N89-26583

A Situated Reasoning Architecture for Space-based Repair and Replace Tasks*

Ben Bloom
Debra McGrath
Jim Sanborn

The MITRE Corporation
7525 Colshire Drive
McLean, VA 22102-3481

Abstract

An area of increasing interest within AI and Robotics is the integration of techniques from both fields to the problem of controlling autonomous systems. Space-based systems, such as NASA's EVA Retriever, provide complex, realistic domains for this integration research. Space is a dynamic environment, where information is imperfect, and unexpected events are commonplace. As such, space-based robots need low level control for collision detection and avoidance, short-term load management, fine-grained motion, and other physical tasks. In addition, higher level control is required to focus strategic decision making as missions are assigned and carried out. Throughout the system, reasoning and control must be responsive to ongoing change taking place in the environment.

This paper reports on current MITRE research aimed at bridging the gap between high level AI planning techniques and task-level robot programming for telerobotic systems. Our approach is based on incorporating *situated reasoning* into AI and Robotics systems in order to coordinate a robot's activity within its environment. Thus, the focus of this research is on controlling a robot embedded in an environment, as opposed to the generation and execution of lengthy robot plans. We present an integrated system under development in a "component maintenance" domain geared towards repair and replacement of Orbital Replacement Units (ORUs) designed for use aboard NASA's Space Station *Freedom*. The domain consists of a component-cell containing ORU components and a robot (manipulator and vision system) replacing worn and/or failed components based on the collection of components available at a given time. High level control reasons in "component space" in order to maximize the number operational component-cells over time, while the task-level controls sensors and effectors, detects collisions, and carries out pick and place tasks in "physical space." Situated reasoning is used throughout the system to cope with, for example, non-deterministic component failures, the uncertain effects of task-level actions, and the actions of external agents operating in the domain.

*This work is funded by MITRE-Sponsored Research Projects 97060 and 97140. The authors wish to thank MITRE for its ongoing support of this work. Email address: sanborn@ai.mitre.org.

PRECEDING PAGE BLANK NOT FILMED

1 Introduction

There is a resurgence of interest within the AI and Robotics communities in integrated efforts leading to the development of robust, autonomous systems for use in dynamic, uncertain, and unpredictable domains. NASA in particular has several efforts underway, including its *Systems Autonomy Technology Program* (SATP), a ten year program to establish NASA as a world leader in intelligent autonomous systems research and development, the EVA Retriever, and the Mars Rover project. These programs are aimed at addressing two issues in space exploration:

