have to be a "primitive action". However, many issues would need to be addressed to make this work. For instance, how do you model the effects of going to a door without simulating the individual steps toward the door? This issue is also discussed by McDermott in [32], although it is not clear if XFRM provides any general theory for modeling abstract operators.

7.3. Contributions

We have shown how PROPEL's planner can be used as a feedforward component to extend a controller's operating range using a general purpose programming language. A tool that facilitates the use of planning techniques by control system programmers may lead to wider use and evaluation of planning techniques for realistic applications. We now summarize PROPEL's primary contributions.

- a procedurally expressive search engine,
- a unified planning and control architecture,
- an interruptable planner for a general programming language,
- an tool for studying integrated planning and control.

A procedurally expressive search engine

PROPEL uses an expressive action representation that captures the procedural complexities of practical control programs, yet can still be simulated by a search-based planner. This provides three primary benefits. First, we can represent behavior for larger, more complex control applications. This can increase the use and evaluation of AI planning techniques by control system programmers. Second, the increased expressiveness allows a single action representation to span the continuum from predictive to reactive systems. This facilitates the study of tradeoffs between pre-programmed and search-based behavior, and allows applications to be *tuned* along this continuum. Third, the expressiveness makes it practical for the planner and controller to use the same language, which makes it more practical to use the same interpreter for both planning and control. This in turn facilitates the study and development of tightly integrated, closed-loop planning and control systems.

A unified planning and control architecture

Our system provides unified planning and control because the planner and controller use exactly the same action representation, data structures and procedure interpreter. This facilitates smooth transitions back and forth between planning and execution.

The unified interpreter helps when switching *from planning to control* because SCRs that describe the planner's control stack can be applied to the controller's control stack. When the controller drops off the end of a plan (i.e., it runs out of SCRs), it continues executing by using heuristic defaults without stopping. This transition from using planner advice to using pre-programmed heuristic advice is transparent to the controller.

The unified architecture helps when switching *from control to planning* (for replanning) because the planner can reason about the controller's state and history in order to recover from execution failures. The planner can evaluate error recovery options by processing the controller's control stack. This ability of the planner to reason about execution failures is troublesome when the planner and controller speak different languages [19]. Of course there are limits to how the shared data structures can be used. For instance, the controller must obey the laws of physics when recovering from a failure, and must therefore backtrack physically rather than computationally like the planner. Other systems that integrate planning and control must also respect these limits.

Another benefit of the unified representation is that designers spend less effort developing and maintaining separate planning and control programs. A single program is required to encode most control procedures because only those that directly call effector commands require action simulations. For our Tileworld example, the same local heuristics were used by both the planner and the controller. Also, the same `best-nodes` function was used by the planner and the controller, except when the wall-following procedures were being used. The `terminate?` functions also differ only slightly between planning and control.

An interruptable planner for a general programming language

To meet real-time constraints, the planner must be interruptable and able to provide useful results at *any time* [2,9]. Although PROPEL's planner is not inherently "anytime", it is inherently interruptable. PROPEL's general programming constructs facilitate the design of complex control procedures that can be interrupted at any time according to either a time limit or an asynchronous halt message. The partial plans produced when the planner is interrupted provide advice for the most immediate choice points that will be faced by the controller. The degree to which the partial plans improve with planning time is application-dependent, and relies on how the search control heuristics interact with the environment.

A tool for studying integrated planning and control

There are many issues that need to be addressed before the fields of planning and control are *fully* unified. We hope that PROPEL's unified architecture can accelerate the exploration of those issues using a procedurally expressive action representation. We intend to use PROPEL to explore the continuum of points along the reactive/predictive spectrum by studying the tradeoffs and tuning applications so that search is used only where it is *required*. Using this platform, an experimenter can easily compare two procedures that differ only where one uses a choice point and the other uses a preprogrammed conditional. We also hope to use PROPEL to study how heuristics and the environment affect the quality of partial plans produced by an interruptable planner.

