Reliable Real-time Robot Operation Employing Intelligent Forward Recovery

Richard E. Smith and Maria Gini
Computer Science Department, University of Minnesota, Minneapolis, Minnesota 55455
Received March 17, 1986; accepted May 16, 1986

Modern computer-controlled robots typically fail at their tasks whenever they encounter an error, no matter how minor. The physical environment of a typical assembly robot is too unstructured to benefit from conventional software approaches to reliability. We present an approach which interfaces the real-time operation of the robot with an intelligent subsystem that performs error analysis and forward recovery if a failure occurs. Our approach involves a special representation of the robot's program that efficiently tracks the robot's operation in real time and is easy to modify to include automatically generated recovery procedures.

コンピューター制御ロボットは通常エラーが発生すると、それがどんなにマイナーなエラーでも、仕事を中止してしまう。普通の組立てロボットの周囲環境は良く組織されていないので、従来のソフトウェアでは信頼性の高い制御は期待できない。我々は、ロボットのリアルタイム操作と、エラー発生時にエラー解析と前向き回復(forward recovery)を実行するインテリジェントサブシステムのインターフェースを呈示する。リアルタイムでロボットの操作を効率的に追跡し、自動的に発生させた回復手順を容易に挿入できるロボットプログラムの特別の表現も含んでいる。

I. INTRODUCTION

Computer-controlled robot manipulators have opened new possibilities for automating industrial processes. Robots are operating today on a wide variety of tasks such as object handling, painting, and welding. Despite the growing interest and falling hardware prices many manufacturers have found it very difficult to install computer-controlled robots and automate complicated processes. Most robots in use today are performing simple motion sequences and do not rely on sensors for feedback or control. Few robots do parts assembly or use complex sensors such as cameras. Robot use is impeded by several problems inherent in robot programming,[1] particularly in the description and reliable execution of robot tasks.

Journal of Robotic Systems, 3(3), 281–300 (1986)
 CCC0471-2223/86/030281-20$4.00

Reliability problems in robot programming are different than in conventional software environments. The problems do not involve specification and validation of procedures, nor do they involve unreliable communication or processor resources. The problems stem from the variability of everyday matter and of physical mechanisms. Robot tasks often fail due to minor variations in the robot's motion or in the objects that the robot manipulates. Computational analysis of failures such as jammed or dropped objects is always time consuming and often mathematically intractable. It is seldom feasible to implement general purpose error detection and recovery procedures because such errors can manifest themselves in so many ways.

A promising approach to this problem is to apply artificial intelligence techniques such as knowledge-based programming. This involves the implementation of automated reasoning systems to deal with knowledge about robot manipulators, work cells, and manipulator programs. Reasoning about three-dimensional space and motion is also necessary. Although promising work has taken place in these areas, the techniques are seldom efficient enough in computation time to work effectively with industrial robots in time critical situations.

Our approach is to factor out the reflexive, real-time operations of the system from the reflective, automated reasoning components. Thus the robot can respond immediately when it must and take the time to ponder when its next action is not yet known. The system reasons about the task before its execution and generates a special program form that incorporates knowledge about the task at hand and about necessary sensor readings. The special form is used to monitor and control the robot in real-time as it performs its task. If a failure is detected, the real-time system pauses while an intelligent recovery system determines what happened and how to recover. The recovery system then gives the real-time system a procedure to lead the robot from its erroneous state into the next correct state in its task.

II. PROGRAMMING INDUSTRIAL ROBOTS

Robot manipulators consist of jointed mechanical members that are moved by servomotors in each joint. The robot is operated by supplying each servomotor with an appropriate sequence of joint angles. Many robots are programmed manually by a *teach pendant* that directly controls the robot's joints. An operator programs the robot by moving it through the required motions while the teach pendant records them. The robot then reproduces the motion sequence continuously when told to run. Teach pendant programming, however, seldom permits conditional programming or the use of sensor feedback other than joint positions. Another problem is that it requires direct interaction between the programmer and the robot, a situation that is both costly in robot time and dangerous for the programmer.[2]

The problems of teach pendant programming have led robotics researchers to develop *off-line* programming techniques. By off-line we mean "off-line with respect to the robot" and not "off-line with respect to the computer" as commonly intended in computer science. Off-line programming allows robot programs to be completely developed without involving the robot itself. Only final testing and tuning requires the real robot and real parts. The robot can continue to work on the assembly line while

the new program is developed, a major advantage when robots are components of complex industrial automation systems. Another advantage is that off-line programming can be performed in an environment more suited to programming than the factory floor. Larger computers can be used because they are not tied to a single robot. This opens the area of robot programming to computer-aided engineering techniques.

A number of languages have been developed for off-line robot programming. The simplest of these languages only provides an interface to the servomotors and motions are described in terms of robot joint angles. However, most languages now accept Cartesian coordinates and automatically perform the kinematic transformations either at compile time or at run time. Typical of these languages are AL and VAL.[3,4] These languages closely resemble conventional programming languages such as Ada or PL/1 except that they include statements that operate robots and other industrial equipment. The programmer specifies robot motions in terms of the coordinates of the destination the manipulator is supposed to reach. Grasping and releasing objects is performed by commanding the servo that controls the robot's gripper. Robot manufacturers and users have found these languages to be cost effective and satisfactory, if not ideal.

In the search for better robot languages, several researchers have turned to artificial intelligence. Much of this research has centered on *planning systems*. These systems take high level descriptions of robot tasks and automatically plan appropriate robot manipulator actions. These languages attempt to express the desired task in terms of the objects involved and operations on them: "Put Object A on Object B," for example. The planner determines what objects need to move, how to grasp them, what order to do things in, what manipulators to use, and so on. Although such an approach is very attractive, planning systems are still too rudimentary to solve the complex problems found in real applications of robots.

III. ROBOT RELIABILITY

The fundamental issue in robot reliability is that robot manipulators and the parts they deal with must all be in precise locations and follow precise motions. This problem appears during robot task development as a requirement that the programmer know the exact coordinates of everything taking part in the task. The location problem becomes important during execution of the task when inaccuracies of part sizes and manipulator positions creep in. Other problems may be parts slipping out of the manipulator's gripper, jammed parts feeders, or side effects where misplaced parts collide with other parts.

