Autonomous Robots, ??, 1–18 (??)

Interleaving Planning and Robot Execution for Asynchronous User Requests

KAREN ZITA HAIGH
khaigh@cs.cmu.edu
http://www.cs.cmu.edu/~khaigh

MANUELA M. VELOSO
mmv@cs.cmu.edu
http://www.cs.cmu.edu/~mmv

Computer Science Department, Carnegie Mellon University, Pittsburgh PA 15213-3891
;

Abstract.
ROGUE is an architecture built on a real robot which provides algorithms for the integration of high-level planning, low-level robotic execution, and learning. ROGUE addresses successfully several of the challenges of a dynamic office gopher environment. This article presents the techniques for the integration of planning and execution.

ROGUE uses and extends a classical planning algorithm to create plans for multiple interacting goals introduced by asynchronous user requests. ROGUE translates the planner's actions to robot execution actions and monitors real world execution. ROGUE is currently implemented using the PRODIGY4.0 planner and the Xavier robot. This article describes how plans are created for multiple asynchronous goals, and how task priority and compatibility information is used to achieve appropriate efficient execution. We describe how ROGUE communicates with the planner and the robot to interleave planning with execution so that the planner can replan for failed actions, identify the actual outcome of an action with multiple possible outcomes, and take opportunities from changes in the environment.

ROGUE represents a successful integration of a classical artificial intelligence planner with a real mobile robot.

1. Introduction

We have been working towards the goal of building autonomous robotic agents that are capable of planning and executing high-level tasks, and learning from the analysis of execution experience. This article presents our work extending the high-level reasoning capabilities of a real robot. One of the goals of this research project is to have the robot move autonomously in an office building reliably performing office tasks such as picking up and delivering mail and computer printouts, returning and picking up library books, and carrying recycling cans to the appropriate containers.

Our framework consists of integrating Xavier [O'Sullivan, Haigh, & Armstrong, 1997, Simmons *et al.*, 1997] with the PRODIGY4.0 planning and learning system [Veloso *et al.*, 1995]. The resulting architecture and algorithms is ROGUE [Haigh & Veloso, 1997]. ROGUE provides a setup where users can post tasks for which the planner generates appropriate plans, delivers them to the robot, monitors their execution, and learns from feedback from execution performance. ROGUE effectively enables the communication between Xavier, PRODIGY4.0 and the users.

In this article, we focus on presenting the techniques underlying the planning and execution control in ROGUE. The learning algorithm is under development and results are being compiled and can be found in [Haigh & Veloso, 1998]. The planning and execution capabilities of ROGUE form the foundation for a complete, learning, autonomous agent.

ROGUE generates and executes plans for multiple interacting goals which arrive asynchronously and whose task structure is not known *a priori*. ROGUE interleaves tasks and reasons about task priority and task compatibility. ROGUE enables the communication between the planner and the robot, allowing the system to successfully interleave planning and execution to detect successes or failures and to respond to them. ROGUE controls the execution of a real robot to accomplish tasks in the real world.

1.1. System Architecture

Figure 1 shows the general architecture of the system. ROGUE accepts tasks posted by users, calls the task planner, PRODIGY4.0, which generates appropriate plans, and then posts actions to the robot, Xavier, for execution. ROGUE provides appropriate search control knowledge to the planner and monitors the outcome of execution.

Xavier is a mobile robot being developed at Carnegie Mellon University [O'Sullivan, Haigh, & Armstrong, 1997, Simmons *et al.*, 1997] (see Figure 2).

It is built on an RWI B24 base and includes bump sensors, a laser range finder, sonars, a color camera and a speech board. The software controlling Xavier includes both reactive and deliberative behaviours, integrated using the Task Control Architecture (TCA) [Simmons, 1994]. TCA provides facilities for scheduling and synchronizing tasks, resource allocation, environment monitoring and exception handling. The reactive behaviours enable the robot to handle real-time local navigation, obstacle avoidance, and emergency situations (such as detecting a bump). The deliberative behaviours include vision interpretation, maintenance of occupancy grids & topological maps, and path planning & global navigation. The underlying architecture is described in more detail by Simmons *et al.* [1997].

PRODIGY is a domain-independent planner that serves as a testbed for machine learning research [Carbonell, Knoblock, & Minton, 1990, Veloso *et al.*, 1995]. The current version, PRODIGY4.0 is a nonlinear planner that follows a state-space search guided by means-ends analysis and backward chaining. It reasons about multiple goals and multiple alternative operators to achieve the goals. It reasons about interacting goals, exploiting common subgoals and addressing issues of resource contention.

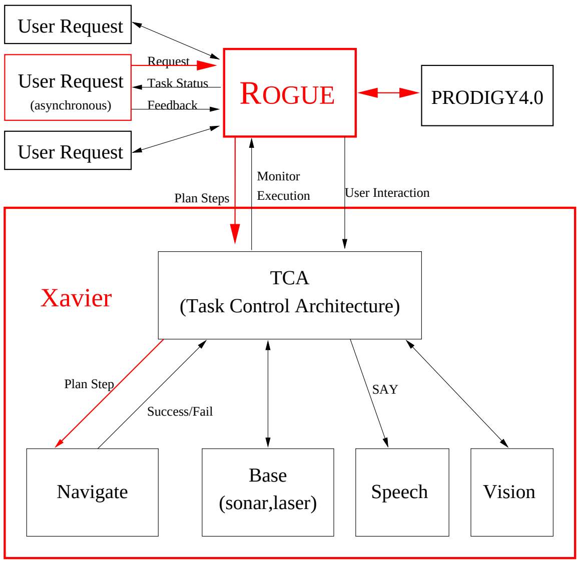

Fig. 1. ROGUE Architecture.

Fig. 2. Xavier the Robot.

Xavier reliably performs actions requested of it, but has no task planning abilities. PRODIGY4.0 , meanwhile, is a complex task planner that had never been used in a real execution domain; as such, it had never been used for asynchronous goals or in an environment where the state spontaneously changes. In combining the two systems, the challenges for ROGUE include developing a communication mechanism for control and feedback, as well as extending the planner to handle the dynamics of a real-world task.

1.2. *Planning and Execution in ROGUE*

There are a few approaches to creating plans for execution. Shakey [Nilsson, 1984] was the first system to use a planning system on a robot. This project was based on a classical planner that ignored real world uncertainty [Fikes, Hart, & Nilsson, 1972] and followed a deterministic model to generate a single executable plan. When execution failures occurred, replanning was invoked.

This pioneering approach has been acknowledged as partly successful, but also has been criticized for its lack of reactivity, and has led to significant research into planning systems that can handle the uncertainty of the real world. *Conditional planning* is one approach that aims at considering in the domain model all the possible contingencies of the world and planning ahead for each individual one [Atkins, Durfee, & Shin, 1996, Mansell, 1993, Pryor, 1994, Schoppers, 1989]. In most complex environments, the large number of possible contingencies means that complete conditional planning becomes infeasible, but may nevertheless be appropriate in particularly dangerous domains.

Probabilistic planning takes a more moderate approach in that it only creates conditional plans for the most likely problems [Blythe, 1994, Dean & Boddy, 1988, Gervasio & DeJong, 1991, Kushmerick, Hanks, & Weld, 1993]. It relies on replanning when unpredictable or rare events take place. Although this approach generates fast responses to most contingencies, it may miss potential opportunities that arise from changes in the world. It should be noted that none of these systems have ever been applied to a real robotic system.

Another moderate approach is that of *parallel planning and execution*, in which the planner and the executor are decoupled [Drummond *et al.*, 1993, Lyons & Hendriks, 1992, McDermott, 1992, Pell *et al.*, 1997]. The executor can react to the environment without a plan. The planner continually modifies the behaviour of the executor to increase the goal satisfaction probability. This approach leads to a system with fast reactions, but a set of default plans need to be pre-prepared, and in some situations may lead away from the desired goal. Furthermore, the planner creates its plans based on assumptions about the world that may have changed during planning time.

