

**A Knowledge-based System
for On-line Robot Error Recovery**

Martin Boyer

B. Eng.

Department of Electrical Engineering
McGill University

A thesis submitted to the Faculty of Graduate Studies and Research

in partial fulfillment of the requirements for the degree of

Master of Engineering

July 1988

© Martin Boyer

Abstract

This research investigates the problems associated with manipulation and, to a certain extent, programming errors in a shared operator/computer control of a robot system. The principle is to trace all actions, at run time, to provide on-line detection and recovery of errors. A world model is constructed and maintained for the purpose of predicting the effects of actions and signaling errors when the actual outcome of an action differs from its required effect. Default reasoning is used extensively to speed up processing and compensate for the high cost of sensing. After a task planner has dealt with the general organisation of the program, the system presented here has the responsibility of coping with variations of the real world to attain the desired goal with the given plan.

A test case, overhead power line maintenance, demonstrates the functioning of the system and, although the work is based on this particular context, the scheme described comprises a generic "substrate" which deals with common basic robot actions—such as move and grasp and is supplemented by task and environment specific knowledge such as which parts can be mated, sizes, and weights. This part of the system is static for a given task and a good portion of it, the substrate, is valid for a wide range of tasks.

Résumé

Cette recherche investigate les problèmes associés aux erreurs de manipulation et, jusqu'à un certain point, de programmation d'un manipulateur robotisé, en commande partagée entre un opérateur et un contrôleur intelligent. À la base, toutes les actions sont suivies, au moment de l'exécution, pour permettre la détection et le recouvrement des erreurs en ligne. Un modèle du monde est construit et maintenu afin de prédire les effets des actions et de signaler une erreur lorsque le résultat d'une action diffère de l'effet requis. Un raisonnement par défaut est utilisé de façon systématique pour accélérer le traitement et pour compenser le coût élevé d'utilisation des capteurs. Une fois la tâche planifiée, le système présenté ici a la responsabilité de s'adapter aux variations de l'environnement afin d'atteindre le but désiré au moyen du plan original.

Un cas type, l'entretien de lignes de distribution électrique, démontre le fonctionnement du système et, quoique développé dans ce contexte particulier, le système comprend un "substrat" générique. Ce substrat décrit les manipulations de base, telles les déplacements et l'action de la pince, et comprend des connaissances spécifiques à la tâche et à l'environnement, telles que les pièces pouvant être jointes, les dimensions et les masses. Cette partie du système ne varie pas pour une tâche donnée et une bonne part, le substrat, est valable pour un grand nombre d'applications.

Acknowledgements

I wish to thank first my thesis supervisor, Doctor Laeeque Daneshmend, for providing many of the original ideas, careful guidance, and for generally doing more than he had to. I also extend my gratitude to the students and staff of the Computer Vision and Robotics Laboratory. Mike Parker in particular, for help on numerous technical problems and many comments on human interfaces.

Je tiens également à remercier mon co-directeur à l'Institut de recherche d'Hydro-Québec, le Docteur Pierre Girard, un homme éclairé qui, par sa grande ouverture d'esprit et sa confiance, a rendu mon séjour à l'IREQ des plus agréables et stimulants. J'exprime aussi ma reconnaissance envers l'Institut pour un important support financier et technique, de même qu'envers le personnel du laboratoire de robotique, tout spécialement Jean Lessard pour m'avoir supporté durant plus de deux ans, mais surtout pour de riches discussions sur les besoins et possibilités de la télérobotique. En dernier lieu, je remercie mes parents ainsi que Louis et Manon, pour leurs encouragements et leur amitié au cours d'un hiver difficile, sans lesquels cette thèse n'aurait jamais été complétée.

