In Proceedings of the AAAI-96 Spring Symposium "Planning with Incomplete Information for Robot Problems"

Interleaving Planning and Robot Execution for Asynchronous User Requests

Karen Zita Haigh and Manuela Veloso
School of Computer Science
Carnegie Mellon University
Pittsburgh PA 15213-3891
khaigh@cs.cmu.edu mmv@cs.cmu.edu
http://www.cs.cmu.edu/~khaigh http://www.cs.cmu.edu/~mmv

Abstract

This paper describes Rogue, an integrated planning and executing robotic agent. Rogue is designed to be a roving office gopher unit, doing tasks such as picking up & delivering mail and returning & picking up library books, in a setup where users can post tasks for the robot to do. We have been working towards the goal of building a completely autonomous agent which can learn from its experiences and improve upon its own behaviour with time. This paper describes what we have achieved to-date: (1) a system that can generate and execute plans for multiple interacting goals which arrive asynchronously and whose task structure is not known *a priori*, interrupting and suspending tasks when necessary, and (2) a system which can compensate for minor problems in its domain knowledge, monitoring execution to determine when actions did not achieve expected results, and replanning to correct failures.

1. Introduction

We have been working towards the goal of building autonomous robotic agents that are capable of planning and executing high-level tasks. Our framework consists of the integration of the Xavier robot agent and the prodigy planning system in a setup where users can post tasks for which the planner generates appropriate plans, delivers them to the robot, and monitors their execution.

Xavier is a robot developed by Reid Simmons at Carnegie Mellon [6]. One of the goals of the 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 [8]. Our on-going contribution to this ultimate goal is at the high-level reasoning of the process, allowing the robot to efficiently handle multiple interacting goals, and to learn from its experience. We are investigating techniques for the robot to autonomously perform many-step plans, and to appropriately handle asynchronous user interruptions with new task requests. We aim at developing techniques that will allow the system to use experience to improve its performance and model of the world.

Integrating planning and real execution by a robot is a complex task that we believe requires learning from prior experience to significantly improve the overall performance of the autonomous agent. Other researchers investigate the problem of interleaving planning and execution (including [1; 3; 4; 5]). We build upon this work and pursue our investigation from three particular angles: that of real execution in an autonomous agent, in addition to simulated execution, that of challenging the robot with multiple asynchronous user-defined interacting tasks, and that of interspersing execution and replanning as an additional learning experience.

In this paper, we focus on presenting our current work on the interleaving of planning and execution by a real robot within a framework with the following sources of incomplete information:

- the tasks requested by the users are not completely specified,
- the set of all the goals to be achieved is not known *a priori*,
- the domain knowledge is incompletely or incorrectly specified, and
- the execution steps sent to the robot may not be achieved as predicted.

The learning portions of the system is the focus of our future work and will not be discussed here.

The paper is organized as follows: In Section 2 we introduce the Rogue architecture, our developed integrated system. We illustrate the behaviour of Rogue for a single goal when no errors occur during execution in Section 3. We describe the behaviour of the architecture with multiple goals and simple execution errors in Section 4. Finally we provide a summary of Rogue's current capabilities in Section 5 along with a description of our future work to incorporate learning methods into the system.

2. General Architecture

Rogue[1] is the system built on top of prodigy4.0 to communicate with and to control the high-level task

[1]In keeping with the Xavier theme, Rogue is named after the "X-men" comic-book character who absorbs powers and

planning in Xavier[2]. The system allows users to post tasks for which the planner generates a plan, delivers it to the robot, and then monitors its execution. Rogue is intended to be a roving office gofer unit, and will deal with tasks such as delivering mail, picking up printouts and returning library books.

Prodigy and Xavier are linked together using the Task Control Architecture [9; 10] as shown in Figure 1. Currently, Rogue's main features are (1) the ability to receive and reason about multiple asynchronous goals, suspending and interrupting actions when necessary, and (2) the ability to reason about and correct simple execution failures.

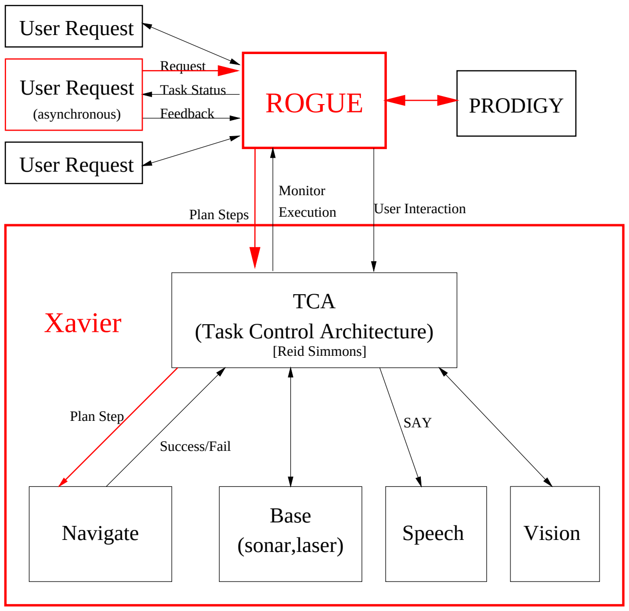


Figure 1: Rogue Architecture

Xavier

Xavier is a mobile robot being developed at CMU [6] (see Figure 2). It is built on an RWI B24 base and includes bump sensors, a laser range finder, sonars and a color camera. Control, perception and navigation planning are carried out on two on-board Intel 80486-based machines. Xavier can communicate with humans via an on-board lap-top computer or via a natural language interface; however speech recognition occurs off-board and is therefore slower.