PROPEL's unified architecture also facilitates the development and evaluation of different executive models for closed-loop planning and control. We are currently using PROPEL to implement a computer model of human frontal lobe function [26,27]. The model is based on neuropsychological models of human executive functions that provide the flexibility required to operate in novel situations. In this model, the default reactions

are used in routine operating conditions. The planner is used to detect novel conditions, and to replace irrelevant, ineffective, and interfering reactions with deliberate plans.

8. Conclusion

Although modern control software has been very successful, it usually relies on the ability to predict all operating conditions in advance. In cases where environmental conditions and the effects of control actions cannot be entirely predicted at design time, planning can serve as a feedforward control mechanism. Our hypothesis is that a planner can extend the operating range of a real-time controller by generating novel behavior to handle unusual situations.

To test that hypothesis, we have developed a general programming language that permits predictive search techniques to be embedded within real-time control programs. We have shown how this language, with its *procedural search engine*, provide a unified architecture for tightly integrated planning and real-time control. The planner performs look-ahead search on the procedures and advises the controller about which selections to make at choice points. The planner's advice facilitates a graceful degradation of the controller's performance when it encounters situations that were not fully pre-programmed at design time. We have described how PROPEL provides benefits derived from a procedurally expressive action representation and a unified planning and control architecture. We hope these features can be used to explore the many questions that remain about the nature of integrated planning and control.

Acknowledgements

Many thanks to the members of the ERE group—Mark Drummond, John Bresina and Keith Swanson for many useful discussions on the relation between planning and control, and for their comments on early versions of this work. Special thanks to Keith for listening to these ideas for a long time, and for providing essential advice about how to present it coherently. Additionally, I'd like to thank David Thompson and Peter Robinson, my colleagues on the Intelligent Scientific Instrument project, for supporting this effort and providing excellent feedback on previous drafts. Thanks to Peter Friedland for providing a research environment that facilitated the development of this work.

Appendix A. The PROPEL grammar

(**Defprocedure** ⟨name⟩(⟨arg⟩*)
  [**:Goal** ⟨pattern⟩]
  [**:Preconditions** ⟨cond⟩+]
  :Body ⟨expr⟩+
  [**:Simulation** ⟨expr⟩+])

(**Defglobal** ⟨symbol⟩ ⟨lisp-term⟩)

Expr → If | While | Until | For | Assign | Choose-value | Subroutine-call |
Choose-procedure | Fail | Run-process | Wait-until | Sleep
If → (**If** ⟨cond⟩ **Then** ⟨expr⟩+ [**Else** ⟨expr⟩+])
Cond → ⟨lisp-function⟩ | ⟨assign⟩
While → (**While** ⟨cond⟩ **Do** ⟨expr⟩+)
Until → (**Until** ⟨cond⟩ **Do** ⟨expr⟩+)
For → (**For** ⟨symbol⟩ **In** ⟨lisp-term⟩ [**When** ⟨cond⟩] {**Do** | **Collect**} ⟨expr⟩+)
Assign → (⟨var⟩ ← ⟨expr⟩)
Choose-value → (⟨var⟩ ← (**Choose-value** ⟨lisp-term⟩ [**:heuristic** ⟨lisp-function⟩]))
Subroutine-call → ⟨lisp-function⟩ | ⟨propel-procedure⟩
Choose-procedure → (**Choose-procedure** ⟨pattern⟩ [**:heuristic** ⟨lisp-function⟩])
Run-process → (**Run-process** ⟨function-name⟩ ⟨arg⟩*)
Wait-until → (**Wait-until** ⟨cond⟩)
Sleep → (**Sleep** ⟨seconds⟩)
Fail → **(Fail)**
Propel-procedure → (⟨name⟩ ⟨arg⟩*)
Lisp-function → (⟨name⟩ ⟨arg⟩*)
Lisp-term → ⟨lisp-function⟩ | ⟨symbol⟩
Pattern → (⟨symbol⟩+)
Name→ ⟨symbol⟩
Arg → ⟨symbol⟩
Var → ⟨symbol⟩
Function-name → ⟨symbol⟩
Symbol → ⟨lisp-symbol⟩

Notation.