We take a third approach: that of *interleaving planning and execution*, as do several other researchers [Ambros-Ingerson & Steel, 1988, Dean *et al.*, 1990, Georgeff & Ingrand, 1989, Nourbakhsh, 1997]. Interleaving planning with execution allows the system to reduce its planning effort by pruning alternative possible outcomes immediately, and also to respond quickly and effectively to changes in the environment. For example, the system can notice limited resources such as battery power, or notice external events like doors opening and closing. In these ways, interleaving planning with execution can create opportunities for the system while reducing the planning effort.

One of the main issues raised by interleaved planning and execution is when to stop planning and start executing. Dean [1990] selects between alternative actions by selecting the one with the highest degree of information gain, but is therefore limited to reversible domains. Nourbakhsh [1997], on the other hand, executes actions that prefix all branches of a conditional plan created after making simplifying assumptions about the world. The assumptions are built so that the planner always preserves goal reachability, even in an irreversible world.

ROGUE has two methods for selecting when to take an action. The first method selects an action when it is the first in a chain of actions that are known to lead towards the goal. PRODIGY4.0 uses means-ends analysis to build plans backwards, working from the goal towards the initial state. Each action is described in terms of required preconditions and possible effects; actions are added to the plan when their effects are desirable. When all the preconditions of an action are believed to be true in the current state, ROGUE executes the action. Since PRODIGY4.0 already has a partial plan from the initial state to the goal state, the action ROGUE selects is clearly relevant to achieving the goal. Actions whose failures may lead to irreversible states are avoided until it has exhausted all other possible ways of reaching the goal. The second method is used when there are multiple actions available for selection. ROGUE selects between these actions in order to maximize overall expected execution efficiency.

When ROGUE selects an action for execution, it executes a procedure that confirms the preconditions of the action, then executes the action, and finally confirms the effects. In addition to the explicit confirmation of preconditions and effects of actions, our system also monitors events that may affect pending goals. Each goal type has a set of associated monitors that are invoked when a goal of that type enters the system. These monitors run parallel to planning and may modify the planner's knowledge at any time. A given monitor may, for example, monitor battery power or examine camera images for particular objects.

The ability to handle asynchronous goals is a basic requirement of a system executing in the real world. A system that only handles asynchronous goals in a first-come-first-served manner is inefficient and loses many opportunities for combined execution. ROGUE easily incorporates asynchronous goals into its system without losing context of existing tasks, allowing it to take advantage of opportunities as they arise. By intelligent combining of compatible tasks, ROGUE can respond quickly and efficiently to user requests.

Amongst the other interleaving planners, only PRS [Georgeff & Ingrand, 1989] handles multiple asynchronous goals. ROGUE however abstracts much of the lower level details that PRS explicitly reasons about, meaning that ROGUE can be seen as more reliable and efficient because system functionality is suitably partitioned [Pell *et al.*, 1997, Simmons *et al.*, 1997]. NMRA [Pell *et al.*, 1997] and 3_T [Bonasso & Kortenkamp, 1996] both function in domains with many asynchronous goals, but both planners respond to new goals and serious action failures by abandoning existing planning and restarting the planner. As stated by Pell *et al.*, establishing standby modes prior to invoking the planner is "a costly activity, as it causes [the system] to interrupt the ongoing planned activities and lose important opportunities." Throwing out all existing planning and starting over not only delays execution and but also can place high demands on sensing to determine current status of partially-executed tasks.

In summary, ROGUE's interleaving of planning and execution can be outlined in the following procedure for accomplishing a set of tasks:

1. A user submits a request, and ROGUE adds the task information to PRODIGY4.0's state.
2. PRODIGY4.0 creates a plan to achieve the goal(s), constrained by ROGUE's priority and compatibility knowledge, incorporating any new requests.
3. ROGUE sends selected actions to the robot for execution, confirming that its preconditions are valid.
4. ROGUE confirms the outcome of each action. If failure, ROGUE notifies PRODIGY4.0 and forces replanning.

In this section we have motivated our work and summarized the most relevant related work. In Section 2, we describe PRODIGY4.0, present how it plans for multiple asynchronous goals, and introduce ROGUE's priority and compatibility rules (steps 1 and 2 above). We include a detailed example of the system's behaviour for a simple two-goal problem, when the goals arrive asynchronously. In Section 3 we present execution, monitoring and how the system detects, processes and responds to failure (steps 3 and 4). Finally we provide a summary of ROGUE's capabilities in Section 4.

2. Planning for Asynchronous Requests

The office delivery domain involves multiple users and multiple tasks. A planner functioning in this domain needs to respond efficiently to task requests, as they arrive asynchronously. One common method for handling these multiple goal requests is simply to process them in a first-come-first-served manner; however, this method leads to inefficiencies and lost opportunities for combined execution of compatible tasks [Goodwin & Simmons, 1992].

ROGUE is able to process incoming asynchronous goal requests, to prioritize them, and to suspend lower priority actions when necessary.

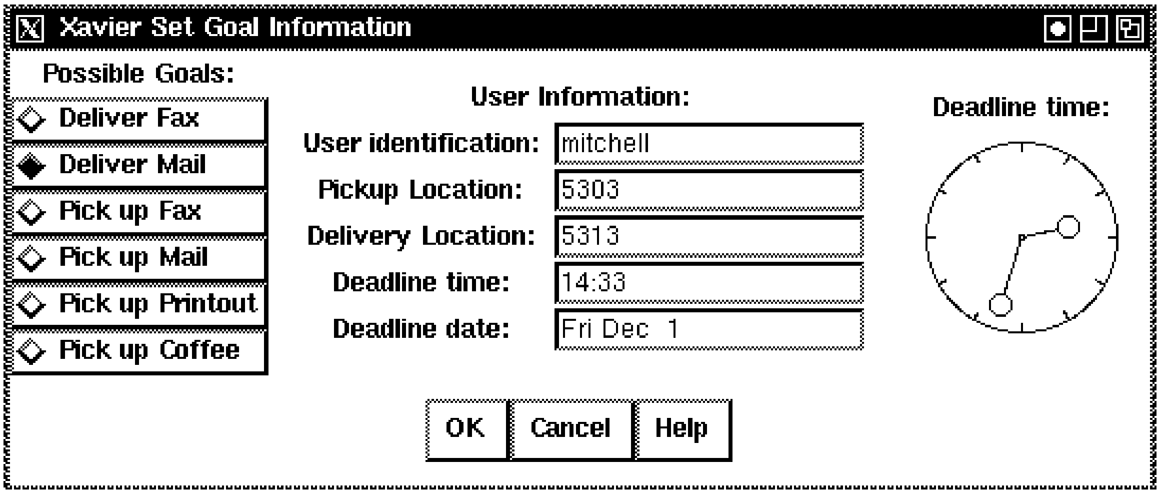

Fig. 3. User request interface.

It successfully interleaves compatible requests and creates efficient plans for completing all the tasks.

2.1. Receiving a Request

Users submit their task requests through one of three different interfaces: the World Wide Web [Simmons *et al.*, 1997], Zephyr [DellaFera *et al.*, 1988, Simmons *et al.*, 1997], or a specially designed graphical user interface (Figure 3) [Haigh & Veloso, 1996].

The slots in this last interface are automatically filled in with default information available from the user's plan file, and the deadline time defaults to one hour in the future. The interface can be extended with additional tasks at any time.

Each of these three interfaces forwards the request to ROGUE by TCA messages. When each new request comes in, ROGUE adds it to PRODIGY4.0's list of unsolved goals, and updates the task model, as shown in Table 1. The literal `(needs-item <user> <item>)` indicates that a request, sent by user `<user>`, is pending. This function is domain-dependent because the literals added relate strictly to this domain; however, the structure would be identical for any other domain with asynchronous goals.

There is currently no explicit mechanism for a user to rescind a request; however PRODIGY4.0 will no longer plan for (or attempt to apply operators for) the associated top-level goal if it is simply removed from G and PG.

Table 1. Integrating new task requests into PRODIGY4.0.

Define: $C \leftarrow$ current state
Define: $G \leftarrow$ top-level goal
Define: $PG \leftarrow$ pending goals cache (unsolved top-level goals and their subgoals)

Let R be the list of pending unprocessed requests
For each *request* $\in R$, turn *request* to goal:
- $C \Leftarrow C \cup \{$ `(needs-item` *request*`-userid` *request*`-object)`
`(pickup-loc` *request*`-userid` *request*`-pickup-loc)`
`(deliver-loc` *request*`-userid` *request*`-deliver-loc)` $\}$
- $G \Leftarrow$ `(and` G `(has-item` *request*`-userid` *request*`-object))`
- $PG \Leftarrow$ `(and` PG `(has-item` *request*`-userid` *request*`-object))`
- *request*-completed $\Leftarrow$ nil

Table 2. PRODIGY4.0 decision points, adapted from [Veloso *et al.*, 1995].