1. For manned missions, human EVA is dangerous, expensive, and time-consuming.

2. For unmanned missions, signal delay times require autonomous control throughout non-trivial time intervals.

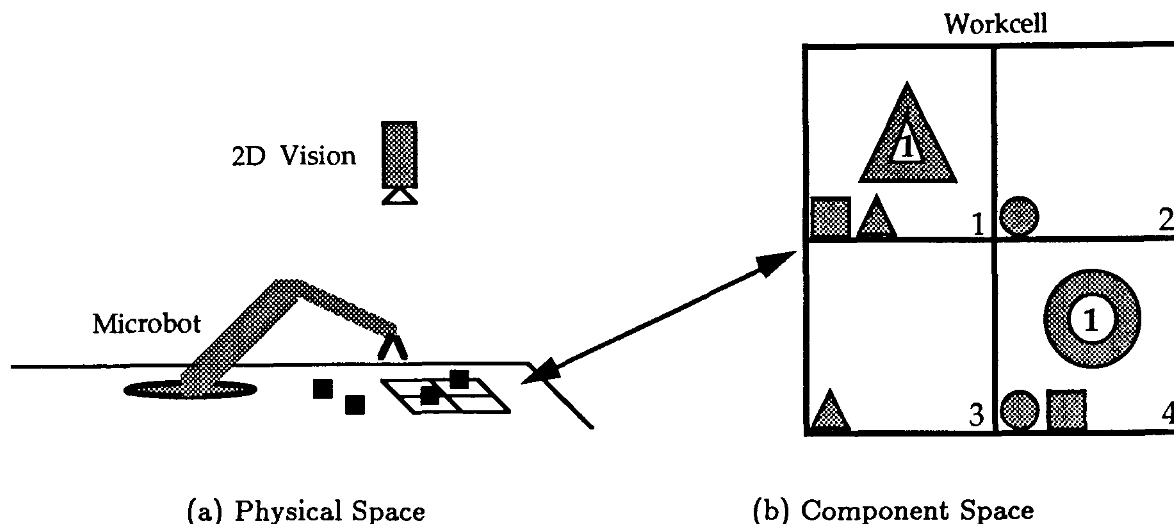
One of the major hurdles in building autonomous systems is the integration of off-line deliberative reasoning (e.g., task planning, route planning, and resource allocation, etc.) with real-time situated control (e.g., collision and obstacle avoidance, path expansion, load management, calibration from landmarks, etc.). The former type of reasoning is *goal-directed*, and has led to the development of several constraint-posting planners (e.g., [Ste81], [Wil84], [Cha85]) generating plans to satisfy multiple goals. The latter type of reasoning is *event-driven*, in that responses to existing, perceived, or projected situations are required in order to maintain overall system integrity. AI research has begun to address situated reasoning, as well its integration with off-line plan generation (see, for example, [Kae86], [GLS87], [AC87], [Dea87], and [SH88]).

This paper presents the initial results of a combined MITRE research effort integrating AI planning and situated reasoning techniques with task-level robotics and perception for space-based autonomous systems. The long-term goal of this research is to integrate off-line planning, situated reasoning, and sensor/actuator subsystems across various levels of abstraction in order to provide both the reactive behavior necessary for survival in realistic environments, and the introspective reasoning required to carry out deliberate tasks and achieve desired goals. The work presented in this report lays the groundwork for this long-term goal by providing an integrated situated reasoning and task-level control architecture, as well as a system operating in a realistic application domain. Examples from this domain are used throughout the paper to illustrate the approach.

2 A Component Repair and Maintenance Domain

One of the many application areas for space-based autonomous systems is routine extra-vehicular maintenance. By "routine maintenance," we refer to a general class of situations in which components of a system are scheduled for maintenance (as determined by expected lifetime) and are also tended to when they fail unexpectedly. In such an application, a robot must allocate available resources (spare parts, or modular components such as ORUs) in order to maximize the overall operating status of a collection of components.

The "routine repair and replace" domain shown in Figure 1 captures this idea. It consists of a robotic manipulator, vision system, a ***component workcell***, and a collection of ***components***. The workcell is an $N \times M$ array of ***compartments***,



(a) Physical Space (b) Component Space

Figure 1: A Simple Component Repair and Maintenance Domain

each of which may be "filled" by components of various types. In this way, the workcell may be thought of as a modular "breadboard" into which components are inserted to become operational. Each workcell compartment is labelled according to the types of components that may fill it. A component is said to be *acceptable* to a compartment whenever it may be used to fill that compartment. Finally, an expected lifetime (the *mean time to failure*, MTTF) is associated with each type of component in order to model both routine and unexpected component failures.

In the current system, the workcell is modelled by a "bin" on a tabletop, with different shaped objects (cylinders, rectilinear and triangular blocks, etc.) representing various component types (see Figure 1(a)). States of the domain are subject to constant flux at the hands of external (human) agents, whose unanticipated actions may include adding, removing, moving, and breaking components. In addition, the fact that components may fail unexpectedly at any time also requires attention to the ongoing situation. The choice of this repair and replace domain was influenced by the following considerations:

1. The architecture should be realistically scalable to handle any of a variety of repair and replace tasks to be performed in environments characterized by dynamics and uncertainty (such as Space Station ORU replacement).