⟨ ⟩: non-terminal,
[ ]: optional,
{ }: grouping,
+: one or more,
*: zero or more,
|: disjunction.

Appendix B. The Tileworld application

```
(defglobal *state* (read-sensors))
(defglobal *rank* 1)

(defprocedure move-tile (tile destination)
   :body
   (pickup-tile tile)
   (deliver-tile destination))

(defprocedure pickup-tile (tile)
```

```
   :body
   (tile-loc <- (get-tile-loc tile *state*))
   (go-to-room (what-room? tile-loc))
   (go-next-to-cell tile-loc)
   (grasp-tile (adjacent-direction (get-agent-loc *state*)
                                   tile-loc)))

(defprocedure deliver-tile (destination)
   :body
   (go-to-room (what-room? destination))
   (agent-dir <- (opposite-dir (get-grasp-dir *state*)))
   (agent-destination <- (adjacent-cell destination agent-dir))
   (go-to-cell agent-destination))

(defprocedure go-to-room (destination-room)
  :body
  (agent-room <- (what-room? (get-agent-loc *state*)))
  (if (not (equal agent-room destination-room))
      then (if (not (hallway? agent-room ))
               then (go-thru-door 'exit agent-room))
           (if (not (hallway? destination-room))
               then (go-thru-door 'enter destination-room))))

(defprocedure go-thru-door (direction room)
   :body
   (door-loc <- (choose-value (door-locations room)
                  :heuristic (closest-loc(get-agent-loc *state*))))
   (doorstep-loc <- (get-doorstep-location door-loc direction))
   (go-to-cell doorstep-loc)
   (move-dir <- (adjacent-direction doorstep-loc door-loc))
   (if (grasping-object? *state*)
       then (carry move-dir)
            (carry move-dir)
       else (move move-dir)
            (move move-dir)))

(defprocedure go-next-to-cell (cell)
   :body
   (pickup-loc <-
      (choose-value (adjacent-cells cell)
         :heuristic (closest-loc (get-agent-loc *state*))))
   (go-to-cell pickup-loc))

(defprocedure go-to-cell (goal-loc)
   :body
```

```
   (until (and (agent-loc <- (get-agent-loc *state*))
               (equal agent-loc goal-loc))
       do (move-dir <- (choose-value '(n s e w)
                          :heuristic
                             (closest-dir (get-agent-loc *state*)
                                          goal-loc)))
          (choose-procedure (take a step in move-dir goal-loc)
             :heuristic (prefer-first-choice))))

;********** Effector command level procedures ***************

(defprocedure move (?dir)
   :goal (take a step in ?dir ?goal-loc)
   :preconditions (not (grasping-object? *state*))
                  (agent-loc <- (get-agent-loc *state*))
                  (target-loc <- (adjacent-cell agent-loc ?dir))
                  (cell-empty? target-loc *state*)
                  (in-bounds? target-loc)
   :body (move-agent ?dir)
         (wait *reactor-sleep-time*)
         (*state* <- (read-sensors))
   :simulation
         (*state* <- (remove-fact '(at agent-loc) *state*))
         (*state* <- (add-fact '(at target-loc) *state*))

(defprocedure carry (?dir)
   :goal (take a step in ?dir ?goal-loc)
   :preconditions (grasp-dir <- (get-grasp-dir *state*))
                  (agent-loc <- (get-agent-loc *state*))
                  (target-loc <- (adjacent-cell agent-loc ?dir))
                  (cell-empty? target-loc *state*)
                  (tile-loc <- (adjacent-cell agent-loc grasp-dir))
                  (new-tile-loc <- (adjacent-cell tile-loc ?dir))
                  (cell-empty? new-tile-loc *state*)
                  (in-bounds? new-tile-loc)
                  (in-bounds? target-loc)
   :body (move-agent ?dir)
         (wait *reactor-sleep-time*)
         (*state* <- (read-sensors))
   :simulation
         (tile <- (cell-contents tile-loc *state*))
         (*state* <- (remove-fact '(at agent-loc) *state*))
         (*state* <- (remove-fact '(in-cell tile tile-loc) *state*))
         (*state* <- (add-fact '(at target-loc) *state*))
         (*state* <-
```

```
            (add-fact '(in-cell tile new-tile-loc) *state*)))

(defprocedure grasp-tile  (direction)
   :preconditions (agent-loc <- (get-agent-loc *state*))
                  (tile-loc <- (adjacent-cell agent-loc direction))
   :body (grasp direction)
         (wait *reactor-sleep-time*)
         (*state* <- (read-sensors))
   :simulation
         (*state* <- (add-fact '(grasp-dir direction) *state*)))