Beyond its research abilities, Xavier can autonomously perform one of a number of simple tasks for users via it's on-line WWW page: `http://www.cs.cmu.edu/~Xavier`. To date, Xavier has been operational more than 60 hours, covering almost 20km and completing 90% of its tasks.

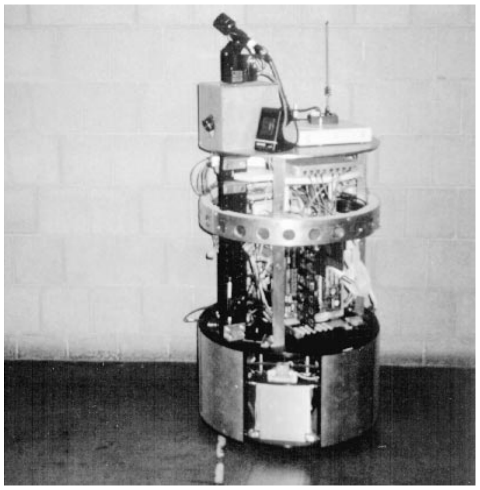


Figure 2: Xavier the robot

The software controlling Xavier includes both reactive and deliberative behaviours, integrated using the Task Control Architecture (TCA) [9; 10]. 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.

All modules and behaviours operate independently, concurrently and in a distributed manner; they can also be modified or added incrementally without affecting existing behaviours. The clear separation between reactive and deliberative behaviours increases system predictability by isolating different concerns: the robot's behaviour during normal operation is readily apparent, while strategies for handling exceptions can be individually analyzed.

Xavier has a simulator whose primary function is to test and debug code before running it on the real robot. Figure 3 shows an image of the simulator and the navigate module. The navigate module performs path planning (an A* algorithm), global navigation and position estimation. The existence of this simulator allows software to be developed, extensively tested and then debugged off-board before testing and running it on the real robot. The simulator is functionally equivalent to the real robot: it creates noisy sonar readings, it has poor dead-reckoning abilities, and it gets stuck going through doors. Most of these "problems" model the actual behaviour of the robot, allowing code developed on the simulator to run successfully on the robot with no modification. The simulator of course has limited

experience from those around her. The connotation of a wandering beggar or vagrant is also appropriate.

[2]We will use the term Xavier when referring to features specific to the robot, PRODIGY to refer to features specific to the planner, and Rogue to refer to features only seen in the combination.