2. The scenario requires the integration of physical control (robotics) with high-level reasoning (AI).

3. The hardware required for developing the testbed scenario was readily available.

2.1 Physical vs. Component Space

The reasoning required for successful operation in this domain falls into two classes: reason-

ing in *physical space*, and reasoning in *component space*. Physical space reasoning includes the planning, executing, and monitoring of collision-free paths for the manipulator and moved objects, detecting obstacles, noticing objects when they are moved, and generally dealing with physical aspects of the domain. The physical space reasoner used to control the manipulator and vision system shown in Figure 1(a) is known as the *Task-level Robot Programming System* (TLRPS). Component space reasoning includes allocating available components among empty compartments, prioritizing replacement and repair tasks according to various heuristics, as well as reacting to unexpected component failures, moves, additions, and deletions. The *Component Space Reasoner* (CSR) provides this functionality for the component space corresponding roughly to Figure 1(b).

2.2 Interface Language

This section presents the communication specification between TLRPS and CSR. The interface has been designed to distinguish between physical and component space aspects of the domain.

2.2.1 CSR→TLRPS

`Put_in` *object compartment*$_i$: Move *object* from its present (table) location into the (empty) *compartment*$_i$.

`Put_at` *object* x y : Place *object* on the tabletop at TLRPS coordinates (x y); *object* is assumed to be either on the table or within the workcell.

`Put_down` *object* x y : Put down (held) *object* on the tabletop at TLRPS coordinates (x y).

2.2.2 TLRPS→CSR

`Begin_update` : Initiates an update of object and location information from TLRPS. Followed by one or more instance of:

- `Delete` *object* : *object* has been removed from the domain.
- `Move` *object* x y θ : *object* now centered at (x y) rotated by θ degrees.
- `Add` *object* x y θ : *object* has appeared in the domain, centered at (x y) rotated by θ.

`End_update` : Signals the end of the update.

`Failed_grasp` *object* : TLRPS could not grasp *object*.

`Collision` *held_object object_in_path* : *object_in_path* prevents moving *held_object*.

`Unreachable_object` *object* : *object* cannot be reached in its current location.

`Unreachable_location` x y : (x y) cannot be reached.

3 Technical Approach

This section describes the operation of TLRPS, CSR, and their integration in the repair and replace domain. Since this domain is non-static, each system must cope with discrepencies between anticipated and actual states of the domain. For example, components may move from expected locations and may fail (or be broken) before their MTTF has elapsed. Since external agents may change the environment, neither system can make accurate long-term projections regarding future states. Rather, the system must optimize local behavior based on existing and projected states given the overall component maintenance goals. The top-level system architecture is shown in Figure 2.

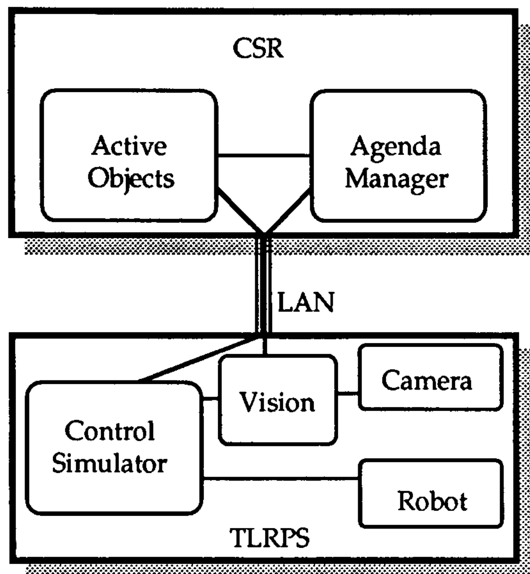


Figure 2: Top-level System Architecture

3.1 Physical-Space Reasoning

TLRPS resides on a Silicon Graphics IRIS 4D/70GT workstation running the IGRIP 3D robotics modeling and simulation system from Deneb Robotics. The vision system consists of software provided by NASA running on an IBM PC-AT with added frame grabber and image processing boards from Data Translation. The robot currently in use is a Microbot Alpha I.