A variety of computer-aided engineering systems are helping with the problem of robot task development. Typical systems combine solid modeling, motion simulation, and computer graphics to provide a graphic simulation of robot motion. Robot programmers use these systems to test their programs by visually watching for collisions; some systems provide automatic collision detection as well.[5,6] Interactive development and simulation systems increase the likelihood that the robot's task is described accurately. The systems can provide important savings when they reduce the use of production robots for programming and debugging.

The development systems, however, do not contribute much to robot reliability

during task execution. It is well known that programs developed off-line tend to be unreliable and error prone. In many cases reliability is increased by expanding the robot's task program to use sensors to improve accuracy and to detect and recover from errors; code to handle sensors and errors often comprises 80% of the robot's task program. Unfortunately, development systems can not effectively test such code since sensor interaction can only be simulated.[7]

Error detection and recovery, then, depends on the experience, intuition, and common sense of the robot programmer.[8] The programmer is responsible for anticipating all the possible errors and for determining the actions to take to recover from them. Since not all the possible errors can be considered, a long testing cycle is still needed. This all makes developing and testing a robot program a long and painful job. It is common to spend between 3 and 6 man-months to develop a new, reasonably complex program. Automatic techniques for error detection and recovery can have important advantages in engineering savings and robot reliability.

IV. AUTOMATIC ROBOT ERROR RECOVERY

Recovering from execution errors is difficult because they typically involve discrepancies between what the system believes is true and what is true in the physical world. Discrepancies can occur because the robot itself has inherent inaccuracies, actions are not exactly reproducible, and there are real-time constraints. It is tricky to determine when a slight discrepancy constitutes an error as opposed to normal variation and whether a minor miscalculation will result in disruption or go unnoticed. Inappropriate decisions may be costly. Failures to detect an incorrect arm position may result in the arm crashing into a wall. On the other hand, a program that has worked well hundreds of times may fail because of a small difference in the size of one part, a difference that should be insignificant. The range of possible errors is so vast that it is impossible to anticipate and implement recovery strategies for each application. Automatic recovery systems must be able to reason about the problem in order to decide what to do.

Error recovery techniques are often classified as involving either *forward recovery* or *backward recovery*.[9] In both cases there is the notion that an error indicates a difference between the actual and the desired state of the system. Recovery consists of fixing the cause of the problem and then proceeding with system operation from a state known to be correct. In backward recovery the system recovers by "going back" to a previous state that is known to be correct. This usually involves undoing some actions performed previously. In forward recovery, the system performs actions that lead it to a state the system is supposed to eventually achieve. Forward recovery avoids undoing previous actions, but is more difficult to do since it requires an intelligent recovery system.

Backward error recovery has been used successfully in software systems where actions can be characterized purely by the state of information. The strategy associated with backward error recovery often involves the notion of rolling back the system to a known correct previous state. This approach works poorly in the robot environment because robot actions can not be reversed exactly. In some cases the attempt to undo some action may compound the problem by leading to further disruption of part

locations. Also some operations such as fastening might not be reversible by the available tools.

Since robot actions are not recoverable in the sense necessary for backward recovery, a form of forward recovery is more appropriate. Forward recovery consists of determining the difference between the actual state and the desired state, then developing a sequence of operations to achieve the desired state. The robot needs to recognize situations such as "dropped part" and decide, perhaps, to "ignore the dropped part and get a new one from the feeder." We need to figure out side effects of motion (collision of a part involved in a failure with another part, for example) in order to characterize the current state of the work cell.

In prior research on robot planning systems, much of the work has concentrated on detection and correction of errors in the robot plan rather than during the plan's execution.[10] Such work does not really represent either forward or backward recovery, since recovery may involve rewriting plan steps both ahead and behind the point of error in the plan. A few systems have attempted to detect unexpected situations during execution and do a form of forward recovery. STRIPS[11] was used to control Shakey, a mobile robot. Shakey was able to detect unexpected situations in its environment and to react by replanning its actions. Research with the JPL Robot also addressed robot error correction[12,13] including a specialized system for analyzing the causes of failures in robot programs and for replanning the robot activity. These studies all make several assumptions: knowledge about events is correct, there are no uncertain data, correct predicates are generated from sensor data every time they are needed, and enough knowledge is provided to take into account all the possible states of the environment. Current planning systems are too rudimentary and slow to solve the complex problems found in real applications of robots.

V. A SYSTEM FOR AUTOMATIC ERROR RECOVERY

We have designed a system that handles failures in robot programs performing parts manipulation and assembly. The system starts with a robot program and applies general knowledge about robot manipulation tasks to generate an expanded form of the program. This expanded form is then used to monitor and control the robot's operation in real time. If a failure is detected during the robot's operation, information collected during execution is passed to a recovery process that performs the forward recovery. The recovery process analyzes the failure and plans the sequence of robot operations necessary to correct the failure and proceed with the robot's task.

An overall view of the system components is given in Figure 1. The boxes correspond to procedures and the ovals correspond to data shared among procedures. There are three major parts to the process:

A. The *Preprocessor* creates a task description that is adequate for handling error analysis and recovery.
B. During execution the *AP Processor* tracks the robot's activities to detect errors if they occur.
C. If an error occurs, the *Recoverer* collects the facts necessary to plan a recovery and alters the robot's existing task to do the recovery steps.