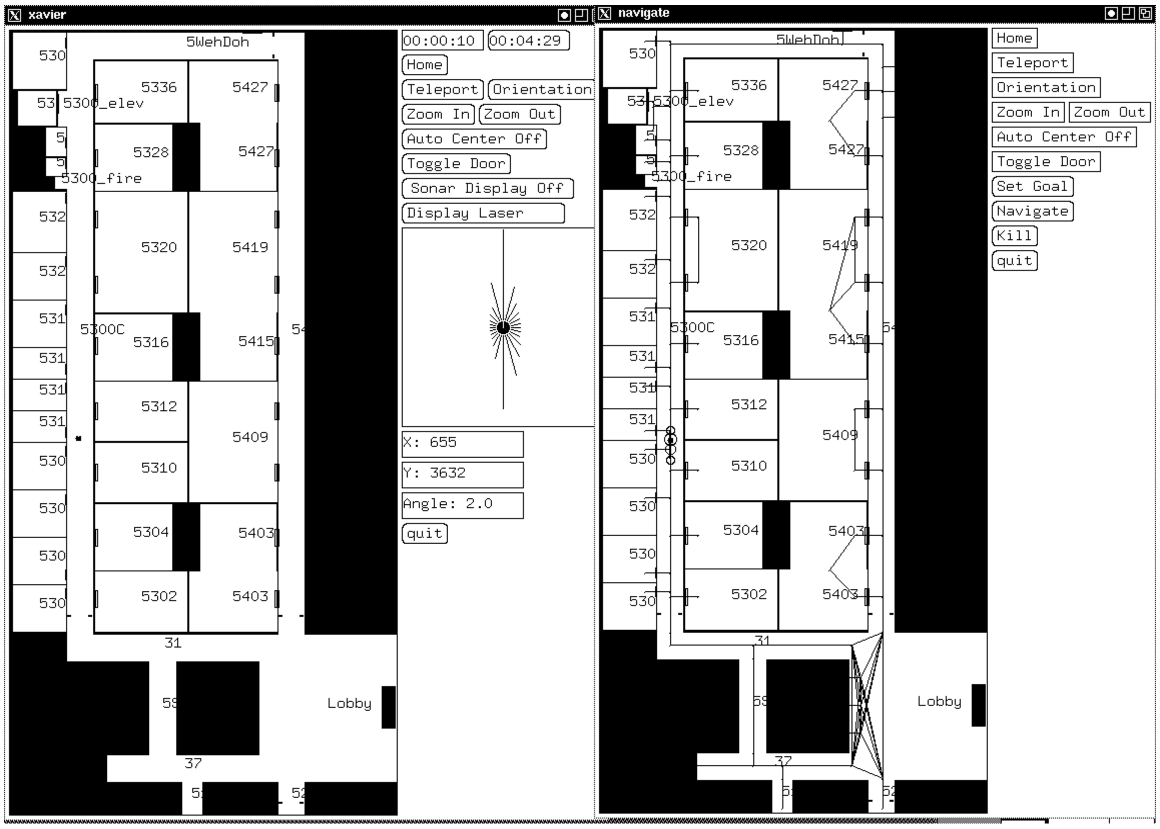


Figure 3: Simulator (left window) and Navigate (right window) in action

capabilities for dynamism: currently only doors can be opened and closed at the whim of the user. The simulator also has the ability to replay recorded data, and so actual data collected on the robot can be used by the simulator.

Prodigy

Prodigy is a domain-independent problem solver that serves as a testbed for machine learning research [2; 13]. Prodigy4.0 is a nonlinear planner that uses means-ends analysis and backward chaining to reason about multiple goals and multiple alternative operators to achieve the goals.

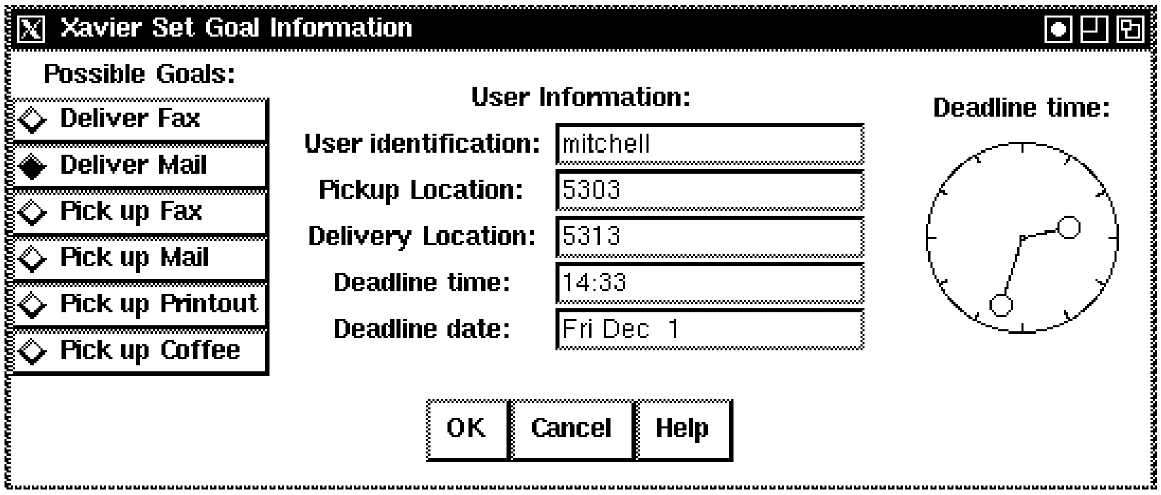
The planning reasoning cycle involves several decision points, including which goal to select from the set of pending goals, and which applicable action to execute.

Prodigy provides a method for creating *search control rules* which reduces the number of choices at each decision point by pruning the search space or suggesting a course of action. 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. Dynamic goal selection from the set of pending goals enables the planner to interleave plans, exploiting common subgoals and addressing issues of resource contention.

Prodigy maintains an internal model of the world in which it simulates the effects of selected applicable operators. Applying an operator gives the planner additional information (such as consumption of resources) that might not be accurately predictable from the domain model. Prodigy also supports real-world execution of its applicable operators when it is absolutely necessary to know the outcome of an action; for example, when actions have probabilistic outcomes, or the domain model is incomplete and it is necessary to acquire additional knowledge. During the application phase, user-defined code is called which can map the operator to a real-world action sequence [11]. Some examples of the use of this feature include shortening combined planning and execution time, acquiring necessary domain knowledge in order to continue planning (*e.g.* sensing the world), and executing an action in order to know its outcome and handle any failures.

3. Base-line Behaviour

This section describes Rogue's underlying architecture in more detail, describing the interface for users to create task requests, and then, through the use of an example, describes how the planner generates a plan to achieve the request and executes it, successfully mak-


Xavier Set Goal Information

Possible Goals:

- ◇ Deliver Fax
- ◆ Deliver Mail
- ◇ Pick up Fax
- ◇ Pick up Mail
- ◇ Pick up Printout
- ◇ Pick up Coffee

User Information:

User identification:	mitchell
Pickup Location:	5303
Delivery Location:	5313
Deadline time:	14:33
Deadline date:	Fri Dec 1

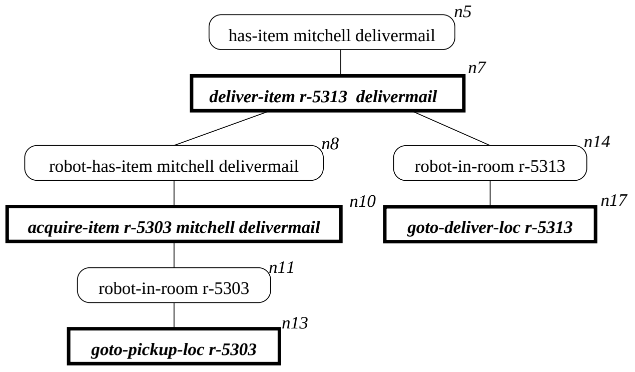
Figure 4: User Request Interface

ing an office delivery. The features described here were developed using the Xavier simulator and then tested on the actual robot.