PRODIGY4.0
1. If the goal statement G is satisfied in the current state, terminate.
2. Either (A) Subgoal: add an operator to *Plan* (**Back-Chainer**), or
   (B) Apply: apply an operator from *Plan* (**Operator-Application**).
   Decision point: Decide whether to apply or to subgoal.
3. Recursively call **PRODIGY4.0** on the resulting plan.

Back-Chainer
1. Pick an unachieved goal or precondition g.
   Decision point: Choose an unachieved goal.
2. Pick an operator *op* that achieves g.
   Decision point: Choose an operator that achieves this goal.
3. Add *op* to *Plan*.
4. Instantiate the free variables of *op*.
   Decision point: Choose an instantiation for the variables of the operator.

Operator-Application
1. Pick an operator *op* in *Plan* which is an *applicable operator*, that is the preconditions of *op* are satisfied in the current state.
   Decision point: Choose an operator to apply.
2. Update the current state with the effects of *op*.

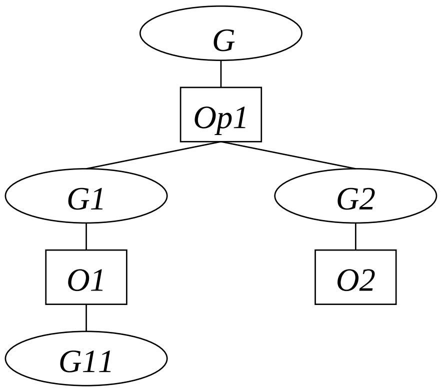

Fig. 4. Representation of a plan. G is the top-level goal, and $Op1$ is the operator that achieves it. $G1$ and $G2$ are two preconditions of $Op1$ that are not true in the current state, and are achieved by $O1$ and $O2$ respectively.

2.2. Creating the Plan

PRODIGY4.0 creates a plan for its unsolved goals by selecting operators whose effects achieve those goals. It continues adding operators to the incomplete plan until a solution to the problem is found. In Figure 4 we show a simple incomplete plan.

The plan is built by a backward-chaining algorithm, which starts from the list of goals, G, and adds operators, one by one, to achieve its *pending goals*, i.e., to achieve preconditions of other operators that are not satisfied in the current state.

When all the preconditions of an operator are satisfied in the current state, PRODIGY4.0 simulates the effects of the action by *applying* the operator. Each time an operator is applied, the current state is updated with the effects of the action. PRODIGY4.0 terminates planning when each of the goals are true in the state.

The planning cycle involves several decision points, including which goal to select from the set of pending goals, and which applicable operator to apply. Table 2 shows the decisions made while creating the plans. **Back-Chainer** shows the decisions made while back-chaining on the plan, **Operator-Application** shows how an operator is applied, and **Prodigy4.0** shows the top-level procedure.

Planning involves specifying a task model including operators, search control rules and domain description.

2.2.1. Operators ROGUE's operators rely heavily on Xavier's existing behaviours, including path planning, navigation, vision and speech. ROGUE does not reason, for example, about which path the robot takes to reach a goal.

In Table 4, for example, the robot cannot deliver a particular item unless it (a) has the item in question, and (b) is in the correct location[1].

If any of the four preconditions are false, it will create a plan to achieve each of the preconditions. It takes the top level goal, `(has-item <user> <item>)`, and selects an operator that will achieve it. It continues building the plan recur-

Table 3. Goal selection search control rule.

```
(control-rule SELECT-TOP-PRIORITY-AND-COMPATIBLE-GOALS
   (if (and (candidate-goal <goal>)
            (or (ancestor-is-top-priority-goal <goal>)
                (compatible-with-top-priority-goal <goal>))))
   (then select goal <goal>))
```

Table 4. Item delivery operator.

```
(operator DELIVER-ITEM <room> <user> <item>
   (preconds (and (needs-item <user> <item>)
                  (robot-has-item <user> <item>)
                  (deliver-loc <user> <room>)
                  (robot-in-room <room>)))
   (effects ((add (has-item <user> <item>))
             (del (needs-item <user> <item>))
             (del (robot-has-item <user> <item>)))))
```

sively, adding operators for each precondition that is not true in the state, until all of the operators in the leaf nodes have no untrue preconditions, yielding a network of plan steps and goals such as the one shown in Figure 5.

In Table 5, we show how an operator represents that the robot will not go to a pickup location unless it needs to pickup an item there. It does

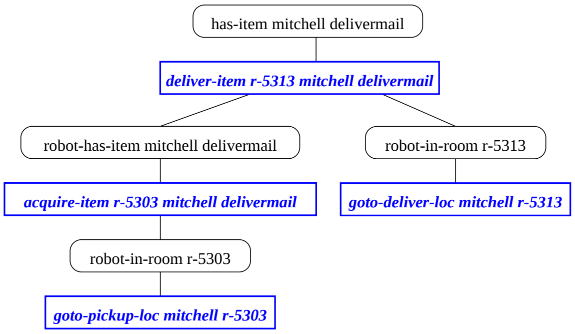

Fig. 5. Plan for a single task problem. Goal nodes are shown in ovals, selected operators are shown in rectangles.

Table 5. Goto pickup location operator.

```
(operator GOTO-PICKUP-LOC <user> <room>
   (preconds
       (and (needs-item <who> <item>)
            (not (robot-has-item <who> <item>))
            (pickup-loc <who> <room>)))
   (effects ((<old-room> ROOM))
            ((del (robot-in-room <old-room>))
             (add (robot-in-room <room>)))))
```

not matter where the robot's current location is; the variable `<old-room>` is only instantiated when the robot arrives at the goal location[2].

Other operators in the domain represent different task actions. By abstracting each request to the robot, such as which path the robot takes, ROGUE can more fully address issues arising from multiple interacting tasks, such as efficiency, resource contention, and reliability.

2.2.2. Search Control Rules PRODIGY4.0 provides a method for creating *search control rules* that reduce the number of choices at each decision point in Table 2 by pruning the search space or suggesting a course of action while expanding the plan. In particular, control rules can select, prefer or reject a particular goal or action in a particular situation. Control rules can be used to focus planning on particular goals and towards desirable plans.

Each time PRODIGY4.0 examines the set of unsolved pending goals, it fires its search control rules to decide which goal to expand. ROGUE interacts with PRODIGY4.0 by providing the set of control rules used to constrain PRODIGY4.0's decisions. Table 3 shows ROGUE's goal selection control rule that calls two functions, forcing PRODIGY4.0 to select the goals with high priority as well as the goals that can be opportunistically achieved (without compromising the main high-priority goal).

The function `(ancestor-is-top-priority-goal)` calculates whether the goal is required to solve a high-priority goal. ROGUE prioritizes goals according to a modifiable metric. For example, in the current implementation, this metric involves looking at the user's position in the department, the type of request and the deadline: $Priority = PersonRank + TaskRank + DeadlineRank$, where $DeadlineRank$ is defined as shown in Figure 6. (When the deadline is reached, the goal is removed from PRODIGY4.0's

$$DeadlineRank = \begin{cases} \frac{R_{max}}{t_1 - t_0} \times (t - t_0) & t_0 \leq t \leq t_1 \\ 0 & \text{otherwise} \end{cases}$$

Where: t is the current time
R_{max} is the maximum possible rank value
t_1 = deadline − expected_execution_time
t_0 = deadline − 2× expected_execution_time

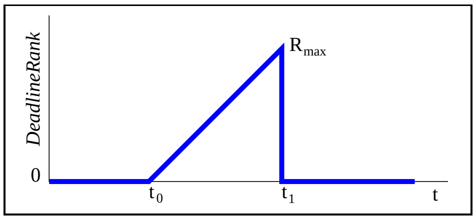

Fig. 6. Calculating the priority rank of the deadline.

pending goals list; otherwise even a task of priority 0 would eventually be attempted after all other pending tasks have been completed.) This function could easily be replaced with alternatives (e.g. [Williamson & Hanks, 1994]).