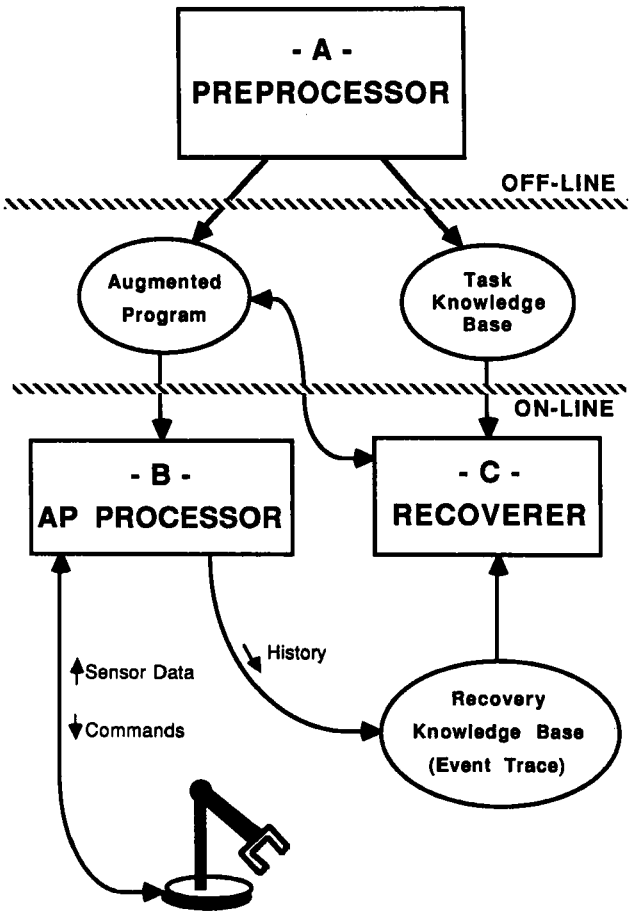

Figure 1. Organization for the intelligent robot recovery system.

We are interested in physical errors and faults in the robot work cell. These include errors by the robots themselves, tools, feeders, and other components. The errors we aim to detect include collisions, jammed parts, gripper slip, orientation and alignment errors, or missing parts. Our primary goal is to develop a framework in which the activity of error recovery can be automated without excessive overhead. Our main objective is to make the robot more robust so that there is less need for operator intervention during execution. Another important objective is to reduce the robot's programming time by shifting some of the burden of error detection from the programmer onto the robot system itself.[14]

The system starts with a working robot program; one that works correctly in reasonably typical situations on the robot in question. The system translates this working program into an augmented form that is used to monitor and control the robot's operation in real-time. Information is collected during execution that can then be used to analyze the failure and plan the recovery. Here is a description of the steps in detail.

First, the system analyzes the robot's task program off-line to generate a specialized description of the robot's task. This task description has two parts: the program describing the robot's task, and additional facts about what the task's intentions are and what physical objects (robots, parts, tools, etc.) the task uses. The first part is referred to as the *Augmented Program* (AP). The AP describes the sequence of actions the robot must perform, the sensor readings necessary to verify the performance, and information about how the robot's actions affect objects in the robot's work cell. The second part of the task description is called the *Task Knowledge Base*. This contains detailed information about the work cell layout and attributes of the robot and the parts.

Next, we monitor the execution of the robot program and we maintain a trace of events. Sensors are used to verify the robot's proper operation and to detect errors. Information in the AP about sensor usage is used to sift through sensor input and to extract relevant data. Relevant sensor data is used to invoke subsequent robot actions and to trace robot activities.

Once an error has been detected, the recovery system uses the event trace and information about the task to determine the causes and effects of the error. This information is then used to build an appropriate recovery procedure. The steps in the recovery procedure are then appended to the existing AP and the AP Processor is restarted with the first step of the recovery procedure.

VI. THE AUGMENTED PROGRAM

The Augmented Program is the internal representation of the robot's task. The primary purpose of the AP is to relate the robot's expected position and sensor readings to its expected operation. This gives us a basis for collecting useful information on the robot's operation for recovery if an error occurs. In addition, the AP includes knowledge about how the robot's actions are related to changes in the robot's work cell and the progress of the robot's task. For reasons of efficiency and ease of modification the AP is structured internally as a finite automaton.

The AP provides the interface between the knowledge-based components of the system and the real-time components. In addition to the command, sensory, and procedural information necessary to control the robot the AP contains information about how the robot's actions affect the objects in the work cell and the goals of the robot's task. This gives the AP some higher level knowledge about the robot's operation without burdening it with time consuming inference procedures. The AP structure is designed to be interpreted efficiently in real-time.

The AP incorporates two kinds of information not obvious in the original AL program: sensor information and implications about objects in the workspace. This information is extracted from the original program and from the programmer by the Preprocessor as described elsewhere.[15,16] The sensor information in the AP is used to guide the real-time system in its use of sensors. The AP specifies which sensors need to be monitored during each step of the robot's task and often identifies specific sensor values that are meaningful. All unspecified sensor readings can be ignored by the system, thus saving computation time and expense. The higher level knowledge contained in the AP is not used to directly control the robot. This knowledge is saved in

an *event trace* and used if an error occurs. By interpreting the event trace the Recoverer can determine how the robot was expected to interact with objects in its work cell and what the robot intended to do.

Another requirement is that the AP must be in a form that can be modified dynamically by the Recoverer. When a failure has been analyzed and a recovery plan devised, the Recoverer needs to add new instructions to the AP. By executing these new instructions the robot will bring the work cell into a valid state so that it can continue execution. We do not want to have to recode and recompile the robot's entire task program; we prefer a representation that allows careful patching. This process is described in a later section.

The AP represents the sequence of actions in the robot's task as a finite automaton. Note how the automaton representation provides a natural way to map events to actions. Significant sensor readings are treated as "tokens" that may cause state transitions and actions related thereto. The discrete states in the automaton representation also provide a useful way of identifying separate parts of the robot's task for automated reasoning and analysis. Structuring the task as an automaton also makes it easy to modify; new actions can simply be added as new state transitions that eventually lead to previously existing ones.

Figure 2 gives a short example of an augmented program. Figure 2(a) is a robot program written in AL[3] that tells the robot to grasp and lift an object. The corresponding AP is given in Figure 2(b). Each state in the AP contains no more than one robot operation; the waiting time between state transitions usually corresponds to waiting time between the initiation and completion of a robot motion. The AP has two extra states: one for successful completion (state 5 in the example) and one to enter if an error occurs (state 6 in the example).