A single camera with a fixed viewpoint is used to capture the layout of the robot's workspace. The vision software in the PC-AT classifies the objects in the workspace according to its training data set corresponding to the physical component objects. A workspace description containing object types and their locations and orientations is then sent to the IRIS. Image processing software on the IRIS interprets the workspace description on each cycle and modifies the world model accordingly, generating CSR update messages, as well as updating the IGRIP simulation's 3D graphic display of the robot and the workspace. Users may interact directly with TLRPS by entering task-level commands via pop-up menus, or turn control over to CSR. When a task-level command is received by TLRPS (whether from CSR or a human user), the command is simulated in 3D graphics and executed by the robot. The simulation runs one step ahead of the actual robot execution, checking for possible collisions and out-of-reach conditions. If such a condition is detected, TLRPS performs error recovery operations, registering the simulation and the robot to a safe configuration. An exception-dependent error message is generated and sent to CSR when it is controlling TLRPS.

3.2 Component-space Reasoning

CSR is divided into two main modules: an ***agenda manager*** for prioritizing tasks and issuing commands to TLRPS, and a collection of ***objects*** corresponding to the various physical objects in the domain at a given time. CSR's top-level control loop is

(0) **get** component types used
(1) **await** TLRPS update
(2) **process** TLRPS update
(3) **if** there is an executable task
(4) **then** issue it to TLRPS
(5) goto (1)

The initialization step (0) determines components type definitions and their associated parameters. This is accomplished by consulting a file provided before startup. Once this information has been processed, CSR enters its main control loop. The first stage of this loop (Step 1 above) simply puts CSR into a wait state until an update from TLRPS is available. Recall that these updates consist of a collection of **Add**,

`Delete`, and `Move` messages. The second stage (Step 2) processes the update's content. During this stage, messages are sent to existing CSR objects mentioned in the update; the message sent depends on what the update to the object happens to be. During the processing of these messages, objects may request that the agenda manager generate new tasks to, or remove existing tasks from, its collection of tasks. In the final stage (Steps 3 and 4), the agenda manager prioritizes its collection of tasks and, if any are executable, issues a command to TLRPS to carry out the most important executable task. Within CSR, a *task* is one of:

`routine_replace` *component* $compartment_i$: replace worn (but operating) *component* with an existing new component acceptable to $compartment_i$.

`immediate_repair` *component* $compartment_i$: replace failed *component*.

`await_component` $compartment_i$: fill $compartment_i$ when an acceptable component is added.

`install_component` *component* $compartment_i$: put *component* into $compartment_i$ and start it operating.

`deinstall_component` *component* $compartment_i$: stop operating *component* and remove it from $compartment_i$.

`move_component` *component* x y : move *component* from its current location to $(x\ y)$.

3.2.1 Active Objects

Most of the reasoning done by CSR is triggered during update processing. For each physical object in the domain, CSR generates a *component* object in its component space. The workcell, and each of its compartments, is also represented as an active CSR object. Updates from TLRPS relating to physical objects are then translated into messages to CSR objects, which may take various update-dependent actions. This approach associates reasoning with changing information at the object level, rather than on the more traditional rule-interpreter/database approach. As such, reasoning and control are modified by changing objects' responses to messages, rather than by the actions contained in rule consequents.

To illustrate this approach to object modelling and situated reasoning, we follow the processing of CSR's initial update. This update consists of a collection of `Add` messages from TLRPS. The first such message processed corresponds to the workcell. On encountering this message, CSR generates a workcell object, an object for each of the workcell's compartments, and initializes their parameters. Following this, a component object is created for each added component. As part of its `Add` message processing, the component checks to see whether or not it lies within some compartment. If not, its status is "new." Otherwise, the compartment object is consulted to determine whether or not the component should be accepted. If so, the compartment "installs" the component, by setting the component's status to "operating," and schedules a `routine_replace` task for sometime in the future, depending on the MTTF of the newly installed component. If the compartment does not accept the component, it attempts to have the offending component removed by generating a `move_component` task.