The function `(compatible-with-top-priority-goal)` allows ROGUE to determine when different goals have similar features so that it can opportunistically achieve lower priority goals while achieving higher priority ones. For example, if multiple people whose offices are all in the same hallway asked for their mail to be picked up and brought to them, ROGUE would do all the requests in the same episode, rather than only bringing the mail for the most important person. Compatibility is defined by physical proximity (*"on the path of"*) with a fixed threshold for being too far out of the way.

It is possible that these rules will select too many compatible tasks, become "side-tracked," and therefore fail on the high-priority task. A preset threshold would serve as a pragmatic solution to this problem. We also do not deal with the issue of thrashing, i.e., receiving successively more important tasks resulting in no forward progress, because it has not been an issue in practice.

PRODIGY4.0 also uses a search control rule to select a good execution order of the applicable actions. It makes the choice based on an execution-driven heuristic which minimizes the expected total traveled distance from the current location.

2.3. *Suspending and Interrupting Tasks*

ROGUE needs to be able to respond quickly when new tasks arrive and also when priorities of existing tasks change. PRODIGY4.0 supports these changing objectives by making it easy to suspend and reactivate tasks.

PRODIGY4.0 grows the plan incrementally, meaning that each time it selects a goal to expand, the remainder of the plan is unaffected. The system can therefore easily suspend planning for one task while it plans for another. The planning already done for the suspended goals remains valid until PRODIGY4.0 is able to return to them.

When PRODIGY4.0 does in fact return to the suspended actions, it validates their preconditions in the state, expanding the plan if necessary, or continuing execution if appropriate.

Generally, the plans for the interrupted goals will not be affected by the planning and execution for the new goal. Occasionally, however, actions executed to achieve the new goal might undo or achieve parts of the interrupted plan. For example, the robot might have finished its new task in the pickup location of the interrupted task. There are also occasions in which exogenous events may change the state, such as if a user passed the robot in the corridor and took his mail at that time.

In cases like these, the execution monitoring algorithm will update PRODIGY4.0's state information and PRODIGY4.0 will know which preconditions it needs to re-achieve or to ignore. In Section 3.2 we discuss in more detail how side-effects of actions and exogenous events may affect interrupted or pending plans.

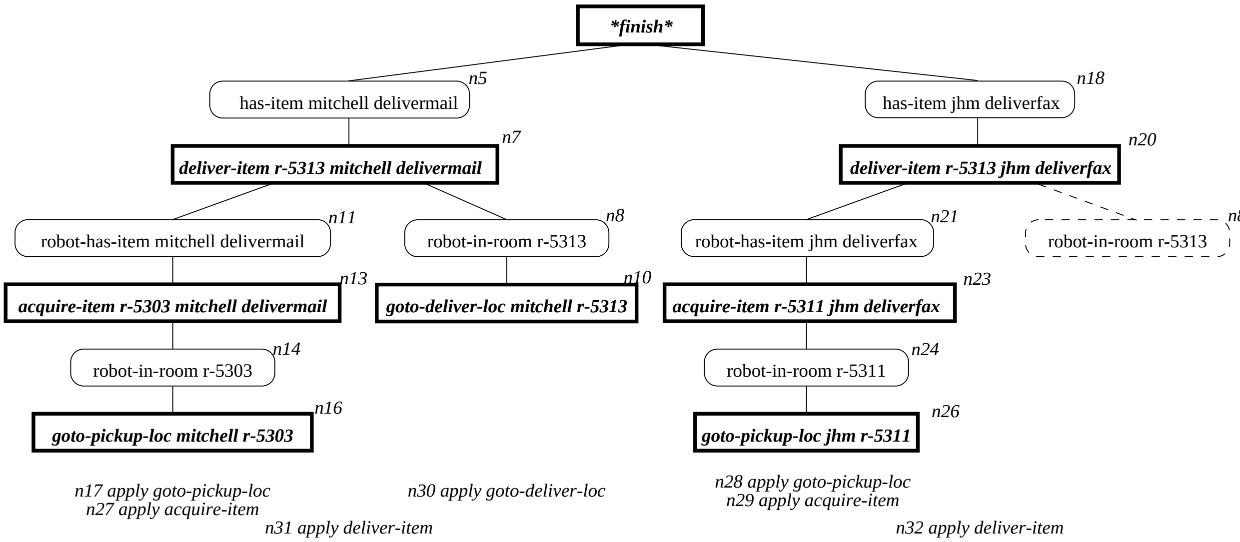

Fig. 7. Plan for a two-task problem; goal nodes are in ovals, required actions are in rectangles.

2.4. *Example*

We now present a detailed example of how PRODIGY4.0, ROGUE and Xavier interact in a two goal problem: `(has-item mitchell delivermail)` and `(has-item jhm deliverfax)`. The second (higher priority) goal arrives while ROGUE is executing the first action for the first goal.

Figure 7 shows the planning graph generated by PRODIGY4.0. We describe below the details of how it is generated. This example shows:

- how PRODIGY4.0 expands tasks,
- how search control rules affect PRODIGY4.0's selections, and
- how an asynchronous task request affects the plan.

We assume for the purposes of this example that no failures occur during execution. The example is perhaps overly detailed for a reader familiar with back-chaining planners; those readers could skip to the next section without loss of continuity.

We show the algorithmic sequence of steps of PRODIGY4.0. At each step, we show the lists of pending goals, *PG*, applicable operators, *Applicable-Ops*, and executed operators, *Executed-Ops*[3].

---

1. Request `(has-item mitchell delivermail)` arrives. ROGUE adds this goal to PRODIGY4.0's pending goals list, *PG*, and adds the following knowledge to PRODIGY4.0's state information:

```
(needs-item mitchell delivermail)
(pickup-loc mitchell r-5303)
(deliver-loc mitchell r-5313)
```

PG is `(has-item mitchell delivermail)`
Applicable-Ops is `nil`
Executed-Ops is `nil`

2. PRODIGY4.0 fires its goal-selection search control rules, which selects this goal (node 5 of Figure 7) as the highest priority goal (since it is the only choice). PRODIGY4.0 examines this goal to find an appropriate operator. It finds (DELIVER-ITEM <user> <room> <object>), and instantiates the variables: <user> := mitchell, <room> := r-5313 and <object> := delivermail, yielding the instantiated operator shown in node 7 of Figure 7. Using means-ends analysis, PRODIGY4.0 identifies two preconditions not satisfied in the state: (robot-has-item mitchell delivermail) and (robot-in-room r-5313). PRODIGY4.0 adds these preconditions to the pending goals list.

   PG is (and (robot-has-item mitchell delivermail)
   (robot-in-room r-5313))
   Applicable-Ops is nil
   Executed-Ops is nil

3. PRODIGY4.0 continues expanding the plan for this task, yielding nodes 5 through 16. At this moment, two operators in the plan have all their preconditions met in the current state.

   PG is nil
   Applicable-Ops is (and (GOTO-DELIVER-LOC mitchell r-5313)
   (GOTO-PICKUP-LOC mitchell r-5303))
   Executed-Ops is nil

4. PRODIGY4.0 examines the set of *Applicable-Ops*, and based on ordering constraints (goal clobbering), selects (GOTO-PICKUP-LOC mitchell r-5303) to apply. ROGUE takes the applied operator, (GOTO-PICKUP-LOC mitchell r-5303) (node 17), and sends it to the robot for execution. (It does not need to verify preconditions in the real world since none can be changed by exogenous events.)

   PG is nil
   Applicable-Ops is (GOTO-DELIVER-LOC mitchell r-5313)
   Executed-Ops is nil

5. Request (has-item jhm deliverfax) arrives. ROGUE adds this goal to *PG*.
   ROGUE does not interfere with the currently executing action, namely (GOTO-PICKUP-LOC mitchell r-5303). Goodwin [1994] discusses methods to decide when to interfere.