Automaton representations have been popular for many years in a number of applications including lexical analysis, data communications,[17] and interactive graphics,[18] as well as industrial process control.[19] Automata are popular in applications with a broad range of possible input sequences that must all be handled quickly and predictably. Systems with a small number of states can usually be analyzed and validated simply by drawing and studying the state diagram. Another feature is that control procedures described as automata can be implemented in a variety of ways: sequential circuits, microcode, or software.[17]

Perhaps the only (or the most serious) drawback of an automaton representation is that it may be hard for humans to comprehend. State diagrams are reasonably easy to understand when there are a small number of states, but textual representations typically reduce to a language with no control structure except **IF** ***p*** **GOTO** ***q***. A robot pro-

```
begin
  open minihand to 5;
  move miniarm to frame (rot(xhat,-90), vector(4,20,2));
  center miniarm;
  move miniarm to frame (rot(xhat,-90), vector(14,20,12));
end;
```

Figure 2(a). Short AL program that picks up a cube.

```
(setq grab.ap
    '([1  ((robot-do open mini 5.0))
       ((open mini 5.0) 2)
       ((hand-error mini) 6)]

      [2  ((robot-do move mini (-90 90 90 4 20 2)))
       ((reach mini (-90 90 90 4 20 2)) 3)
       ((joint-error mini) 6)

      [3  ((expect grasp mini obj_block) (robot-do center mini))
       ((center mini) 4)
       ((hand-error mini) 6)]

      [4  ((imply grasp mini obj_block) (expect carry mini obj_block)
           (robot-do move mini (-90 90 90 14 20 12)))
       ((reach mini (-90 90 90 14 20 12)) 5)
       ((joint-error mini) 6)
       ((a-untouch mini) 6)
       ((b-untouch mini) 6)]

      [5  ((imply idle mini) (imply done) (robot-do end))]
      [6 ((imply error) (robot-do end))]))
```

Figure 2(b). Augmented program representation of the short AL program.

gramming language designed like this would not represent progress. For this reason, the automaton is used as an *internal* representation generated from a procedural programming language, in our case, AL.

The mapping of the AL program into an automaton is as follows:

- Statements in the original program are split and regrouped into actions that are executed when the automaton makes the appropriate state transition.
- Sensor readings that verify the completion of an action are used to trigger state transitions by the automaton. If a sensor reading verifies successful completion of an action, the transition leads the AP Processor to the next instruction in the program. If the sensor reading indicates that the action failed, the transition leads the AP Processor into an error state.
- When entering a new state, the AP Processor executes the actions associated with that state. It then sits idle in that state until it detects one of the sensor readings it is waiting for. The AP Processor then performs another state transition.

Consider the automaton diagram (Fig. 3) that represents the first few instructions of the example program from Figure 2. Significant sensor readings are indicated by predicates such as **(reach *rob pos*)** and **(open *rob wid*)**. A transition, then, is caused when a continuously tested predicate such as those becomes true. We generalize the

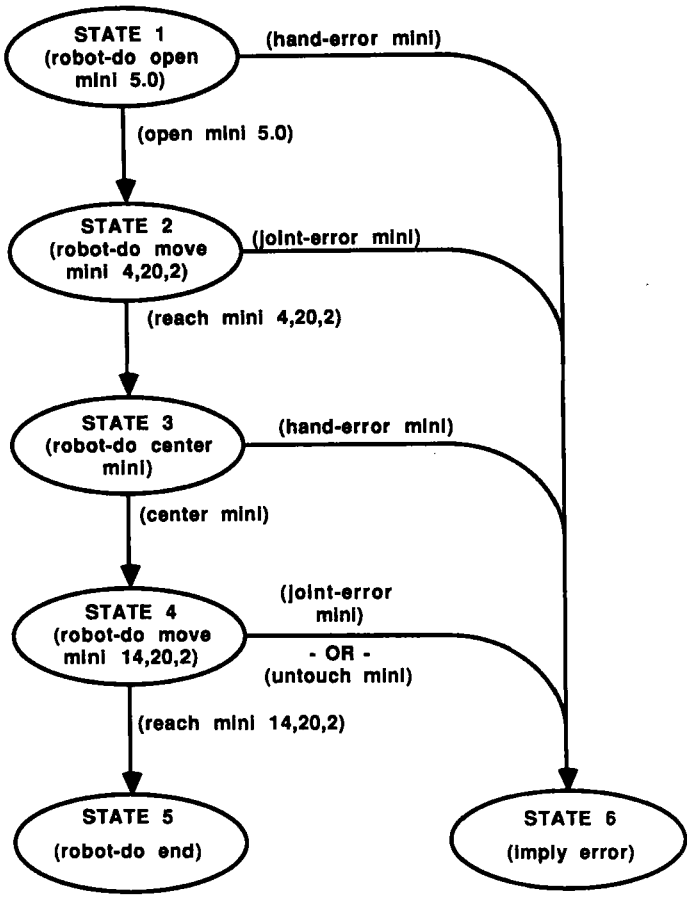

Figure 3. State diagram for the example AP.

form of these predicates so that transitions may also be caused by testing user-defined variables in the original program. This makes it a simple matter to represent the control structures that occur in typical procedural programming languages.

The syntax for augmented programs is given in Figure 4. Augmented Programs are represented as Lisp forms. Each state is represented by a state clause that corresponds to the state's circle in the state diagram and the arcs leaving that state. The state clause is prefixed by a number representing that state. Next is a list of actions to do when the state is entered. This list of actions may contain a robot command and some computation. The rest of the state clause is a list of event predicates. Each event predicate consists of a predicate and an AP state number. The predicates describe sensors to check while in the enclosing state. All such predicates are tested concurrently while the AP is in the enclosing state. If a predicate tests true, the AP Processor enters the new state specified by the state number at the end of the event predicate.

```
<AP>              ::=   ( <state clause> { <state clause> } )
<state clause>    ::=   ( <state number> <actions> { <event predicate> } )
<state number>    ::=   integer
<event predicate> ::=   ( <predicate> <state number> )
<actions>         ::=   ( [ (compute forms ) ]
                        { (imply args ) }
                        { (expect args ) }
                        [ (robot-do args ) ] )
```

Figure 4. Syntax of an augmented program.

The ⟨actions⟩ section of a state may contain any of the following forms. The only significant restriction is that there must be only one **robot-do** form specifying a single robot action.

compute Gives a list of forms that are evaluated at execution time. These forms usually contain on-line computations that control the robot's activities. For example, if the robot has to store objects in a sequence of positions, we would use **compute** forms to specify the arithmetic that computes new object positions.

imply Specifies facts about the workspace that are implied by the original program. Such facts are usually derived from *intentions* recognized during the pre-processing phase. Typically these forms identify when a particular robot arm is carrying a particular kind of object.

expect Specifies what we expect to happen by the end of the current state if all goes correctly. These usually match the contents of **imply** clauses appearing in the next state we expect to enter.

robot-do Specifies commands to be given to the robot.