Once this initial update has been processed, CSR knows which compartments contain operating components, which need to be filled, as well as which components are available in

the domain. Empty compartments generate `await_component` tasks and compete for newly arriving components according to several criteria, including (1) how long the compartment has been empty, and (2) the number of different types of components the compartment accepts. The next section discusses the use of these heuristics in prioritizing tasks.

3.2.2 Tasks and the Agenda Manager

In order to take action, component and compartment objects generate tasks which are added to CSR's *agenda*: a simple partially ordered set of tasks. All tasks have the following properties, which are used in determining their relative importance and/or execution status:

`status` : one of

- `:pending` : not yet ready for execution,
- `:executable` : ready for execution,
- `:active` : in progress, or
- `:done` : finished, does not indicate success or failure.

`priority` : a measure of the task's importance, one of

- `:normal` : a routine task, such as `routine_replace`,
- `:asap` : a non-routine task, such as `immediate_repair`, or
- `:now` : highest priority tasks, such as `await_component`.

`resource_measure` : measure of difficulty in obtaining resources for this task.

`timestamp` : actual task instantiation time.

`supertask` : associated parent task.

`subtasks` : subtasks comprising an abstract task.

Tasks fall into two classes: *primitive* tasks, corresponding directly to TLRPS commands, and *abstract* tasks, which have no analogous TLRPS command. Abstract tasks may have an associated *test* executed once per cycle whenever the task's status is `:pending`. All tasks have an *act*, which is executed when an `:executable` task becomes `:active`. In the current system, `install_component`, `deinstall_component`, and `move_component` are primitive (corresponding to various types of `put_in`, `put_at`, and `put_at` commands, respectively); all other CSR tasks are abstract.

In most traditional plan generation and execution systems, a complete plan to achieve a goal is generated and then executed stepwise. An underlying assumption of this approach is that the world will behave as expected during plan execution. If exceptions occur during execution, the usual recourse is to more planning. As an example, a "routine replace" operation on a compartment is normally composed of two steps:

1. `deinstall_component` *oldc* $compartment_i$
2. `install_component` *newc* $compartment_i$

under the assumption that *newc* will be available when step (2) is to be executed. However, if *newc* is removed sometime during the execution of step (1), the resulting situation is the same as one in which $compartment_i$ were simply waiting to be filled. Recall that in this case, the empty compartment generates an `await_component` task to find and then install a suitable component.

In general, CSR uses component and compartment objects to assist in carrying out plans

whenever the domain is cooperative, but also ensures that the appropriate behavior results when "assumptions" fail. Rather than planning steps for anticipated future states, CSR generates one step at a time, and uses feedback from the world to determine its next step. The definition for routine_replace looks like

```
(deftask routine_replace
         (compartment
          component
          new_component
          replace_time)
  :test (and (> (now) replace_time)
             new_component)
  :act (generate_task
          'deinstall_component
          component compartment)
       (reserve compartment
                new_component))
```

Notice that there is no mention of a task corresponding to step (2) above in this definition. If all goes well in the world, the compartment will generate an install_component using the new component, reserved for it by the routine_replace task, once the deinstall has been successfully carried out. However, if for some reason the new component is no longer available, the compartment simply generates an await_component, which searches for another suitable replacement. Since no assumption is made as to whether or not the new component will remain available, the appropriate response occurs in either case. This approach requires that CSR objects track their allocations (so that, for instance, the compartment can determine whether or not its reserved component is available or not), but this is easily managed by informing objects of their allocations and taking appropriate action during update message processing.