   PG is (has-item jhm deliverfax)
   Applicable-Ops is (GOTO-DELIVER-LOC mitchell r-5313)
   Executed-Ops is nil

6. The navigation module finally indicates completion of the action. ROGUE verifies the outcome (postconditions) of the action, i.e., that it has arrived at the location r-5313 (see Section 3 for a description). Now the action (ACQUIRE-ITEM r-5303 mitchell delivermail) is applicable.

   PG is (has-item jhm deliverfax)
   Applicable-Ops is (and (GOTO-DELIVER-LOC mitchell r-5313)
   (ACQUIRE-ITEM r-5303 mitchell delivermail))
   Executed-Ops is (GOTO-PICKUP-LOC mitchell r-5303)

7. PRODIGY4.0 fires ROGUE's search control rules, which select the new goal (since it is higher priority than the current task) (node 18). It expands the plan as above, except that instead of selecting a new operator to achieve the goal (robot-in-room r-5313), it notices that the operator

(GOTO-DELIVER-LOC mitchell r-5313) has the same effect, and does not redundantly add a new operator.

```
PG is nil
Applicable-Ops is (and (GOTO-DELIVER-LOC mitchell r-5313)
                       (ACQUIRE-ITEM r-5303 mitchell delivermail)
                       (GOTO-PICKUP-LOC jhm r-5311))
Executed-Ops is (GOTO-PICKUP-LOC mitchell r-5303)
```

8. PRODIGY4.0 selects (ACQUIRE-ITEM r-5303 mitchell delivermail) to apply (node 27); ROGUE verifies its preconditions and then sends it to the robot for execution. When the action is complete, ROGUE verifies the postconditions (i.e. that it now has mitchell's mail).

```
PG is nil
Applicable-Ops is (and (GOTO-DELIVER-LOC mitchell r-5313)
                       (GOTO-PICKUP-LOC jhm r-5311))
Executed-Ops is (GOTO-PICKUP-LOC mitchell r-5303)
                (ACQUIRE-ITEM r-5303 mitchell deliver-mail)
```

9. The execution constraint control rule now selects (GOTO-PICKUP-LOC jhm r-5311) as the next applicable operator (node 28). ROGUE sends it to Xavier for execution and monitors its outcome.

```
PG is nil
Applicable-Ops is (and (GOTO-DELIVER-LOC mitchell r-5313)
                       (ACQUIRE-ITEM r-5311 jhm deliverfax))
Executed-Ops is (GOTO-PICKUP-LOC mitchell r-5303)
                (ACQUIRE-ITEM r-5303 mitchell deliver-mail)
                (GOTO-PICKUP-LOC jhm r-5311)
```

10. ROGUE then acquires the fax.
11. ROGUE then goes to room 5313.
12. ROGUE delivers both items.

The final execution order described in this example is shown in Figure 8. This example illustrates the asynchronous handling of goals in ROGUE.

3. Execution and Monitoring

In this section we describe how ROGUE mediates the interaction between the planner and the robot. We show how symbolic action descriptions are turned into robot commands, as well as how robot sensor data is incorporated into the planner's knowledge base so that the planner can compensate for changes in the environment or unexpected failures of its actions.

The key to this communication model is based on a pre-defined language and model translation between PRODIGY4.0 and Xavier. The procedures to do this translation are manually generated, but are in a systematic format and may be extended at any time to augment the actions or sensing capabilities of the system. It is an open problem to automate the generation of these procedures because it is not only challenging to select what features of the world may be relevant for replan-

```
Solution:
    <GOTO-PICKUP-LOC mitchell r-5303>
  [arrival of second request]
    <ACQUIRE-ITEM r-5303 mitchell delivermail>
    <GOTO-PICKUP-LOC jhm r-5311>
    <ACQUIRE-ITEM r-5311 jhm deliverfax>
    <GOTO-DELIVER-LOC mitchell r-5313>
    <DELIVER-ITEM r-5313 jhm deliverfax>
    <DELIVER-ITEM r-5313 mitchell delivermail>
```

Fig. 8. Final execution sequence.

ning, but also how to detect those features using existing sensors.

3.1. Executing Actions

Each action that PRODIGY4.0 selects must be translated into a form that Xavier will understand. ROGUE translates the high-level abstract action into a command sequence appropriate for execution.

PRODIGY4.0 allows arbitrary procedural attachments that are called during the operator application phase of the planning cycle [Stone & Veloso, 1996]. Typically, we use these functions to give the planner additional information about the state of the world that might not be accurately predictable from the model of the environment. For example, this new information might show resource consumption or action outcomes.

ROGUE extends this information-gathering capability because, instead of *simulating* operator effects, ROGUE actually sends the commands to the robot for *real world* execution. Actually executing the planner's actions in this way increases system reliability and efficiency because the system can respond quickly to unexpected events such as failures and side effects, and combined planning and execution effort is reduced since actions are interleaved, and the planner knows the exact outcome of uncertain events.

In general, these procedures:

1. verify the preconditions of the operator,
2. execute the associated actions, and
3. verify the postconditions of the operator.

Some of these procedures also contain simple failure recovery procedures, particularly for actions that have common and known failures. For example, an action might simply be repeated, as in the `navigateToGoal` command. These procedures resemble schemas [Georgeff & Ingrand, 1989, Hormann, Meier, & Schloen, 1991] or RAPs [Firby, 1989, Gat, 1992, Pell *et al.*, 1997], in that they specify how to execute the action, what to monitor in the environment, and some recovery procedures. ROGUE's procedures, however, do not contain complex recovery or monitoring procedures, such as when they have different costs or probabilities, since we feel that it is more appropriate for the planner to reason about when they should be used.

These command sequences may be executed directly by ROGUE (e.g. a command like `finger` to determine an office location), or sent via the TCA interface to the Xavier module designed to handle the command. The action `(ACQUIRE-ITEM <room> <user> <item>)`, for example, is mapped to a sequence of commands that allows the robot to interact with a human. The action `(GOTO-PICKUP-LOC <user> <room>)` is mapped to the commands shown in Table 6, extracted from an actual trace: (1) Announce intended action, (2) Ask Xavier's path planner to find the coordinates of a door near the room, (3) Navigate to those coordinates, and (4) Verify the outcome.

In the following section, we explain in more detail how ROGUE monitors the outcome of the action, and how failures may cause replanning or affect plans of interrupted tasks.

3.2. Monitoring

There are two types of events that ROGUE needs to monitor in the environment.

The first centers around *actions*. Each time ROGUE executes an action, it needs to verify its outcome because actions may have multiple outcomes or fail unexpectedly. ROGUE may need to invoke replanning, or select actions at a branching condition. ROGUE also needs to verify the preconditions of an action before executing it because the world may change, invalidating one of the system's beliefs. ROGUE uses a layered verification process, incrementally calling methods with greater cost and accuracy, until a predefined confidence

Table 6. The set of actions taken for executing the PRODIGY4.0 operator <GOTO-DELIVER-LOC mitchell r-5309>.

```
<GOTO-DELIVER-LOC MITCHELL R-5309>

SENDING COMMAND (tcaExecuteCommand "C_say" "Going to room 5309")
ANNOUNCING: Going to room 5309
SENDING COMMAND (tcaQuery "nearRoomQ" "5309")
   ...Query returned #(TASK-CONTROL::NEARROOMREPLY 567.0d0 3483.0d0 90.0d0)
SENDING COMMAND (tcaExpandGoal "navigateToG" #(TASK-CONTROL::MAPLOCDATA 567.0d0 3483.0d0))
   ...waiting...
   ...Action NAVIGATE-TO-GOAL finished (SUCCESS).
SENDING COMMAND (tcaQuery "visionWhereAmI")
   ...Query returned #(TASK-CONTROL::VISIONWHEREAMI "5309")
```

threshold is reached. Action monitors are invoked only when the action is executed.

The second centers around exogenous events in the *environment*. Certain events may cause changes in the environment that affect current goals, or opportunities may arise that ROGUE can take advantage of. For example, ROGUE can monitor battery power, or examine camera images for open doors or particular objects. Environment monitors are invoked when relevant goals are introduced to the system.

Both types of monitoring procedures specify (1) what to monitor and (2) the methods that can be used to monitor it. *Action monitors* monitor the preconditions and effects of the action, while *environment monitors* are determined by the programmer. The action monitors, based on the planning domain model, provide a focus for execution monitoring. It is an open problem to autonomously decide what exogenous events to monitor that will be relevant for planning.