The history of the robot's activities is stored in the Event Trace. The AP Processor generates the Event Trace automatically from data stored in the AP as the process goes from state to state. Each trace entry is prefixed with the time at which the entry was generated. When an event predicate tests true, the following information goes into the Event Trace:

- The sensor readings implied by the successful predicate are added to the Event Trace.
- The number identifying the new AP state entered is written to the Event Trace in the form (*ttt* **new-state** *nnn*).
- If the event predicate contains an **imply** form, the contents of the form are added to the Event Trace.
- If the event predicate contains an **expect** form, its contents are placed in a list of expectations in the Event Trace, prefixed by the keyword **expect.**
- The expected result of the action performed in a robot-do form are added to the list of expectations that appear in the Event Trace.

```
(setq event-trace
    '((0 initialize)
      (0 new-state 1)
      (0 expect (open mini 5.0))
      (10 sense open mini 5.0)
      (10 new-state 2)
      (10 expect (reach mini (-90 90 90 4 20 2)))
      (27 sense reach mini (-90 90 90 4 20 2))
      (27 new-state 3)
      (27 expect (grasp mini obj_block) (center mini))
      (36 sense center mini)
      (36 sense open mini 4.0)
      (36 new-state 4)
      (36 imply grasp mini obj_block)
      (36 expect (carry mini obj_block) (reach mini (-90 90 90 14 20 12)))
      (48 sense reach mini (-90 90 90 14 20 12))
      (48 new-state 5)
      (48 imply idle mini)
      (48 imply done)))
```

Figure 5. Event Trace from successful execution of the example AP.

Figure 5 shows what the event trace would look like after successfully performing the task described in Figure 2. A trace of an unsuccessfully executed program would look similar except that an (*ttt* **imply error**) entry would appear near the end of the trace after the last state transition. This is discussed in more detail in a later section.

This trace illustrates some important points about the AP and the AP Processor. For example, notice how the clock times match up between various trace entries because state transitions are seen as instantaneous events. Also notice how the **sense** entries do not always exactly match the corresponding event predicates in the AP, particularly between states 3 and 4. This is because the hand width is an extra piece of information we get from performing a **center** operation and is reported separately from the completion of the **center** itself. We see something similar happen if an error predicate such as (*ttt* **sense untouch-a** *rob*) became true, since such a predicate needs to report the hand's location as well as the sensor change.

VII. AP PROCESSOR STRUCTURE

The AP Processor is responsible for controlling the execution of the robot's task and for interfacing with the error recovery system. Its primary activity is to collect position and sensor data and map it into events that cause state transitions. This requires some form of economical multiprogramming to support multiple robots and sensors. All activities must be handled rapidly enough to provide real-time response to state transition events. The AP Processor must also maintain history of robot's operation

and, if an error occurs, interface to Recoverer for modification and restart after an error.

A diagram of the AP Processor is given in Figure 6. Boxes represent programs or tasks and ovals represent data passed between tasks. Everything involved in the real-time operation of the robot is shown in the diagram. The AP specifies the operations the robot is to perform and the sensor readings necessary to verify success. The robot's task (and related subtasks, if required) generate commands given to the robot and *sensor expectations* passed to the *sensor filter* tasks. The sensor filters then monitor the sensor readings for significant readings as specified in the expectations. Significant

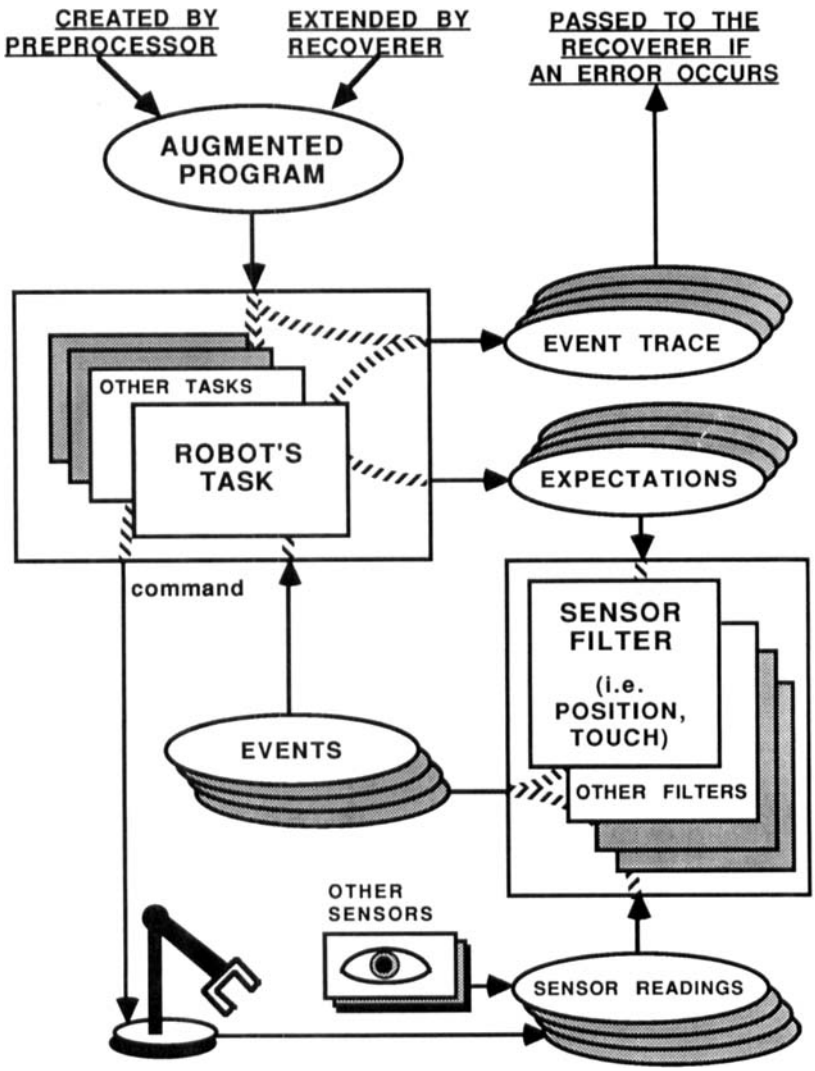

Figure 6. AP Processor structure.

readings generate *events* that then cause state transitions in the robot's task. Commands and events are placed in the event trace along with information from the AP about objects and intents.