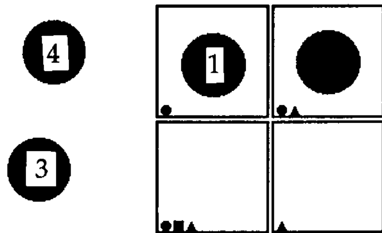
The task agenda is ordered in decreasing order of importance. This ordering is maintained by merging new tasks, and re-merging tasks when their parameters change, according to the following sequence of pairwise tests:

1. status – `:executable > :pending`
2. priority – `:now > :asap > :normal`
3. resources – prefer more constrained
4. time – prefer older tasks
5. arbitrary

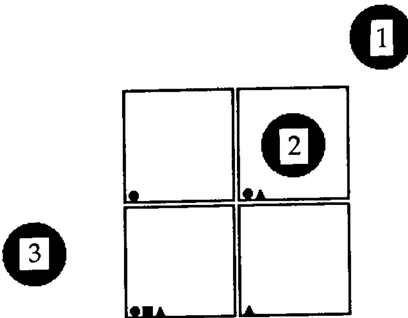
One case where re-merging is necessary is when an acceptable component becomes available to a pending await_component task. New components announce their availability by sending an advertise-available-resource message to the agenda manager, which in turn offers the new resource to pending tasks in decreasing order of importance. If a task allocates the available resource, the resource is allocated to it, and it is remerged into the agenda. When an available component is offered to an await_component task, it is allocated so long as it is acceptable to the task's compartment.

Another way an await_component task may find an acceptable component is for it to *usurp* the resources of another, less important task. In general, this process examines the task agenda from back to front until either (1) an acceptable resource is found, in which case it is usurped and the two tasks are re-merged into the agenda, or (2) the process reaches a task with higher priority than the intented usurper's, in which case no suitable resource is available.

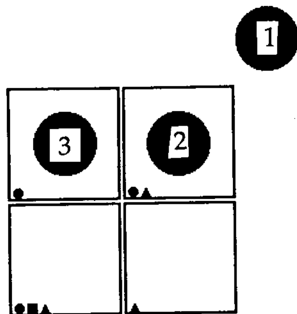
Figure 3(a) shows the state of a workcell as cylinder_1, in *compartment*$_1$, is to



(a) deinstall



(b) usurp



(c) install

Figure 3: CSR in action.

be replaced with `cylinder_4`. The work-cell also contains a cylinder (`cylinder_2`) operating in *compartment*$_2$ and scheduled for replacement by `cylinder_3` at some later time. The `routine_replace` task associated with *compartment*$_1$ acts, generating a `deinstall_component` task, which is placed at the head of the agenda (since it is primitive, and hence, immediately executable). This in turn issues a TLRPS `put_at` command to remove `cylinder_1`. Sometime during the execution of this command, `cylinder_4` is removed, resulting in the situation shown in Figure 3(b). At this point, since *compartment*$_1$ is empty, and no longer has a reserved component, it generates an `await_component` task. `Cylinder_3` should be "available" to this new task, but it is currently allocated to the `routine_replace` task associated with *compartment*$_2$. However, since the new task has higher priority, it usurps the resources (`cylinder_3`) of the lower priority `routine_replace` task. This changes its status to `:executable`, its action generates a primitive `install_component` task, which in turn issues a TLRPS `put_in` command to complete the replacement, as shown in Figure 3(c).

3.3 CSR/TLRPS Integration

The two systems presented above have been developed jointly but at physically different sites in McLean, Virgina (CSR), and Houston, Texas (TLRPS). CSR is implemented in Portable Common Loops (PCL) and resides on a Symbolics Lisp Machine. As noted above, TLRPS resides on a Silicon Graphics machine. The two systems are connected on an Ethernet LAN at the Houston site. They communicate via TCP/IP streams over this network, using the interface language presented in Section 2.2.

3.4 The TLRPS Simulator

Since CSR and TLRPS have been developed at different sites, a TLRPS simulator has been implemented for CSR's development and testing. In addition to simulating TLRPS's physical-space reasoning, allowing external agents to manipulate the domain, this simulator models component operation so that unexpected (i.e., pre-MTTF) component failures may occur. It also provides a graphic user-interface to CSR.