Although action monitoring is sequential and of limited time-span, while environment monitoring is parallel and continuous, the two sets of procedures have similar effects on planning.

Once ROGUE has done the required monitoring, ROGUE needs to update PRODIGY4.0's state description as appropriate. In execution monitoring, the update occurs when the object is detected, or when battery power falls below a certain threshold. In action monitoring, the critical update is when the *actual* outcome of the monitoring does not meet the *expected* outcome. These updates will force PRODIGY4.0 to re-examine its plan, adding or discarding operators as necessary.

If the primary effect of the action has been unexpectedly satisfied, ROGUE adds the knowledge to PRODIGY4.0's state description and PRODIGY4.0 does not attempt to achieve it. Observing the environment and maintaining a state description in this way improves the efficiency of the system because it will not attempt redundant actions.

If a required precondition is no longer true as a side-effect of some other action or environment monitoring, ROGUE deletes the relevant precondition from PRODIGY4.0's state. PRODIGY4.0 will therefore replan in an attempt to find an action that will re-achieve it.

In action monitoring, if the action fails, ROGUE will first try the built-in recovery methods. These recovery methods are very simple; more complex ones are treated as separate operators for PRODIGY4.0 to reason about. For example, ROGUE will try calling the navigation routine a predefined number of times before deciding that the action completely failed. At the scene of a pickup or delivery, if ROGUE times-out while waiting for a response to a query, ROGUE will prompt for a user a second time before failing. If, despite the built-in recovery methods, ROGUE determines that the action has completely failed, ROGUE will delete the effect from PRODIGY4.0's state description, and PRODIGY4.0 will replan to achieve it.

Occasionally during environment monitoring, knowledge will unexpectedly be added to the state that causes an action to become executable, or a task to become higher priority. Each time PRODIGY4.0 makes a decision, it re-examines all of its options, and will factor the new action or goal into the process.

In this manner, ROGUE is able to detect execution failures and compensate for them, as well as to respond to changes in the environment. The interleaving of planning and execution reduces the need for replanning during the execution phase and increases the likelihood of overall plan success because the planner is constantly being updated with information about changes in the world. It allows the system to adapt to a changing environment where failures can occur.

3.3. Example of how ROGUE Handles Failures

One of Xavier's actions that ROGUE monitors is the `navigateToG`oal command, used by both the `(GOTO-PICKUP-LOC <user> <room>)` and the `(GOTO-DELIVER-LOC <user> <room>)` operators. `navigateToG` reports a success when the robot arrives at the requested goal. `navigateToG` may fail under several conditions, including detecting a bump, corridor or door blockage, or lack of forward progress. The module is able to autonomously compensate for certain problems, such as obstacles and missing landmarks. Navigation is done using Partially Observable Markov Decision Process models [Simmons & Koenig, 1995], and the inherent uncertainty of this probabilistic model means that the module may occasionally report success even when it has not actually arrived at the desired goal location.

When `navigateToG` reports a failure or a low-probability success, ROGUE verifies the location. ROGUE first tries to verify the location autonomously, using its cameras. The vision module looks for a door in the general area of the expected door, and finds the room label, and reads it. If this module fails to find a door, fails to find a label, or returns low confidence in its template matching, ROGUE falls back to a second verification procedure, namely using the speech module to ask a human. We assume that verification step gives complete and correct information about the robot's actual location; other researchers are focussing on the open problem of sensor reliability [Hughes & Ranganathan, 1994, Thrun, 1996].

If ROGUE detects that in fact the robot is *not* at the correct goal location, ROGUE updates the navigation module with the new information and re-attempts to navigate to the desired location. If the robot is still not at the correct location after a constant number of tries (three in our current implementation), ROGUE updates PRODIGY4.0's task knowledge to reflect the robot's actual position, rather than the expected position.

In general, PRODIGY4.0 has several different operators that can achieved a particular effect, and will successfully replan for the failure. In this case, however, there are no other alternative methods of navigating, and PRODIGY4.0 declares that the task can not be successfully achieved.

3.4. Example of how ROGUE Handles Side-effects

Occasionally, suspending one task for a second one will mean that work done for the first will be undone by work done for the second. ROGUE needs to detect these situations and plan to re-achieve the undone work. Consider a simple situation that illustrates this re-planning process:

Task one:	**Task two:**
1a. goto 5301	2a. goto 5409
1b. pick up mail	2b. pick up fed-ex package
1c. goto 5315	2c. goto 4320
1d. drop off mail	2d. drop package off

Many possible interleaved planning and execution scenarios may occur; below are two possibilities.

- **[Normal:]** Executes 1a and 1b. While executing, the request for two arrives. ROGUE decides that task two is more important. Task two is suspended; step 1c is pending. Plans for and executes task two. Returns to step 1c, verifies that it is still needed to complete the task and can still be done, then does 1c and 1d.
- **[Undone Action:]** Executes 1a. While executing 1b, the request for task two arrives. 1b times-out, indicating that the mail-room person wasn't there to give the robot the

mail. ROGUE decides task two is more important, and suspends task one; step 1b is pending. ROGUE plans for and executes task two. PRODIGY4.0 returns to step 1b, discovers that a precondition is not true: `(robot-in-room <5301>)`. PRODIGY re-plans to achieve it, and then re-executes step 1a, and then finishes the task as expected.

4. Conclusion

ROGUE is fully implemented and operational. The system completes all requested tasks, running errands between offices in our building. Execution results are presented in detail elsewhere [Simmons *et al.*, 1997].

We have presented one aspect of ROGUE, an integrated planning and execution robot architecture. We have described how PRODIGY4.0 gives ROGUE the power

- to integrate asynchronous requests,
- to prioritize goals,
- to suspend and reactivate tasks,
- to recognize compatible tasks and opportunistically achieve them,
- to execute actions in the real world, integrating new knowledge which may help planning, and
- to monitor and recover from failure.

ROGUE represents a successful integration of a classical artificial intelligence planner with a real mobile robot. The complete planning and execution cycle for a *single* task can be summarized as follows:

1. ROGUE receives a task request from a user.
2. ROGUE requests a plan from PRODIGY4.0.
3. PRODIGY4.0 generates a plan and passes executable steps to ROGUE.
4. ROGUE translates and sends the planning steps to Xavier.
5. ROGUE monitors execution and identifies goal status; in case of failure, PRODIGY4.0's state information domain modified and PRODIGY4.0 will replan for decisions.

ROGUE handles multiple goals, interleaving the individual plans to maximize overall execution efficiency.

Figure 9 summarizes the information exchanged between the user, PRODIGY4.0, and Xavier under ROGUE's mediation. ROGUE constrains PRODIGY4.0's decisions through calculations on task priority, task compatibility, and execution efficiency. ROGUE translates PRODIGY4.0's symbolic action descriptions into Xavier commands, and also translates Xavier's perception information into PRODIGY4.0 domain description.

The contributions of our work to the Xavier project are in the high-level reasoning parts of the system, allowing the robot to efficiently handle multiple, asynchronous interacting goals, and to effectively interleave planning and execution in a real world system. Execution monitoring based on a planning model allows the systematic identification of environment monitors.

Interleaving planning with execution enhances a deliberative robot system in numerous ways. One such benefit is that the system can sense the world to acquire necessary domain knowledge in order to continue planning. For example, it can ask directions, look to see if doors are open or closed, or check whether it needs to recharge its batteries. Another benefit is reduced planning effort because the system does not need to plan for all possible failure contingencies; instead, it can execute an action to find out its actual outcome.

ROGUE advances the state of the art of the integration of planning and execution in robotic agent. In a unique novel way, ROGUE is designed as the integration of two independently developed platforms. PRODIGY4.0 is a general-purpose planner and Xavier can be viewed as a general-purpose navigational robot. ROGUE merges the functionality of these two systems in a real implementation that demonstrates the feasibility of connecting both systems in a rich task environment, namely the achievement of asynchronous user requests. (ROGUE therefore also shows how the PRODIGY4.0 planner and the TCA approach in Xavier are in fact robust architectures.)