Contents

<i>List of Figures</i>	<i>viii</i>
Chapter 1 Introduction	1
1.1 The Problem	1
1.2 Previous Work/Theory	2
1.2.1 Pre-execution Planning and Verification	2
1.2.2 Failure Reason Analysis	4
1.2.3 Object Oriented Programming	7
1.3 Motivation	8
1.3.1 Applications	8
1.3.2 Replanning versus Error Recovery	9
1.3.3 What Is ISER?	11
1.4 Thesis Overview	12
Chapter 2 Problem Description and Representation	13
2.1 Task Description	13
2.1.1 The Operator	14
2.2 Typical Task	14
2.2.1 Actions and Associated Errors	14
2.2.2 Level of Representation	16
2.3 Frame Representation	17
2.3.1 Objects	18
2.3.2 Tools	21
2.3.3 Actions	23
2.3.4 Where is the Knowledge?	26

2.3.5 ART: The Automated Reasoning Tool	26
2.4 Domain Independence	27
Chapter 3 A Dynamic World Model	28
3.1 Effects of Actions	28
3.2 Use of Sensors	30
3.3 Action Trace	31
3.4 High-level Parser	31
Chapter 4 Error Detection and Analysis	33
4.1 Error Types	33
4.2 Error Sources	35
4.3 Run-time Error Detection	37
4.4 Post-mortem Analysis	38
4.4.1 The Failure Tree	39
4.4.2 Ultimate Source of Error	40
4.5 Complexity Analysis	40
Chapter 5 Error Recovery	42
5.1 Error Recovery Algorithm	42
5.1.1 Constraining the Search Space	44
5.2 Recovery Propagation	46
5.3 Example	47
5.4 Recovery Cost	50
Chapter 6 Conclusion	58
6.1 Summary and Discussion	58

6.2 Contributions.....	59
6.3 Future Work.....	60
Appendix A. Sample ISER Session.....	62
Appendix B. The High Level Parser	67
<i>References</i>	68

List of Figures

1.1	Replanning versus Error Recovery	10
2.1	Objects classes in ISER.	19
2.2	Action classes in ISER.	24
5.1	The Action Search Space	45
5.2	Error Recovery	46
5.3	Perfect and Actual plans	53
5.4	Program Costs as function of the Number of Steps	56
A.1	Typical ISER display	63
A.2	The ISER menu.	64
A.3	The conductor object, initial state.	64
A.4	The conductor object, goal state.	64
A.5	Error analysis state	66

Chapter 1

Introduction

1.1 The Problem

Robot manipulators are increasingly used in uncontrolled environments. This requires the coordination of complex systems, using multiple tools and sensors, executing more complex tasks, and driven by intelligent controllers. Often, such environments are associated with non repetitive tasks, such as maintenance and repair, usually assisted by a human operator. This form of robotics, *telerobotics*, is receiving more and more attention for space applications [Sheridan86, Will85] and nuclear reactor maintenance [Moya86]. In those cases where an operator is "in the loop", the original plan can be discarded whenever the operator sees a better way to accomplish the task or when an error occurs while performing the task. Hence the need for on-line planning and error recovery or, in a more general sense, a way for the operator to interact with the planner and the capability to recover from failures due to departures from the original model of the world. Kaemmerer and Allard have described an on-line system to provide such an interface for process control [Kaemmerer87] and Lee et al. discussed the errors associated with small batch production, due to the inherent flexibility involved [Lee84]

To provide adaptiveness to changing world conditions, a usual approach is to replan, as if the error state was a "normal" initial state, to achieve an unchanged goal state. This creates three problems: the first one being a cost to be paid because of the

difficulties inherent to planning. Second, neither the system nor the operator learn from their errors. A third problem might occur since blind re-planning may very well produce the same failure that triggered the need for re-planning, without any useful error message. Pinpointing the source of the error would ease error recovery by reducing the complexity and amount of recovery necessary; if the cause of the error can be "removed" by simple local planning, then no global understanding of the intention of the plan is needed and automatic, straightforward recovery can be accomplished. Furthermore, a great portion of the original plan can be salvaged, reducing the replanning and repair time.

Errors occur at different stages in the conception and execution of a robot task: Before execution, the plan can be incorrect and not correspond to the task specification. During execution, incomplete or incorrect knowledge of the world causes the wrong parameters to be used. And in general, robots are still imperfect and there is not yet any means to adequately model the environment in which they operate. This work does not specifically address the first problem, that of incorrect plans, but deals with the last two problems in an original way; instead of attempting to resolve all necessary information before execution of a plan by a "perfect" robot, the errors are allowed to manifest themselves (assuming the plan is generally good, there should not be many errors). Error recovery is then performed by finding the source of the error and using local planning, with a better understanding of the environment, to remove any side effects of the failure and re-execute the failed action. The key point here is that it can be more efficient to let a few non-catastrophic errors occur and correct them than to try to prevent all errors by systematically performing exhaustive sensing, because the planner cannot foresee very well in uncontrolled environments and hence the recovery procedure must be defined at run-time.