The TLRPS simulator used in testing CSR also runs on a Symbolics Lisp Machine, with the two systems simulating CSR↔TLRPS interface over a local Chaosnet. Figure 3 was generated using screen images from the TLRPS simulator.

4 Current Status and Future Work

This integrated project has addressed the problem of integrating high-level and task-level reasoning in a dynamic environment. The architectures used in both systems are domain-independent, and will be useful for other NASA applications, as well as broader application in manufacturing and assembly, hazardous materials handling, military operations, and undersea work.

The existing system is able to react to any unexpected change that occurs during the execution of a single CSR primitive task. During periods of CSR inactivity, updates are available at a rate of approximately one every ten seconds. CSR's real response time is on the order of one-tenth of a second, so the "snapshot" nature of updates and update processing suffices for this domain. In general, the rate of change in dynamic domains is much faster, so that situated reasoning must be based on *projecting* future states in order to anticipate and avoid exception situations. An approach to situated reasoning based on these observations is presented in [San88].

Due to existing hardware, the current system has very little low-level reactive capability. An improved hardware system and more integrated reasoning and control architecture will be required for more general purpose, robust autonomous control. To this end, MITRE is establishing an ***Autonomous Systems Laboratory*** (ASL). Research in the ASL will focus on the integration of deliberative (off-line) planning, situated reasoning, and hardware subsystems. The ASL will be composed of "off the shelf" sensing, robotics, and AI hardware and firmware representing the significant advances made in these technologies in recent years. The focus of the current project will shift from situated reasoning under a constant goal (e.g., routine repair and replace), toward more flexible control in a domain where several different types of goals are to be achieved over time. A ground-based mobile system operating in a dynamic domain will be used as a test-bed to simulate a flexible space-based automaton for routine extra-vehicular repair, assembly, and retrieval tasks. Deliberative planning will take as input a collection of tasks to be carried out (the "daily schedule") and determine an ordering among these tasks. Its output will be information used to monitor activity and constrain low-level task execution in the domain via a situated reasoning system. This latter system actually controls the physical system as it operates in its environment by controlling its reactions to existing and anticipated states of affairs. This on-line system uses the constraints from the deliberative planner as heuristics in selecting among tasks it can perform, but is independently capable of a basic level of competence in the domain.

Acknowlegements

The authors thank Pete Bonasso and Jim Reynolds for their contributions to this work, as well as useful comments on this paper. In addition, productive discussions have been held with Jack Benoit, Chris Elsaesser, and Vincent Hwang.

References

[AC87] Philip Agre and David Chapman. Pengi: an implementation of a theory of action. In *Proc. AAAI-87*, 1987.

[Cha85] David Chapman. *Planning for Conjunctive Goals.* AI 802, MIT, 1985.

[Dea87] Thomas Dean. Planning, execution, and control. In *Knowledge-Based Planning Workshop*, pages 29–1–29–10, DARPA, 1987.

[GLS87] Michael Georgeff, Amy Lansky, and Marcel Schoppers. *Reasoning and Planning in Dynamic Domains: An Experiment with a Mobile Robot.* International Technical Note 380, SRI, 1987.

[Kae86] Leslie Pack Kaelbling. An architecture for intelligent reactive systems. In M. Georgeff and A. Lansky, editors, *The 1986 Workshop on Reasoning about Actions and Plans*, pages 395–410, Morgan Kaufman, 1986.

[San88] James Sanborn. *A Model of Reaction for Planning in Dynamic Environments.* Master's thesis, University of Maryland, May 1988.

[SH88] James Sanborn and James Hendler. A model of reaction for planning in dynamic environments. *International Journal of AI in Engineering*, 1988. Special issue on planning.

[Ste81] M.J. Stefik. Planning with constraints (molgen: part 1). *Artificial Intelligence*, 16:141–169, 1981.

[Wil84] D.E. Wilkins. Domain-independent planning: representation and plan generation. *Artificial Intelligence*, 22:269ff, 1984.