Any user can create and send a goal request to Rogue via a simple user interface, shown in Figure 4. These requests can come in asynchronously, and include information about what item needs to be moved, where it needs to be picked up, where it needs to be delivered, and who is making the request. Rogue is able to identify and handle incomplete goal information by utilizing default values and accessing various on-line information sources (such as `finger`), or requesting them from the user.

Consider a simple problem where a single request is made: the request is from Figure 4 where the user `mitchell` would like his mail taken from room 5303 to room 5313. Figure 5 shows the search tree generated by prodigy. Figure 6 shows a detailed trace of the complete interaction.

When the request arrives at the Rogue module, Rogue translates it into a prodigy state and goal description and then spawns a prodigy run. Prodigy uses its domain knowledge to create a series of actions that will achieve the goal. When the planner has mapped out the plan with enough detail to know its first action (node 18), it informs Rogue, which sends a command to Xavier who starts executing the plan. The first action can be determined when prodigy knows that the step will be useful in achieving the goal, and will not be dis-achieved by another action. There are four actions that need to be executed in order to achieve the goal, namely *deliver-item* (node 7), *acquire-item* (node 10), *goto-pickup-loc* (node 13), and *goto-deliver-loc* (node 17). The structure of the goal tree indicates that nodes 7 and 10 should be executed after nodes 13 and 17. Simple reasoning shows that achieving node 17 would be pointless since the action would be immediately undone. As a result, Rogue starts to execute


has-item mitchell delivermail
n5
deliver-item r-5313 delivermail
n7
robot-has-item mitchell delivermail
n8
robot-in-room r-5313
n14
acquire-item r-5303 mitchell delivermail
n10
goto-deliver-loc r-5313
n17
robot-in-room r-5303
n11
goto-pickup-loc r-5303
n13

```
Solution:
<goto-pickup-loc mitchell r-5303>
<acquire-item r-5303 mitchell delivermail>
<goto-deliver-loc mitchell r-5313>
<deliver-item r-5313 mitchell delivermail>
```

Figure 5: Search Tree and Solution for single task problem; goal nodes in ovals, executed actions in rectangles.

(goto-pickup-loc). The solution shown in Figure 5 shows the complete ordering of the executed actions.

Each of the actions described in the domain model is mapped to a command sequence suitable for Xavier. These commands are executed in the real-world during the operator application phase of prodigy, as described above. They may be executed directly by the Rogue module (*e.g.* an action like `finger`), or sent via the TCA interface to the Xavier module designed to handle the command. For example, the action *(goto-pickup-loc room)* is mapped to the commands (1) find out the coordinates of the room, and (2) navigate to those coordinates. Each line marked `SENDING COMMAND` (from Figure 6) indicates a direct command sent through the TCA interface to one of Xavier's modules. The command `navigateToG`oal creates a (shortest) path from

```
Listening for incoming requests...

Message from interface: "mitchell" "delivermail" "Oct  1 12:51" "Wed Oct  1 13:48" "r-5303" "r-5313"
  2 n2 (done)
    CALCULATING PRIORITIES
    Request: #<HAS-ITEM MITCHELL DELIVERMAIL>   Rank: 5
  4 n4 <*finish*>
  5   n5 (has-item mitchell delivermail)
  7   n7 <deliver-item r-5313 mitchell delivermail>
  8     n8 (robot-has-item mitchell delivermail)
 10     n10 <acquire-item r-5303 mitchell delivermail>
 11       n11 (robot-in-room r-5303)
 13       n13 <goto-pickup-loc mitchell r-5303>
 14     n14 (robot-in-room r-5313)
 15     n15 goto-pickup-loc ...no choices for bindings (I tried)
 16     n17 <goto-deliver-loc mitchell r-5313>

 17       n18 <GOTO-PICKUP-LOC MITCHELL R-5303>
          SENDING COMMAND (TCAEXPANDGOAL "navigateToG" #(TASK-CONTROL::MAPLOCDATA 567.0d0 2316.5d0))
          ...waiting...
          Action NAVIGATE-TO-GOAL-ACHIEVED finished.

          Verifying Location: R-5303
          SENDING COMMAND (TCAEXECUTECOMMAND "C_say" "Am I really at the door of room R-5303?")
          SENDING COMMAND (TCAEXECUTECOMMAND "C_say" "Please answer on my keyboard.")
Am I really at the door of room R-5303? (y/n): y
          COMPLETED-ACTION (GOTO-LOCATION 1 R-5303)

 18     n19 <ACQUIRE-ITEM R-5303 MITCHELL DELIVERMAIL>
        SENDING COMMAND (TCAEXECUTECOMMAND "C_say" "Please place Tom Mitchell's mail delivery on my tray.")
        SENDING COMMAND (TCAEXECUTECOMMAND "C_say" "Please indicate on my keyboard when you are finished.")
Are you finished placing Tom Mitchell's mail delivery on my tray?  (y/i(mpossible)): y
        COMPLETED-ACTION (ACQUIRE-ITEM 1 "Tom Mitchell's mail delivery")

 19     n20 <GOTO-DELIVER-LOC MITCHELL R-5313>
        SENDING COMMAND (TCAEXPANDGOAL "navigateToG" #(TASK-CONTROL::MAPLOCDATA 567.0d0 4115.0d0))
        ...waiting...
        Action NAVIGATE-TO-GOAL-ACHIEVED finished.

        Verifying Location: R-5313
        SENDING COMMAND (TCAEXECUTECOMMAND "C_say" "Am I really at the door of room R-5313?")
        SENDING COMMAND (TCAEXECUTECOMMAND "C_say" "Please answer on my keyboard.")
Am I really at the door of room R-5313? (y/n): y
        COMPLETED-ACTION (GOTO-LOCATION 1 R-5313)

 19   n21 <DELIVER-ITEM R-5313 MITCHELL DELIVERMAIL>
      SENDING COMMAND (TCAEXECUTECOMMAND "C_say" "Please take Tom Mitchell's mail delivery from my tray.")
      SENDING COMMAND (TCAEXECUTECOMMAND "C_say" "Please indicate on my keyboard when you are finished.")
Are you finished taking Tom Mitchell's mail delivery from my tray?  (y/i(mpossible)): y
      COMPLETED-ACTION (DELIVER-ITEM 1 "Tom Mitchell's mail delivery")

Achieved top-level goals.
-----
Listening for incoming requests...
```