1.2 Previous Work/Theory

1.2.1 Pre-execution Planning and Verification

Brooks [Brooks82] developed a plan checker to take explicit account of errors

and automatically modify a given plan to include sensing and to guarantee its success. More precisely, all errors can be treated as uncertainties and Brooks states that it is possible to make inferences about uncertainties and to use those inferences in computation. The major advantage of this approach is the use of symbolic rather than numeric computation, so that uncertainties in object position and size can be used to infer required initial tolerances or the necessity for sensing. This is one of the most elegant and mathematically sound statements of the problem of robot operation in presence of errors. It has practical limitations, though, in highly uncertain environments when, for instance, it is rather difficult to estimate uncertainty in image analysis and object recognition; in effect, the uncertainties are so large — even after sensing — that the calculated tolerances are meaningless¹. As a matter of fact, the system is intended to be applied to industrial assembly operations, where tolerances are usually a few percentage points of the nominal value, not to unstructured environments, where such a nominal value sometimes does not exist. Donald has pursued this approach further in [Donald86].

Contrary to Brooks quantitative approach, STRIPS [Fikes71] is a symbolic robot task planner. It operates in first-order predicate calculus and sees the world as a set of well-formed formulae (WFFs). Given a collection of WFFs representing the world and the current position of the robot, STRIPS tries to apply certain operations in order to incrementally modify the current state of the world until a goal state is achieved. These operations are represented in first-order predicate calculus as a number of facts about the world which no longer hold and another set of facts which become true after successful completion of the operation. STRIPS however, does not explicitly deal with uncertainties introduced by an imperfect world model, as it operates only on the contents of its database. Further, its world model is not very structured; it consists of a collection of facts not specially grouped. This lack of organisation hinders the development of large databases of actions and objects to work upon. As a result, these factors limit the applicability of STRIPS in real-world situations.

¹ In this case, Brooks' system would reject the plan as impossible to guarantee.

After STRIPS came a number of learning systems; Hayes-Roth describes the foundations for his TL (The Learner) system in [Hayes-Roth83]. His work does not deal extensively with error recovery but, as with most automatic learning systems, the formalisation of knowledge and the organisation of plans can be of great value. Hayes-Roth models plans and actions into theories, in such a way that actions, subjected to conditions, have predicted effects. The failure to realise a predicted effect indicates a flaw in the theory. TL then learns by adjusting its beliefs in accordance with the observed results of actions. This adjustment can be made in one of several ways, all of which deal with restriction or enlargement of range and domain of application of actions. That is fundamentally different from the Srinivas approach described next, where errors are caused mainly by incorrect execution of actions, as opposed to an incorrect plan. Furthermore, and this point is common to many learning systems, the universe in which TL operates has to be accurately observable; The Learner cannot cope with incomplete knowledge.

1.2.2 Failure Reason Analysis

Srinivas presented some of the earliest work on robot error recovery [Srinivas77]; his idea was essentially to let errors occur and then to identify them through failure reason analysis, that is, by "understanding why action A_i resulted in state S_f , a robot can determine where the problem lies and what can be done about it." [Srinivas78] His major contribution was a logical classification of failure types into operational errors, information errors, precondition errors, and constraint errors.

Once an error is detected, a failure tree is constructed as a linked set of failure nodes and action nodes. The tree is successively pruned until a few — or a unique — explanations are found; a chain of reasons represented by a path from the root node of the failure tree to one of its leaf nodes. This reasoning was later put in practice in the next system, by Gini et al. [Gini85, Smith86].