Strictly looking at ROGUE only from the viewpoint of the integration of planning and execution, ROGUE compares well with other special-purpose systems such as NMRA and 3_T. Given the general-purpose character of the PRODIGY4.0

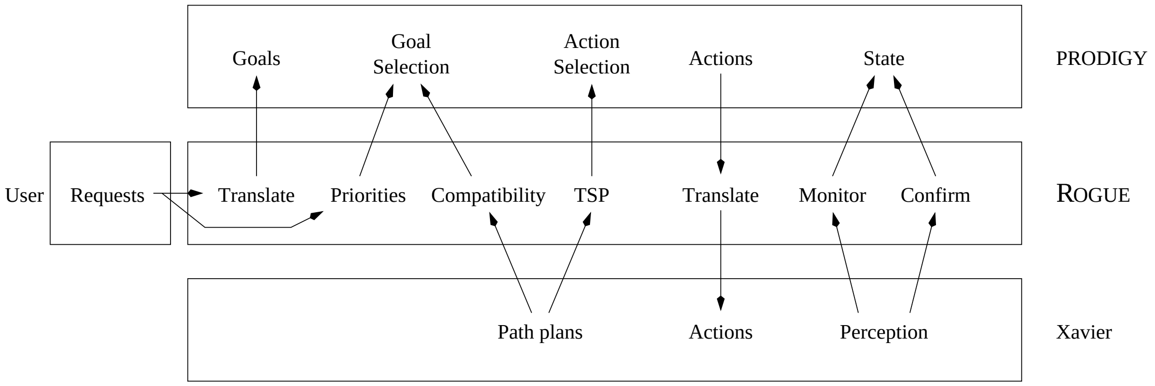

Fig. 9. What ROGUE does.

planner, ROGUE could easily be applied to other executing platforms and tasks by a flexible change of PRODIGY4.0's specification of the domain.

The goal of our research is to build a complete planning, executing and learning autonomous robotic agent. ROGUE's contributions go beyond the integration of planning and execution. ROGUE incorporates learning from execution experience [Haigh & Veloso, 1998]. The learning algorithm involves extracting relevant information from real execution traces in order to detect patterns in the environment to improve the robot's behaviour. The ability to learn task-relevant knowledge conveniently matches PRODIGY4.0's search control representation, and learned situational-dependent arc costs can be incorporated into Xavier's route planner knowledge. Through its interleaved planning and execution behaviour, ROGUE provides an appropriate platform to collect the required execution data for learning.

Acknowledgements

The authors would like to thank Eugene Fink, Sven Koenig, Illah Nourbakhsh, Joseph O'Sullivan, Gary Pelton and the anonymous reviewers for feedback on this article. We would also like to thank the members of the Xavier and PRODIGY groups for feedback, comments and criticism on our research.

This research is sponsored in part by (1) the National Science Foundation under Grant No. IRI-9502548, (2) by the Defense Advanced Research Projects Agency (DARPA), and Rome Laboratory, Air Force Materiel Command, USAF, under agreement number F30602-95-1-0018, (3) the Natural Sciences and Engineering Council of Canada (NSERC), and (4) the Canadian Space Agency (CSA). The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the NSF, DARPA, Rome Laboratory, the U.S. Government, NSERC or the CSA.

Notes

1. The literal (`has-item`) is strictly for bookkeeping (to keep track of what tasks have been completed), and is not used for planning in any way.
2. The representation of the operators, for example (`GOTO-PICKUP-LOC`) and (`GOTO-DELIVER-LOC`), is not intrinsic to the task, but it can be relevant to planning efficiency. We have an implementation of the domain with a single (`GOTO-LOC`) operator with less constrained preconditions, which leads to more backtracking while the planner selects the correct order of desired locations. We can also create a search control rule to guide the planning choices; this is logically equivalent to separating the operators, but with some additional match cost.
3. For readers more familiar with PRODIGY literature, the *Executed-Ops* correspond to PRODIGY4.0's *head-plan*, while the plan shown in Figure 7 corresponds to PRODIGY4.0's *tail-plan*.

References

[Ambros-Ingerson & Steel, 1988] Ambros-Ingerson, J. A., and Steel, S. (1988). Integrating planning, execution and monitoring. In *Proceedings of the Seventh National Conference on Artificial Intelligence (AAAI-88)*. St.

Paul, MN, (Menlo Park, CA: AAAI Press), pp. 83–88.

[Atkins, Durfee, & Shin, 1996] Atkins, E. M.; Durfee, E. H.; and Shin, K. G. (1996). Detecting and reacting to unplanned-for world states. In *Papers from the 1996 AAAI Fall Symposium "Plan Execution: Problems and Issues"*. Boston, MA, (Menlo Park, CA: AAAI Press), pp. 1–7.

[Blythe, 1994] Blythe, J. (1994). Planning with external events. In *Proceedings of the Tenth Conference on Uncertainty in Artificial Intelligence*. Seattle, WA, (San Mateo, CA: Morgan Kaufmann), pp. 94–101.

[Bonasso & Kortenkamp, 1996] Bonasso, R. P., and Kortenkamp, D. (1996). Using a layered control architecture to alleviate planning with incomplete information. In *Proceedings of the AAAI Spring Symposium "Planning with Incomplete Information for Robot Problems"*. Stanford, CA, (Menlo Park, CA: AAAI Press), pp. 1–4.

[Carbonell, Knoblock, & Minton, 1990] Carbonell, J. G.; Knoblock, C. A.; and Minton, S. (1990). Prodigy: An integrated architecture for planning and learning. In VanLehn, K. (ed.), *Architectures for Intelligence*. (Erlbaum: Hillsdale, NJ). Also available as Technical Report CMU-CS-89-189, Computer Science Department, Carnegie Mellon University, Pittsburgh, PA.

[Dean & Boddy, 1988] Dean, T. L., and Boddy, M. (1988). An analysis of time-dependent planning. In *Proceedings of the Seventh National Conference on Artificial Intelligence (AAAI-88)*. St. Paul, MN, (Menlo Park, CA: AAAI Press), pp. 49–54.

[Dean *et al.*, 1990] Dean, T.; Basye, K.; Chekaluk, R.; Hyun, S.; Lejter, M.; and Randazza, M. (1990). Coping with uncertainty in a control system for navigation and exploration. In *Proceedings of the Eigth National Conference on Artificial Intelligence (AAAI-90)*. Boston, MA, (Cambridge, MA: MIT Press), pp. 1010–1015.

[DellaFera *et al.*, 1988] DellaFera, C. A.; Eichin, M. W.; French, R. S.; Jedlinsky, D. C.; Kohl, J. T.; and Sommerfeld, W. E. (1988). The Zephyr notification service. In *Proceedings of the USENIX Winter Conference*. Dallas, TX, (Berkeley, CA: USENIX Association), pp. 213–219.

[Drummond *et al.*, 1993] Drummond, M.; Swanson, K.; Bresina, J.; and Levinson, R. (1993). Reaction-first search. In *Proceedings of the Thirteenth International Joint Conference on Artificial Intelligence (IJCAI-93)*. (San Mateo, CA: Morgan Kaufmann), pp. 1408–1414.

[Fikes, Hart, & Nilsson, 1972] Fikes, R. E.; Hart, P. E.; and Nilsson, N. J. (1972). Learning and executing generalized robot plans. *Artificial Intelligence*, **3(4)**:231–249.

[Firby, 1989] Firby, R. J. (1989). *Adaptive Execution in Complex Dynamic Worlds*. Ph.D. Dissertation, Yale University, New Haven, CT.

[Gat, 1992] Gat, E. (1992). Integrating planning and reacting in a heterogeneous asynchronous architecture for controlling real-world mobile robots. In *Proceedings of the Tenth National Conference on Artificial Intelligence (AAAI-92)*. pp. 809–815.

[Georgeff & Ingrand, 1989] Georgeff, M. P., and Ingrand, F. F. (1989). Decision-making in an embedded reasoning system. In *Proceedings of the Eleventh International Joint Conference on Artificial Intelligence (IJCAI-89)*. Detroit, MI, (San Mateo, CA: Morgan Kaufmann), pp. 972–978.

[Gervasio & DeJong, 1991] Gervasio, M. T., and DeJong, G. F. (1991). Learning probably completable plans. Technical Report UIUCDCS-R-91-1686, University of Illinois at Urbana-Champaign, IL, Urbana, IL.