The AP may contain multiple parallel processes. We implement this by doing nonpreemptive scheduling between active processes. Each time a process does a state transition the process executes the transition actions to completion before another transition may occur. Once the process completes its transition the scheduler seeks another process ready to do work and dispatches it. Individual actions are purposely kept short, similar to the *strip* software architecture in the Pluribus multiprocessor.[20] All processes in the AP Processor, both those represented by APs and those that process sensor data, are implemented as a sequence of striplike procedures. This approach worked on the Pluribus and has worked in other environments such as a real time speech recognition terminal.[21]

Event predicates in the current state of the AP tell the sensor filters what kinds of sensor readings are expected. The sensor filters insure that the appropriate sensors are active and filter the resulting sensor data. If one of the expected sensor readings appears, the sensor filter generates an event that can then cause a state transition in the robot's task.

Robot tasks specified by AP are dispatched in response to events. This typically causes the task to perform an AP state transition, which takes place as follows:

- Cancel all sensor expectations associated with the task's previous state.
- Use the event predicates in the new AP state to instruct the sensor filters about the new state's sensor expectations.
- Perform any numerical computations required in the current state.
- Issue a command to the robot if required in this state.

After performing these steps the task sits idle until one of its expected events occur. When an event occurs, the dispatcher finds what task is waiting for it and causes the task to perform the appropriate state transition.

VIII. AUTOMATIC ERROR RECOVERY

When an error occurs the AP Processor suspends its normal operation and passes the event trace to the Recoverer. The Recoverer analyzes the error and determines the steps necessary to perform forward recovery. The Recoverer then translates the steps into AP states and appends them to the existing Augmented Program in the AP Processor. The Recoverer then restarts the AP Processor with the initial recovery state so that it performs the appropriate recovery actions and then proceeds with its original task.

Figure 7 gives a graphical representation of the relationship between the states in the original AP and the new states that perform the forward recovery. The original states are unchanged. The new states are appended to the end of the original AP and include transitions that lead into states of the original AP. Typically, the recovery steps will perform a sequence of operations that undo any serious disruption in the robot's work cell and then proceed with the original AP right after the point of the

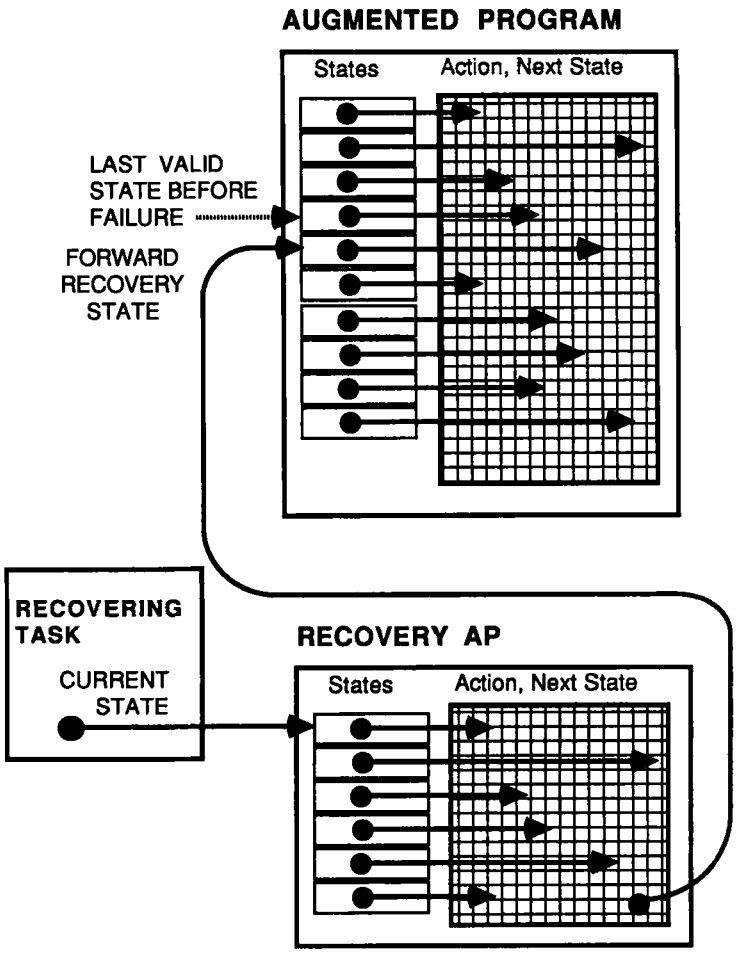

Figure 7. Recovering from a failure.

error. If an error occurs during the recovery, the error state in the original AP will be entered.

Consider the following example. Assume that the robot is performing the program described earlier in Figure 2 and that the object being gripped slips out as it is carried to its destination. When the object leaves the gripper one of the touch sensors will indicate that it is no longer in contact with the object and generate an **untouch** event. Looking at the AP in Figure 2(b) we can see that the robot's task would be in state 4 and that the **untouch** event will cause a transition to state 6, the error state. The AP Processor then stops and passes its event trace to the recoverer. The event trace corresponding to this error is given in Figure 8(a). The Recoverer takes the event trace and combines it with information from the Task Knowledge Base to diagnose the problem and plan the recovery.

To effect the recovery, the Recoverer provides the AP Processor with a new set of state clauses to append to the existing AP. The AP Processor then proceeds by restarting

at the first state added to the AP. The steps of the recovery procedure can reset variables, move the robot arm, and respond to sensor readings the same as in the original parts of the AP. Figure 8(b) shows the event trace of recovering from the dropped object, leading ultimately to the completion of the original task.

IX. RECOVERY ANALYSIS AND PLANNING

Two important aspects of a forward recovery system are error analysis and recovery planning. This is still an area of research and experimentation in our system. So far we have restricted ourselves to simple techniques so that we could test other portions of our system. Our approach tries not to unnecessarily restrict our capabilities for analysis or planning.

Initial work has been with simple recovery heuristics that depend on simple analysis of single failures. One such failure is the *safe variation* failure in which the system detects a transient error. Recovery from such failures does not require action by the robot; the system simply has to restart the robot at some earlier state. A temporary wobble of the robot gripper is one example. Another example is when a task involves small parts acquired from a parts feeder and a part is dropped. The system can essentially "backwards recover" by fetching a new part and ignoring the dropped one.