Figure 6: Sample Run

```
Define: $DK \leftarrow$ domain knowledge
Define: $G \leftarrow$ top-level goal
Define: $PG \leftarrow$ pending goals cache (unsolved top-level goals and their subgoals)

At each PRODIGY interrupt point:
    Let $R$ be the list of pending unprocessed requests
    For each $request \in R$, turn $request$ to goal:
        - $DK \Leftarrow DK \cup \{$ (needs-item request-userid request-object)
                          (pickup-loc request-pickup-loc)
                          (deliver-loc 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
```

Figure 7: Integrating new goal requests into the search tree.

the current location to the requested location, and then uses probabilistic reasoning to navigate to the requested goal. The model performs reasonably well given incomplete or incorrect metric information about the environment and in the presence of noisy effectors and sensors. The command `C_say` sends the string to the speech board, and the response is used by ROGUE while monitoring execution (described in more detail below). There is a large variety of available commands, including those to request or update current location information, to acquire images through the vision camera, and to notice landmarks.

The complete procedure for achieving a particular task is summarized as follows:

1. Receive task request
2. Add knowledge to state model, create top-level goal
3. Create plan
4. Send execution commands to robot, monitoring outcome

Xavier successfully executes the plan, stopping at the requested doors, asking for and then delivering the mail. This behaviour was developed in the simulator and then *tested on the robot*. Despite a few hardware problems and a slow network connection, the robot successfully and completely executed the plan.

We have described above ROGUE's behaviour in the face of a single goal request when no errors occur. The sections below describe how ROGUE handles multiple goal requests, reasoning about prioritizing and interrupting actions, and also how it handles simple plan failures.

4. Additional Behaviours

The capabilities described in the preceding section are sufficient to create and execute a simple plan in an unchanging world. The real world, however, needs a more flexible system that can monitor its own execution and compensate for problems and failures. In addition, simple single-goal plans such as the one described above are overly simplistic and do not address the needs of the people who will be using these robotic agents. This section describes the extensions we have implemented to the base-line system in an attempt to start addressing real-world issues.

Interrupts & Multiple Goals

It is very possible that while Rogue is executing the plan to achieve its first goal, other users may submit goal requests. Rogue does not know *a priori* what these requests will entail. One common method for handling these multiple goal requests is simply to process them in a first-come-first-served manner; however this method ignores the possibility that new goals may be more important or could be achieved opportunistically.

Rogue has the ability to process incoming asynchronous goal requests, prioritize them and identify when different goals could be achieved opportunistically. It is able to temporarily suspend lower priority actions, resuming them when the opportunity arises; and it is able to successfully interleave similar requests. This section describes how these capabilities are implemented.

The requests arrive via the TCA message interface, and wait on a communications socket until they are processed. By using PRODIGY4.0's *interrupt* mechanism, Rogue is able to introduce the new goal request into the planning search tree. A PRODIGY interrupt is user-defined code that is executed by the system at least once per decision; in Rogue, the interrupt is implemented as a routine to check the socket for incoming messages. Pseudocode for doing the full goal integration is shown in Figure 7. The important points are that (a) the relevant information about the request is added to PRODIGY's domain model, and (b) the new goal is added to the list of pending goals – the goals that must be achieved before the planning is complete.

When PRODIGY reaches the next decision point, it fires any relevant search control rules. Search control rules force the planner to focus its planning effort on selected or preferred goals, as described above. Figure 8 shows Rogue's goal selection control rule which forces PRODIGY to examine all of its remaining unsolved goals; it is at this point when PRODIGY first starts to reason about the newly added task request. This particular control rule selects those goals with high priority and those goals which can be opportunistically achieved without compromising the main high-priority goal.