;;;********** Global heuristic and termination functions **********

(defun planner-best-nodes (search-record)
   (let (best)
        (loop for node in (open-nodes search-record)
           when (or (null best)
                    (< (get-global node '*rank*) (first best)))
           do (setq best (cons (get-global node '*rank*) node)))
        (list (cdr best))))

(defun controller-best-nodes (search-record)
   (let (best)
        (loop for node in (open-nodes search-record)
           when (or (null best)
                    (< (get-global node '*rank*) (first best)))
           do (setq best (cons (get-global node '*rank*) node)))
        (list (cdr best))))

(defun planner-terminate? (search-record)
   (or (deadline-expired? search-record)
       (null (search-record-open search-record))
       (search-record-success search-record)))

(defun controller-terminate? (search-record)
   (or (search-record-success search-record)
       (search-record-failure search-record)
       (null (search-record-open search-record))
       (deadline-expired? search-record)))

;;; **** The  following version of planner-best-nodes
;;; **** was used for experiment 1 to prevent the planner from
;;; **** selecting wall-following reactive procedures

(defun planner-best-nodes (search-record)
```

```
   (let (best)
        (loop for node in (open-nodes search-record)
           when (and (not (following-wall? node))
                     (or (null best)
                         (< (get-global node '*rank*)
                            (first best))))
           do (setq best (cons (get-global node '*rank*) node)))
        (list (cdr best))))

;;;;;********** Local heuristic functions ******************

(defun closest-loc (current-loc choice-nodes
                                &optional planner-choice)
   (let ((closest (list 9999))
         (choices (get-choice-values choice-nodes))
         distance)
        (loop for choice in choices
           do (setq distance
                    (manhattan-distance current-loc (car choice)))
              (when (< distance (car closest))
                 (setq closest (cons distance (cdr choice)))))
        (loop for choice in choices
           do (set-global (cdr choice) '*rank*
                 (cond((equal (car choice) planner-choice) 0)
                      ((eq (cdr closest)(cdr choice)) 1)
                      (t 2))))))

(defun closest-dir (agent-loc goal-loc choice-nodes
                    &optional planner-choice)
   (let* ((agent-x (first agent-loc))
          (agent-y (second agent-loc))
          (goal-x (first goal-loc))
          (goal-y (second goal-loc)))
         (loop for choice-node in choice-nodes
            do (setq choice-value (get-choice-value choice-node))
               (set-global choice-node '*rank*
                  (if (equal choice-value planner-choice) 0
                      (case choice-value
                            (w (if (> agent-x goal-x) 1 2))
                            (e (if (< agent-x goal-x) 1 2))
                            (s (if (> agent-y goal-y) 1 2))
                            (n (if (< agent-y goal-y) 1 2)))))))))

(defun prefer-first-choice (choice-nodes &optional planner-choice)
   (loop for choice in (get-choice-values choice-nodes)
```

```
      for rank from 1
      do (set-global (cdr choice) '*rank*
                     (if (equal (car choice) planner-choice)
                         0
                         rank))))

;;;;********** Reactive wall-following procedures ***************

(defprocedure move-around-obstacle (?dir ?goal-loc)
   :goal (take  a step in ?dir ?goal-loc)
   :preconditions (not (grasping-object? *state*))
                  (agent-loc <- (get-agent-loc *state*))
                  (target-loc <- (adjacent-cell agent-loc ?dir))
                  (not (cell-empty? target-loc *state*))
                  (in-bounds? target-loc)
   :body
   (detour-dir <- (choose-value (wall-directions ?dir)
                     :heuristic (closest-dir agent-loc ?goal-loc)))
   (follow-wall detour-dir agent-loc ?dir)
   (move ?dir))

(defprocedure follow-wall (detour-dir agent-loc dir)
   :body
   (detour <- (list agent-loc))
   (until (and (detour-target-loc <- (adjacent-cell agent-loc
                                                    detour-dir))