[Goodwin & Simmons, 1992] Goodwin, R., and Simmons, R. G. (1992). Rational handling of multiple goals for mobile robots. In Hendler, J. (ed.), *Artificial Intelligence Planning Systems: Proceedings of the First International Conference (AIPS-92)*. College Park, MD, (San Mateo, CA: Morgan Kaufmann), pp. 86–91.

[Goodwin, 1994] Goodwin, R. (1994). Reasoning about when to start acting. In Hammond, K. (ed.), *Artificial Intelligence Planning Systems: Proceedings of the Second International Conference (AIPS-94)*. Chicago, IL, (Menlo Park, CA: AAAI Press), pp. 86–91.

[Haigh & Veloso, 1996] Haigh, K. Z., and Veloso, M. (1996). Interleaving planning and robot execution for asynchronous user requests. In *Proceedings of the International Conference on Intelligent Robots and Systems (IROS)*. Osaka, Japan, (New York, NY: IEEE Press), pp. 148–155.

[Haigh & Veloso, 1997] Haigh, K. Z., and Veloso, M. M. (1997). High-level planning and low-level execution: Towards a complete robotic agent. In Johnson, W. L. (ed.), *Proceedings of the First International Conference on Autonomous Agents*. Marina del Rey, CA, (New York, NY: ACM Press), pp. 363–370.

[Haigh & Veloso, 1998] Haigh, K. Z., and Veloso, M. M. (1998). Learning situation-dependent costs: Improving planning from probabilistic robot execution. In Sycara, K. P. (ed.), *Proceedings of the Second International Conference on Autonomous Agents*. Minneapolis, MN, (Menlo Park, CA: AAAI Press). Submission.

[Hormann, Meier, & Schloen, 1991] Hormann, A.; Meier, W.; and Schloen, J. (1991). A control architecture for and advanced fault-tolerant robot system. *Robotics and Autonomous Systems*, **7**(2-3):211–225.

[Hughes & Ranganathan, 1994] Hughes, K., and Ranganathan, N. (1994). Modeling sensor confidence for sensor integration tasks. *International Journal of Pattern Recognition and Artificial Intelligence*, **8**(6):1301–1318.

[Kushmerick, Hanks, & Weld, 1993] Kushmerick, N.; Hanks, S.; and Weld, D. (1993). An algorithm for probabilistic planning. Technical Report 93-06-03, Department of Computer Science and Engineering, University of Washington, Seattle, WA.

[Lyons & Hendriks, 1992] Lyons, D. M., and Hendriks, A. J. (1992). A practical approach to integrating reaction and deliberation. In Hendler, J. (ed.), *Artificial Intelligence Planning Systems: Proceedings of the First International Conference (AIPS-92)*. (San Mateo, CA: Morgan Kaufmann), pp. 153–162.

[Mansell, 1993] Mansell, T. M. (1993). A method for planning given uncertain and incomplete information. In *Proceedings of the Ninth Conference on Uncertainty in Artificial Intelligence*. Washington, DC, (San Mateo, CA: Morgan Kaufmann), pp. 250–358.

[McDermott, 1992] McDermott, D. (1992). Transformational planning of reactive behavior. Technical Report YALE/CSD/RR#941, Computer Science Department, Yale University, New Haven, CT.

[Nilsson, 1984] Nilsson, N. J. (1984). Shakey the robot. Technical Report 323, AI Center, SRI International, Menlo Park, CA.

[Nourbakhsh, 1997] Nourbakhsh, I. (1997). *Interleaving Planning and Execution for Autonomous Robots.* (Dordrecht, Netherlands: Kluwer Academic). PhD thesis. Also available as technical report STAN-CS-TR-97-1593, Department of Computer Science, Stanford University, Stanford, CA.

[O'Sullivan, Haigh, & Armstrong, 1997] O'Sullivan, J.; Haigh, K. Z.; and Armstrong, G. D. (1997). *Xavier.* Carnegie Mellon University, Pittsburgh, PA. Manual, Version 0.3, unpublished internal report. Available via `http://www.cs.cmu.edu/~Xavier/`.

[Pell *et al.*, 1997] Pell, B.; Bernard, D. E.; Chien, S. A.; Gat, E.; Muscettola, N.; Nayak, P. P.; Wagner, M. D.; and Williams, B. C. (1997). An autonomous spacecraft agent prototype. In *Proceedings of the First International Conference on Autonomous Agents.* Marina del Rey, CA, (New York, NY: ACM Press), pp. 253–261.

[Pryor, 1994] Pryor, L. M. (1994). *Opportunities and Planning in an Unpredictable World.* Ph.D. Dissertation, Northwestern University, Evanston, Illinois. Available as Technical Report number 53.

[Schoppers, 1989] Schoppers, M. J. (1989). *Representation and Automatic Synthesis of Reaction Plans.* Ph.D. Dissertation, Department of Computer Science, University of Illinois, Urbana-Champaign, IL. Available as Technical Report UIUCDCS-R-89-1546.

[Simmons & Koenig, 1995] Simmons, R., and Koenig, S. (1995). Probabilistic robot navigation in partially observable environments. In *Proceedings of the Fourteenth International Joint Conference on Artificial Intelligence (IJCAI-95).* Montréal, Québec, Canada, (San Mateo, CA: Morgan Kaufmann), pp. 1080–1087.

[Simmons *et al.*, 1997] Simmons, R.; Goodwin, R.; Haigh, K. Z.; Koenig, S.; and O'Sullivan, J. (1997). A layered architecture for office delivery robots. In Johnson, W. L. (ed.), *Proceedings of the First International Conference on Autonomous Agents.* Marina del Rey, CA, (New York, NY: ACM Press), pp. 245–252.

[Simmons, 1994] Simmons, R. (1994). Structured control for autonomous robots. *IEEE Transactions on Robotics and Automation,* **10**(1):34–43.

[Stone & Veloso, 1996] Stone, P., and Veloso, M. M. (1996). User-guided interleaving of planning and execution. In *New Directions in AI Planning.* (Amsterdam, Netherlands: IOS Press). pp. 103–112.

[Thrun, 1996] Thrun, S. (1996). A Bayesian approach to landmark discovery and active perception for mobile robot navigation. Technical Report CMU-CS-96-122, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA.

[Veloso *et al.*, 1995] Veloso, M. M.; Carbonell, J.; Pérez, M. A.; Borrajo, D.; Fink, E.; and Blythe, J. (1995). Integrating planning and learning: The PRODIGY architecture. *Journal of Experimental and Theoretical Artificial Intelligence,* **7**(1):81–120.

[Williamson & Hanks, 1994] Williamson, M., and Hanks, S. (1994). Optimal planning with a goal-directed utility model. In Hammond, K. (ed.), *Artificial Intelligence Planning Systems: Proceedings of the Second International Conference (AIPS-94).* Chicago, IL, (Menlo Park, CA: AAAI Press), pp. 176–180.

Karen Zita Haigh is currently completing her Ph.D. in Computer Science at Carnegie Mellon University in Pittsburgh, Pennsylvania. Her undergraduate degree was completed in 1992 at the University of Ottawa in Ottawa, Ontario, Canada. Her thesis is a robot learning system that uses feedback from execution experience to improve efficiency of generated plans. It creates situation-dependent costs so that plans are tailored to particular situations, and is used on Xavier's route planner and in ROGUE. She also built analogical reasoning system to automatically generate high-quality routes in a city map. Her research interests include planning, machine learning, and robotics.

Manuela M. Veloso is a Finmeccanica Associate Professor in the Computer Science Department at Carnegie Mellon University. She received her Ph.D. in Computer Science from CMU in 1992. Dr. Veloso received the NSF Career Award and was the recipient of the Finmeccanica Chair in 1995. In 1997, she was awarded the Allen Newell Excellence in Research Award by the School of Computer Science at CMU. Dr. Veloso is the author of a monograph on "Planning by Analogical Reasoning." She is co-editor of two book, "Symbolic and Visual Learning" and "Topics of Case-based Reasoning." Dr. Veloso's research involves the integration of planning, execution and learning in dynamic environments, and in particular with multiple agents. She investigates memory-based machine learning techniques for the processing and reuse of problem experience.