The failure recovery in Figure 8 involves a better but still simple heuristic applied to dropped parts. The heuristic is that parts fall straight down, so a dropped part can be found directly under the place where the gripper detected it drop. Figure 9 gives the AP recovery steps used to perform this recovery attempt on the failure traced in Figure 8(a). Figure 8(b) gives the resulting event trace.

These examples are useful for early experiments but do not yield useful insight into

```
(setq before-trace
    '((0 initialize)
      (0 new-state 1)
      (0 expect (open mini 5.0))
      (10 sense open mini 5.0)
      (10 new-state 2)
      (10 expect (reach mini (-90 90 90 4 20 2)))
      (27 sense reach mini (-90 90 90 4 20 2))
      (27 new-state 3)
      (27 expect (grasp mini obj_block) (center mini))
      (36 sense center mini)
      (36 sense open mini 4.0)
      (36 new-state 4)
      (36 imply grasp mini obj_block)
      (36 expect (carry mini obj_block) (reach mini (-90 90 90 14 20 12)))
      (43 sense untouch-b mini)
      (43 sense reach mini (-90 90 90 8 19 5))
      (43 new-state 6)
      (43 imply error)))
```

Figure 8(a). Event trace of the robot dropping the object.

```
(setq after-trace
    '((200 restart)
      (200 new-state 7)
      (200 expect (open mini 5.0))
      (210 sense open mini 5.0)
      (210 new-state 8)
      (210 expect (reach mini (-90 90 90 8 19 2)))
      (231 sense reach mini (-90 90 90 8 19 2))
      (231 new-state 9)
      (231 expect (grasp mini obj_block) (center mini))
      (243 sense center mini)
      (243 sense open mini 4.0)
      (243 new-state 10)
      (243 imply grasp mini obj_block)
      (243 expect (carry mini obj_block) (reach mini (-90 90 90 14 20 12)))
      (252 sense reach mini (-90 90 90 14 20 12))
      (252 new-state 5)
      (252 imply idle mini)
      (252 imply done)))
```

Figure 8(b). Event Trace of the recovery.

the handling of more complex recovery problems. The simple approaches are simple partially because error interpretation is closely tied to the recovery procedure. If we diagnose a "safe variation" we immediately know what the recovery procedure is: we do nothing. Recovery from the dropped part consists of simply inserting the gripper location and correct destination into an AP similar to that in Figure 9. Actual failures

```
(setq patch.ap
    '([7  ((robot-do open mini 5.0))
          ((open mini 5.0) 8)
          ((hand-error mini) 6)]

      [8  ((robot-do move mini (-90 90 90 8 19 2)))
          ((reach mini (-90 90 90 8 19 2)) 9)
          ((joint-error mini) 6)

      [9  ((expect grasp mini obj_block) (robot-do center mini))
          ((center mini) 10)
          ((hand-error mini) 6)]

      [10 ((imply grasp mini obj_block) (expect carry mini obj_block)
             (robot-do move mini (-90 90 90 14 20 12)))
          ((reach mini (-90 90 90 14 20 12)) 5)
          ((joint-error mini) 6)
          ((a-untouch mini) 6)
          ((b-untouch mini) 6)]))
```

Figure 9. Recovery steps added to the AP after the error.

are seldom so well behaved and manifest themselves in a variety of ways. The Recoverer must be able to collect and reason about a broad range of facts and sensor data.

For example, suppose that the robot was moving its arm and that the gripper's touch sensor unexpectedly came in contact with something. The Recoverer must use information about the intent of the robot's task to disambiguate the sensor reading. If the robot's intention was to try to grasp a part, then the sensor reading indicates several possibilities including temporary wobble of robot gripper, misorientation of expected part, or collision with an unexpected object. On the other hand if the robot was actually carrying a cube, the sensor reading may imply that the gripper dropped the cube. The context of the sensor reading plays an important role in classifying its meaning.

The Recoverer uses a technique called *failure reason analysis* to help relate sensor readings to errors in the robot's program. This technique was originally described in,[13] and we have developed an enhanced version of it.[16] This technique allows the Recoverer to use available information to locate all possible origins of an error in a systematic way. Our enhanced version also takes propagation of errors into account.

The failure reason model we describe only applies to the objects and situations directly related to the sensor reading indicating the error. For example, consider a task where a robot moves cubes from a feeder to a shipping pallet, 12 at a time. What might happen if the cube falls from the gripper and lands on the pallet, knocking another object off? In a simple failure reason model the robot only associates an error with a part if it uses its sensors on the part and finds an error. The lost part won't be missed until someone down the line tries to unload the pallet and finds it one part short.

A promising approach to such problems is to apply techniques of qualitative reasoning about physical space and motion.[22,23] Qualitative techniques give us a more powerful way of deducting possible problems in the work cell implied by sensor readings. These techniques also give us the tools to decide where and how we might use sensors to find out exactly what has happened in the work cell.

General solutions to problems of space and motion are complex and often intractable. We look for effective results by restricting the problem in two ways. First, we deal with problems in the constrained environment of a robot work cell. Information is available at many levels of detail about the objects in the work cell. The number and locations of objects is constrained by the task at hand. The second restriction is that we are not trying to get exact solutions. The physical reasoning techniques simply tell the system what to look for; sensors are then used to find more exact information about location and orientation.

Our approach revolves around a qualitative structural model of the robot work cell. The model starts with the empty work cell described by the Task Knowledge Base and uses the Event Trace to determine the work cell's current contents and configuration. Sensor readings from the Event Trace are used to constrain the probable locations of objects. Models of idealized work cell objects from the Task Knowledge Base provide assumptions that fill in additional details about the workspace. Intentions given in the Event Trace let the model represent the differences, if any, between expected and actual sensor readings. Physical laws (i.e., gravity, friction, conservation of energy) help identify uncertainties in the work cell, particularly when errors are detected.

Given information from the failure reason analysis and from qualitative reasoning about the work cell the Recoverer should have sufficient data to plan a recovery strategy. The available information will indicate the current condition of the work cell and the differences between its current condition and that required to continue with the robot's task. The Recoverer will also know where the AP Processor stopped and be able to determine the intent of its activities at the time of the failure. From this information it will be able to plan the recovery process.