               (target-loc <- (adjacent-cell agent-loc dir))
               (cell-empty? target-loc *state*)
               (cell-empty? detour-target-loc *state*)
               (not (member target-loc detour :test 'equal)))
      do (if (and (detour-loc  <- (adjacent-cell agent-loc
                                                 detour-dir))
                  (cell-empty? detour-loc *state*)
                  (not (member detour-loc detour :test 'equal)))
             then (choose-procedure
                     (take a step in detour-dir bogus-goal)
                     :heuristic (prefer-first-choice))
             elseif (and (backup-dir <- (opposite-dir dir))
                         (detour-loc <-
                            (adjacent-cell agent-loc backup-dir))
                         (cell-empty? detour-loc *state*)
                         (not (member detour-loc detour
                                      :test 'equal)))
             then (backup-dir <- (opposite-dir dir))
                  (choose-procedure
```

```
                    (take a step in backup-dir bogus-goal)
                    :heuristic (prefer-first-choice))
            else (backup-detour-dir <- (opposite-dir detour-dir))
                 (choose-procedure
                    (take a step in backup-detour-dir bogus-goal)
                    :heuristic (prefer-first-choice)))
  (agent-loc <- (get-agent-loc *state*))
  (detour <- (cons  agent-loc detour))))

(defprocedure carry-around-obstacle (?dir ?goal-loc)
   :goal (take a step in ?dir ?goal-loc)
   :preconditions (grasp-dir <- (get-grasp-dir *state*))
                  (agent-loc <- (get-agent-loc *state*))
                  (target-loc <-  (adjacent-cell agent-loc ?dir))
                  (tile-loc <- (adjacent-cell agent-loc grasp-dir))
                  (target-tile-loc <- (adjacent-cell tile-loc ?dir))
                  (or (not (cell-empty? target-tile-loc *state*))
                      (not (cell-empty? target-loc *state*)))
                  (in-bounds? target-loc)
   :body
   (detour-dir <- (choose-value (possible-dirs ?dir)
                     :heuristic (closest-dir agent-loc ?goal-loc)))
   (follow-wall-with-object
      detour-dir agent-loc ?dir grasp-dir tile-loc)
   (carry ?dir))

(defprocedure follow-wall-with-object (detour-dir agent-loc dir
                                       grasp-dir tile-loc)
   :body
   (detour <- (list tile-loc))
   (until (and (target-loc <- (adjacent-cell agent-loc dir))
               (tile-target-loc <- (adjacent-cell tile-loc dir))
               (cell-empty? tile-target-loc *state*)
               (cell-empty? target-loc *state*)
               (not (member tile-target-loc detour :test 'equal)))
      do (if (and (tile-detour-loc <-
                     (adjacent-cell tile-loc detour-dir))
                  (detour-loc <-
                     (adjacent-cell agent-loc detour-dir))
                  (cell-empty? detour-loc *state*)
                  (cell-empty? tile-detour-loc *state*)
                  (not (member tile-detour-loc detour
                               :test 'equal)))
             then (choose-procedure
                     (take a step in detour-dir bogus-goal)
```

```
                :heuristic (prefer-first-choice))
          elseif (and (backup-dir <- (opposite-dir dir))
                      (tile-detour-loc <-
                         (adjacent-cell tile-loc backup-dir))
                      (detour-loc <-
                         (adjacent-cell agent-loc backup-dir))
                      (cell-empty? detour-loc *state*)
                      (cell-empty? tile-detour-loc *state*)
                      (not (member tile-detour-loc detour
                                   :test 'equal)))
          then (backup-dir <- (opposite-dir dir))
               (choose-procedure
                  (take a step in backup-dir bogus-goal)
                  :heuristic (prefer-first-choice))
          else (backup-detour-dir <- (opposite-dir detour-dir))
               (choose-procedure
                  (take a step in backup-detour-dir bogus-goal)
                  :heuristic (prefer-first-choice)))
     (agent-loc <- (get-agent-loc *state*))
     (tile-loc <- (adjacent-cell agent-loc grasp-dir))
     (detour <- (cons  tile-loc detour))))
```