The function `(ancestor-is-top-priority-goal)` calculates whether the goal is a subgoal of a high priority goal. Rogue prioritizes goals according to a simple, modifiable metric. This metric currently involves looking at the user's position in the department and at the type of request: $Priority = PersonRank + TaskRank$. The request also contains deadline information and a "why" slot for additional reasoning to be implemented in the future; this information would allow goal priorities to change with time or situation-dependent features.

The function `(compatible-with-top-priority-goal)` allows Rogue to identify 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 currently defined by physical proximity (*"on the path of"*) with a fixed threshold for being too out of the way, although other features of the domain could (and should) be taken into account.

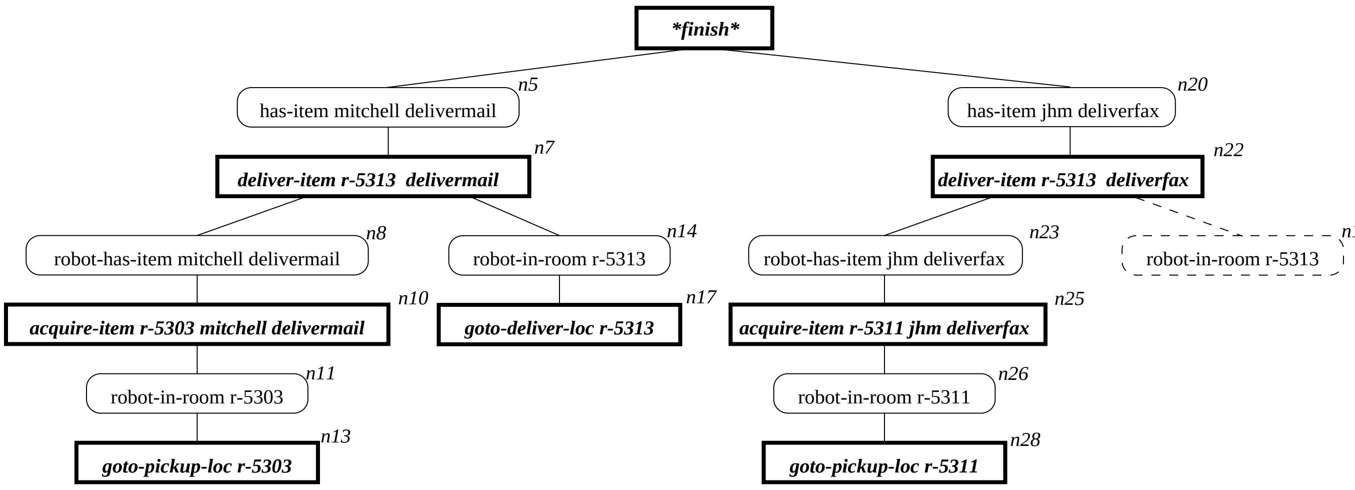
The control rule feature of PRODIGY permits plans and actions for one goal to be interrupted by another without necessarily affecting the validity of the planning for the interrupted goals. Prodigy simply suspends the planning for the interrupted goal, plans for and achieves the new goal, then returns to planning for the interrupted goal. By using its domain model, PRODIGY is able to identify whether the suspended plan has been invalidated; if so, then it will replan the invalid portion of the plan.

The search tree shown in Figure 9 shows how PRODIGY expands the two goals `(has-item mitchell delivermail)` and `(has-item jhm deliverfax)`. The second user (`jhm`) is a more important person, making a more important request. The request arrives via the TCA message interface while Xavier is moving towards room 5303. Rogue examines the new request and identifies that it is more important than the original (current) goal. However, the current goal not only shares a delivery point with the new goal, but also the physical path of the original goal subsumes that of the new goal. Rogue decides therefore that the two goals are compatible and that it can achieve the lower priority goal without seriously compromising the new goal. It continues along its path to room 5309, acquires the first object, then moves to room 5311 where it acquires the second object, then completes the delivery of both items to room 5313.

Figure 10 shows how two plans might be merged. If the two plans are compatible, Rogue identifies the order which most exploits the similarity between the two plans, and merges the steps accordingly (orderings other than the one shown are possible, and steps may be elimintated if appropriate). If however, the two plans are not compatible, Rogue suspends execution of the lower priority plan until the higher priority one is complete. When it resumes execution of less important plan, it *does not re-execute unnecessary parts.* If, for example, Rogue had already acquired the item in question, it would not attempt to reacquire it; the knowledge of having acquired the object is not forgotten (see Figure 11).

```
At each PRODIGY decision point
   (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>))
```

Figure 8: Goal selection search control rule


finish
n5
has-item mitchell delivermail
n20
has-item jhm deliverfax
n7
deliver-item r-5313 delivermail
n22
deliver-item r-5313 deliverfax
n8
robot-has-item mitchell delivermail
n14
robot-in-room r-5313
n23
robot-has-item jhm deliverfax
n14
robot-in-room r-5313
n10
acquire-item r-5303 mitchell delivermail
n17
goto-deliver-loc r-5313
n25
acquire-item r-5311 jhm deliverfax
n11
robot-in-room r-5303
n26
robot-in-room r-5311
n13
goto-pickup-loc r-5303
n28
goto-pickup-loc r-5311

```
Solution:  <goto-pickup-loc mitchell r-5309>  -  executed.
           <acquire-item r-5309 mitchell delivermail>  -  executed.
           <goto-pickup-loc jhm r-5311>  -  executed.
           <acquire-item r-5311 jhm deliverfax>  -  executed.
           <goto-deliver-loc mitchell r-5313>  -  executed.
           <deliver-item r-5313 jhm deliverfax>  -  executed.
           <deliver-item r-5313 mitchell delivermail>  -  executed.
```

Figure 9: Search Tree and Solution for two task problem; goal nodes in ovals, executed actions in rectangles.