If a recovery plan fails, the recovery process begins again. In a sense the Recoverer doesn't have to differentiate between a normal failure and a failure during recovery except that it must not lose sight of the intent of the task that originally failed. In many cases recovery may be easier the second time through since there may be more information about the state of the work cell. Repeated failures may serve as a technique for the Recoverer to experiment with the condition of the work cell and collect information with which to plan a thorough recovery.

X. SUMMARY AND ACKNOWLEDGMENTS

We have presented a method of implementing intelligent robot recovery within a framework that permits real-time robot operation. Recovery is effected by analyzing the state of the robot work cell and planning a sequence of operations that allows the robot to continue its original task after the point of failure, i.e., forward recovery. The system segregates automated reasoning in the system so that it is applied when time constraints are not a problem.

We have developed prototype and simulated components of the system in Franz Lisp on the Unix timesharing system at the Artificial Intelligence Laboratory at the University of Minnesota. We are presently developing a working implementation at the Robotics Laboratory at the University of Minnesota. This implementation uses a LMI Lambda processor for automated reasoning applications and a separate, stand-alone microcomputer to host the AP Processor. The robots controlled include an IBM 7565 and several smaller educational robots. Partial support for this work is gratefully acknowledged to the Graduate School and to the Productivity Center of the University of Minnesota.

References

1. T. Lozano-Perez, "Robot Programming," *Proc. IEEE*, No. 7, **71**, 821–841 (1983).
2. J. D. Millar, "Request for Assistance in Preventing the Injury of Workers by Robots," DHHS (NIOSH) Publication No. 85-103, National Institute of Occupational Safety and Health, Cincinnati, Ohio, December, 1984.
3. M. S. Mujtaba and A. Goldman, "AL Users' Manual," Memo AIM-323, Stanford Artificial Intelligence Laboratory, Stanford, California, 1979.
4. B. E. Shimano, C. C. Geschke, and C. H. Spalding III, "VAL II: A New Robot Control System for Automatic Manufacturing," *Proceedings of the IEEE International Conference on Robotics*, Atlanta, Georgia, March, 1984, pp. 278–279.
5. G. Larson and M. Donath, "Animated Simulation of Intelligent Robot Workcells," Productivity Center, Department of Mechanical Engineering, University of Minnesota, Minneapolis, Minnesota, 1985.

6. M. S. Pickett, R. B. Tilove, and V. Shapiro, "Roboteach: An Off-line Robot Programming System Based on GMSolid," Research Publication GMR-4465, General Motors Research Laboratory, Warren, Michigan, October, 1983.
7. A. P. Ambler and others, "An experiment in the offline programming of robots," *Proceedings of the 12th International Symposium on Industrial Robots*, Paris, France, June, 1982, pp. 491–504.
8. G. Gini and M. Gini, "Dealing with World Model Based Languages," *ACM Trans. on Programming Languages*, vol. 7, no. 2, pp. 334–347, April 1985.
9. B. Randell, P. A. Lee, and P. C. Treleaven, "Reliability Issues in Computing System Design," *Computing Surveys*, Vol. 10, no. 2, pp. 123–165, June 1978.
10. E. Sacerdoti, *A structure for plans and behavior*, American Elsevier Publ. Company, 1977.
11. R. E. Fikes and N. J. Nilsson, "STRIPS: A New Approach to the Application of Theorem Proving to Problem Solving," *Artificial Intelligence*, Vol. 2, pp. 189–208, 1971.
12. L. Friedman, "Robot Learning and Error Correction," *Proceedings of the 5th International Joint Conference on Artificial Intelligence*, Cambridge, Massachusetts, p. 736, 1977.
13. S. Srinivas, "Error Recovery in Robot Systems," Ph.D. Thesis, California Institute of Technology, Pasadena, California, 1977.
14. M. Gini and G. Gini, "Towards Automatic Recovery in Robot Programs," *Proceedings of the 8th International Joint Conference on Artificial Intelligence*, Karlsruhe, West Germany, August 1983, pp. 821–823.
15. M. Gini, R. Doshi, M. Gluch, R. Smith, and I. Zualkernan, "The Role of Knowledge in the Architecture of a Robust Robot Control," *Proceedings of the 1985 IEEE Conference on Robotics and Automation*, St. Louis, Missouri, March, 1985.
16. M. Gini, R. Doshi, S. Garber, M. Gluch, R. Smith, and I. Zualkernan, "Symbolic Reasoning as a Basis for Automatic Error Recovery in Robots," Technical Report No. 85-24, Computer Science Department, University of Minnesota, Minneapolis, Minnesota, August 1985.
17. D. Bjorner, "Finite state automaton—Definition of Data Communication Line Control Procedures," *Proceedings of the AFIPS 1970 Fall Joint Computer Conference*, Vol. 37, AFIPS Press, Montvale, New Jersey, 1970, pp. 477–491.
18. B. H. Barnes, "An Automata Theoretic Approach to Interactive Computer Graphics Command Languages," in *Applied Computation Theory: Analysis, Design, Modelling*, Raymond T. Yeh, ed., Prentice-Hall, Englewood Cliffs, New Jersey, 1978.
19. J. V. Landau, "State Description Techniques Applied to Industrial Process Control," *Computer*, **12**(2), 32–40 1979.
20. S. M. Ornstein, W. R. Crowther, M. F. Kraley, R. D. Bressler, and A. Michel, "Pluribus-A Reliable Multiprocessor," *Proceedings of the AFIPS 1975 National Computer Conference*, AFIPS Press, Montvale, New Jersey, 1975.
21. A. Stowe, S. Glazer, and R. Smith, "A Language and Multi-Tasking Operating System to Support an Eight-Channel Speech Input Terminal," paper given at the 50th Anniversary Meeting, Acoustical Society of America, Cambridge, Massachusetts, 1979.
22. D. G. Bobrow, *Qualitative Reasoning about Physical Systems*, MIT Press, Cambridge, Massachusetts, 1985.
23. K. D. Forbus, "A Study of Qualitative and Geometric Knowledge in Reasoning about Motion," M.S. Thesis, AI-TR-615, MIT Artificial Intelligence Laboratory, Cambridge, Massachusetts, February 1981.