References

[1] M. Beetz and D. McDermott, Declarative goals in reactive plans, in: *Proceedings First International Conference on AI Planning Systems* (1992) 3–12.

[2] M. Boddy and T.L. Dean, Deliberation scheduling for problem solving in time-constrained environments, *Artif. Intell.* **67** (2) (1994) 245–285.

[3] J. Bresina, Design of a reactive system based on classical planning, in: *Working Notes 1993 AAAI Spring Symposium Series (Session on Foundations of Automatic Planning)*, Stanford, CA (1993) 5–9.

[4] J. Bresina and M. Drummond, Integrating planning and reaction: a preliminary report, in: *Proceedings 1990 AAAI Spring Symposium Series (Session on Planning in Uncertain, Unpredictable, or Changing Environments)*, Stanford, CA (1990).

[5] R. Brooks, A robust layered control system for a mobile robot, *IEEE J. Rob. Automation* **2** (1986) 14–23.

[6] J. A. Campbell, ed., *Implementations of Prolog* (Ellis Horwood, Chichester, England, 1984).

[7] D. Chapman, Planning for conjunctive goals, *Artif. Intell.* **32** (1987) 333–377.

[8] K. Currie and A. Tate, O-Plan: the open planning architecture, *Artif. Intell.* **52** (1) (1991) 49–86.

[9] T.L. Dean and M. Boddy, An analysis of time-dependent planning, in: *Proceedings AAAI-88*, St. Paul, MN (1988) 49–54.

[10] T.L. Dean and M. Wellman, *Planning and Control* (Morgan Kaufman, San Mateo, CA, 1991).

[11] M. Drummond, Situated control rules, in: *Proceedings First International Conference on Principles of Knowledge Representation and Reasoning*, Toronto, Ont. (1989) 103–113.

[12] M. Drummond and J. Bresina, Anytime synthetic projection: maximizing the probability of goal satisfaction, *Proceedings AAAI-90*, Boston, MA (1990) 138–144.

[13] M. Drummond, J. Bresina, K. Swanson and R. Levinson, Reaction-first search: incremental planning with guaranteed performance improvement, in: *Proceedings IJCAI-93*, Chambery, France (1993) 1408–1414.

[14] R.E. Fikes and N.J. Nilsson, STRIPS: a new approach to the application of theorem proving to problem solving, *Artif. Intell.* **2** (1971) 189–208.