Monitoring Execution, Detecting Failures & Replanning

Any action that is executed by any agent is not guaranteed to succeed in the real world. Probabilistic planners may increase the probability of a plan succeeding, but the domain model underlying the plan is bound to be incompletely or incorrectly specified. Not only is the world more complex than a model, but it is also constantly changing in ways that cannot be predicted. Therefore any agent executing in the real world must have the ability to monitor the execution of its actions, detect when the actions fail, and compensate for these problems.

The TCA architecture provides mechanisms for creating *exception handlers* which can monitor specific events as they are noticed by the system. These exception handlers can be added incrementally, and are invoked by TCA whenever a message regarding the monitored event appears. Currently, ROGUE monitors the events that indicate when the command `navigateToG` succeeds or fails. `navigateToG` may fail under several conditions, including detecting a bump, detecting corridor or door blockage, and detecting lack of forward progress. The command is able to compensate for certain failures, including obstacle avoidance and missing important features of the environment; if it manages to successfully compensate for these failures, it does not report the failure.

Whenever the navigation module reports that it has completed a command, *either with success or with failure*, ROGUE verifies the location. Currently, the interaction is strictly with nearby people: ROGUE sends a `SAY` command to the speech board, and expects a reply to questions on the keyboard of its onboard laptop computer. It is intended that this human interaction only occur as a last resort, when other autonomous behaviours do not suffice.

If ROGUE detects that in fact the robot is not at the correct goal location, PRODIGY's domain knowledge is updated to reflect the actual position, rather than the expected position. *This update has the direct effect of indicating to* PRODIGY *that the execution of an action*

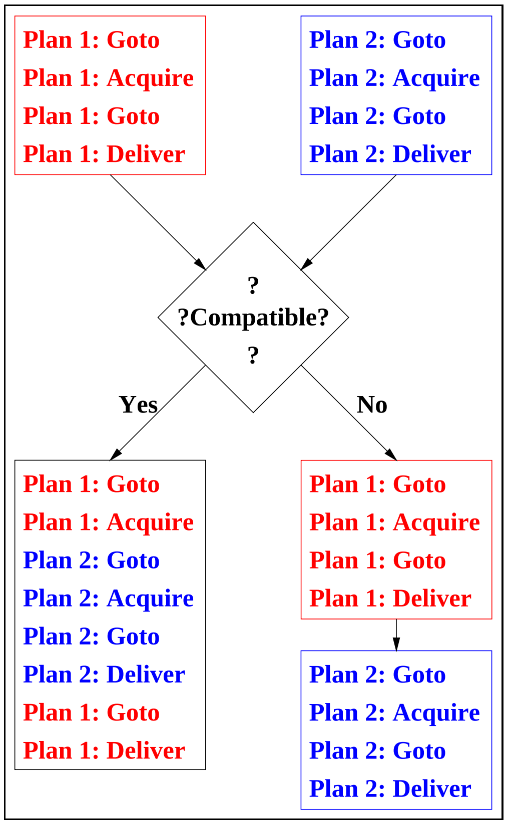


Figure 10: Merging Two Plans

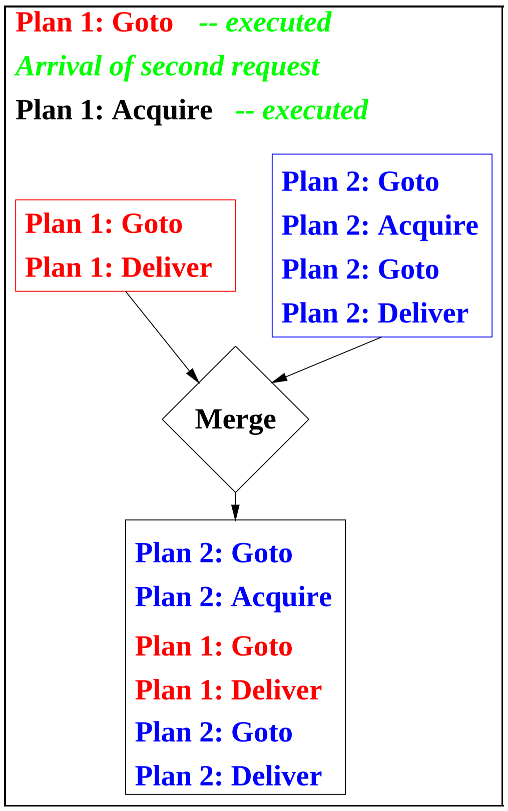


Figure 11: Merging Two Plans, first partially executed

failed, and it will attempt to find another action which can achieve the goal. PRODIGY exhibits this replanning behaviour as an inherent part of its design: the outcome of an action must be the same as the *expected* outcome. When this expectation is invalidated, PRODIGY will attempt to find another solution. This behaviour can effectively be described by the following steps:

```
1. Select appropriate action that will achieve
   or partially achieve the goal
2. Execute action
3. If action succeeded
   Then: Continue planning
   Else: Goto step 1
```

The process is described in more detail by Stone [11].