[15] R.J. Firby, An investigation into reactive planning in complex domains, in: *Proceedings AAAI-87*, Seattle, WA (1987) 202–206.
[16] P. Friedland, Knowledge-based experiment design in molecular genetics, Ph.D. Thesis, Stanford University, Stanford, CA (1979).
[17] M.P. Georgeff and A.L. Lansky, Reactive reasoning and planning, in: *Proceedings AAAI-87*, Seattle, WA (1987) 677–682.
[18] K. Hammond, Explaining and repairing plans that fail, *Artif. Intell.* **45** (1990) 173–228.
[19] S. Hanks and R.J. Firby, Issues and architectures for planning and execution, in: *Proceedings Workshop on Innovative Approaches to Planning, Scheduling and Control*, San Diego, CA (1990) 59–70.
[20] E. Horowitz and S. Sahni, *Fundamentals of Computer Algorithms* (Computer Science Press, Rockville, MD, 1984).
[21] L.P. Kaelbling, Goals as parallel program specifications, in: *Proceedings AAAI-88*, St. Paul, MN (1988) 60–65.
[22] S. Kambhampati and J. Hendler, A validation-structure-based theory of plan modification and reuse, *Artif. Intell.* **55** (1992) 193–258.
[23] R.E. Korf, Planning as search: a quantitative approach, *Artif. Intell.* **33** (1987) 566–577.
[24] R.E. Korf, Real-time heuristic search, *Artif. Intell.* **42** (1990) 189–211.
[25] J.E. Laird and P.S. Rosenbloom, Integrating execution, planning and learning in Soar for external environments, in: *Proceedings AAAI-90*, Boston, MA (1990) 1022–1029.
[26] R. Levinson, Human frontal lobes and AI planning systems, in: *Proceedings Second International Conference on AI Planning Systems (AIPS-94)*, Chicago, IL (1994) 305–310.
[27] R. Levinson, A computer model of human frontal lobe function (a preliminary report), NASA Ames Research Center, AI Research Branch Technical Report FIA-94-15 (1994).
[28] R. Levinson, P. Robinson and D. Thompson, Integrated perception, planning and control for autonomous soil analysis, in: *Proceedings 9th IEEE Conference on AI for Applications*, Orlando, FL (1993) 249–255.
[29] D. Lyons, and A. Hendriks, A practical approach to integrating reaction and deliberation, in: *Proceedings First International Conference on AI Planning Systems* (1992) 153–162.
[30] D. McAllester and D. Rosenblitt, Systematic nonlinear planning, in: *Proceedings AAAI-91*, Anahei, CA (1991) 634–639.
[31] D. McDermott, Planning reactive behavior: a progress report, in: *Proceedings Workshop on Innovative Approaches to Planning, Scheduling and Control*, San Diego, CA (1990) 450–458.
[32] D. McDermott, Transformational planning of reactive behavior, Technical Report YALEU/CSD/RR#941, Yale University, Department of Computer Science, New Haven, CT (1992).
[33] N.J. Nilsson, *Principles of Artificial Intelligence* (Tioga, Palo Alto, CA, 1980).
[34] A. Philips and J. Bresina, NASA Tileworld Manual, NASA Technical Report TR-FIA-91-11, Code FIA, NASA Ames Research Center, Moffett Field, CA (1992).
[35] S. Russell and S. Zilberstein, Composing real-time systems, in: *Proceedings IJCAI-91*, Sydney, Australia (1991) 212–217
[36] A. Saffiotti, K. Konolige and E. Ruspini, A multivalued logic approach to integrating planning and control, *Artif. Intell.* **76** (1995) Special Issue on Planning and Scheduling (this issue).
[37] J. Siskind and D. McAllester, Nondeterministic Lisp as a substrate for constraint logic programming, in:*Proceedings AAAI-93*, Washington, DC (1993) 133–138.
[38] D. Wilkins, *Practical Planning: Extending the Classical AI Paradigm* (Morgan Kaufman, San Mateo, CA, 1988).
[39] R. Zabih, D. McAllester and D. Chapman, Nondeterministic Lisp with dependency-directed backtracking, in: *Proceedings AAAI-87*, Seattle, WA (1987) 59–64.