In this manner, ROGUE is able to detect simple execution failures and compensate for them. The interleaving of planning and execution reduces the need for replanning during the execution phase and increases the likelihood of overall plan success. However, ROGUE cannot autonomously decide whether it has in fact achieved the goal location nor can it nor can it deal with any form of hardware failure or software crash. For those failures it can identify, it has simple hard-wired correction techniques, and persistently tries to achieve the goal until all possible options are exhausted. Our research plan includes finding methods of more informed replanning.

5. Summary

This paper has described ROGUE, an integrated planning and execution robot architecture. ROGUE's current capabilities include:

- receiving asynchronous goal requests from multiple users
- determining simple characteristics about particular users and tasks (such as office numbers and position within the department)
- being able to prioritize goals and focus planning on high priority goals until they are achieved, and then later work on lower priority goals;
- recognizing similar goals and opportunistically achieve them;
- interrupting actions to deal with a more important action, and then restart interrupted action appropriately;
- interleaving planning & execution to acquire data and monitor execution;
- dealing with simple plan failures, such as arriving at an incorrect location;
- interacting with people

This work is the basis for machine learning research with the goal of creating an agent that can reliably perform tasks that it is given. We intend to implement more autonomous detection of action failures and learning techniques to correct those failures. In particular, we would like to learn contingency plans for different situations and when to apply which correction behaviour. We also intend to implement learning behaviour to notice patterns in the environment; for example, how long a particular action takes to execute, when to avoid particular locations (eg. crowded hallways), and when sensors tend to fail. We would like, for example, to be able to say *"At noon I avoid the lounge"*, or *"My sonars always miss this door...next time I'll use pure dead-reckoning from somewhere close that I know well"*, or even something as apparently simple as *"I can't do that task given what else I have to do."*

PRODIGY has been successfully used as a test-bed for machine learning research many times (eg. [7; 14; 12]), and this is the primary reason why we selected it as the deliberative portion of ROGUE. Xavier's TCA architecture supports incremental behaviours and therefore will be a natural mechanism for supporting these learning behaviours.

References

[1] Philip E. Agre and David Chapman. Pengi: An implementation of a theory of activity. In *Proceedings of AAAI-87*, pages 268–272, San Mateo, CA, 1987. Morgan Kaufmann.

[2] Jaime G. Carbonell, Craig A. Knoblock, and Steven Minton. PRODIGY: An integrated architecture for planning and learning. In K. VanLehn, editor, *Architectures for Intelligence*. Erlbaum, Hillsdale, NJ, 1990. Also Available as Technical Report CMU-CS-89-189.

[3] R. James Firby. Task networks for controlling continuous processes. In *Proceedings of AIPS-94*, pages 49–54, Chicago, IL, June 1994.

[4] Kristian Hammond, Timothy Converse, and Charles Martin. Integrating planning and acting in a case-based framework. In *Proceedings AAAI-90*, pages 292–297, San Mateo, CA, 1990. Morgan Kaufmann.

[5] Drew McDermott. Planning and acting. *Cognitive Science*, 2, 1978.

[6] Joseph O'Sullivan and Karen Zita Haigh. *Xavier*. Carnegie Mellon University, Pittsburgh, PA, July 1994. Manual, Version 0.2, unpublished internal report.

[7] M. Alicia Pérez. *Learning Search Control Knowledge to Improve Plan Quality.* PhD thesis, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA, July 1995. Available as Technical Report CMU-CS-95-175.

[8] Reid Simmons. Becoming increasingly reliable. In *Proceedings of AIPS-94*, pages 152–157, Chicago, IL, June 1994.

[9] Reid Simmons. Structured control for autonomous robots. *IEEE Transactions on Robotics and Automation*, 10(1), February 1994.

[10] Reid Simmons, Long-Ji Lin, and Chris Fedor. Autonomous task control for mobile robots. In *Proceedings of the IEEE Symposium on Reactive Control*, Philadelphia, PA, September 1990.

[11] Peter Stone and Manuela Veloso. User-guided interleaving of planning and execution. In *Pro-*

ceedings of the European Workshop on Planning, September 1995.

[12] Manuela M. Veloso. *Planning and Learning by Analogical Reasoning*. Springer Verlag, Berlin, Germany, December 1994. PhD Thesis, also available as Technical Report CMU-CS-92-174, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA.

[13] Manuela M. Veloso, Jaime Carbonell, M. Alicia Pérez, Daniel Borrajo, Eugene Fink, and Jim Blythe. Integrating planning and learning: The PRODIGY architecture. *Journal of Experimental and Theoretical Artificial Intelligence*, 7(1), January 1995.

[14] Xuemei Wang. Learning by observation and practice: An incremental approach for planning operator acquisition. In *Proceedings of the Twelfth International Conference on Machine Learning*, Tahoe City, CA, 1995.