Gini uses a monitor/recoverer system to catch and resolve errors at execution time. Starting with a logically correct AL robot program [Smith86] and from general forms

of post- and preconditions associated with each instruction, the original task description — the program — is expanded into an augmented program containing the original AL program and additional instructions to check sensor data in order to maintain a description of the outcome of the robot operations, the world model. The program is also augmented in another direction to create a list of possible errors and where they are likely to occur. In order to detect these errors, the expected sensor values and tolerances are also calculated. The list of errors and the expected sensor values are kept in the local knowledge base. The monitor uses the augmented program as input and issues robot and sensor instructions. When an unexpected situation is detected, that is, when a condition of the augmented program is not met, control is passed to the recoverer which uses the local knowledge base to identify the error using heuristics about where errors are susceptible to occur and then constructs an augmented program to implement the "recovery strategy" ².

This approach is promising as it organizes a number of concepts into one practical system but, apart from the fact that its recovery strategy is still unclear, a major point prevents its direct application in telerobotics; too much of the system relies on the preprocessor and, in general, on the off-line phase which rules out any computer assistance for those portions of the task where the operator directly controls the manipulator. As a matter of fact, the system is geared towards efficient real-time execution of the task, and that is the ultimate motivation behind the preprocessor, the augmented program, and the local knowledge base. In the type of tasks considered for telerobotics, on the other hand, the decision time is usually longer than the execution time, lessening the need for rapid execution.

The work presented here expands on the ideas of Srinivas to eliminate a priori planning and analysis of the task. This will allow the use of the system along with operator interventions. In Srinivas approach, a task is decomposed in a series of states S_0 to S_n that result from the execution of actions A_0 to A_n . If a failure is detected at state S_f ,

² Though some specific examples are given, a general recovery strategy has yet to come!

by recognizing that it is different from the expected state S_i , it means that *some* prior action A_j ($j < i$) failed to achieve its goal. Srinivas, then, does not really ask "What can be done to go from S_f to S_n ?" but actually: "Why did A_j result in the failure state S_f ?" Four possible causes are then identified: operational errors, precondition errors, information errors, and constraint errors. The process of finding which failure is originally responsible for causing a transition to an error state and how it occurred is termed *failure reason analysis*. In Srinivas' system, it is a process of reduction of the set of all possible explanations to the specific one that applies to the current situation

This analysis is based on a failure tree, which explicitly represents all the states of the task from the failure state to the goal state as failure nodes of the tree and the robot actions as the action nodes linking the failure nodes. This is necessary because all action results and intermediate states cannot be validated at run time, only those conditions easily checked from information directly available from feedback and sensors are actually verified. Consequently, an action failure can manifest itself only later during the task execution and thus the need for backtracking of the sequence of events. In the failure tree, one action node points to (possibly) many failure nodes through "Possible Reason For Failure" links. Each failure node then points to another, unique, action node through links such as "Never Achieved By" or "Incorrectly Provided By". Thus, although the sequence of actions in the task is linear (there is a single agent for change, the robot), the shape of the tree is not. This is because the shape of the failure tree is a better representation of the relation between the actions than their sequence in time.

There are four types of failure nodes, corresponding to the four types of error, plus the trivial goal failure which indicates an unreachable goal. An operational error is the result of inherent problems in executing the action; servo deviations and dropping a carried object are examples of such errors. Information errors are typically caused by sensor inaccuracies or errors in the initial world model. Precondition errors group all non-verified assumptions, either trivially simple, such as the position of an object left by the robot on a stable surface, or more expensive to verify, such as the physical relationships between two

objects. Constraint errors, finally, represent those preconditions that the robot has no way of making true, such as the mobility of a fixed object.

The failure tree grows by adding failure nodes from the current action node and then new action nodes to those failure nodes, and so on. The tree is pruned in several ways: First, looking at the execution trace, precondition and constraint errors can be verified and eliminated. Second, the manifestation of the failure can rule out some reasons for failure and last, for a given action, if none of its "ancestor" actions can be shown to have failed then that action can be considered successful and the tree can be pruned at that point. Once the cause of the error has been established, local knowledge is used to correct it. In practice, this means that the sequence of steps necessary to recover from an action will be in the knowledge base for each type of action. These recovery routines will have a general form such as: undo certain steps, correct action, redo the original steps.

As the author points out, this scheme is incapable of considering interactions in the solution of a problem. Furthermore, it cannot cope with incorrect plans; the system does not reason about the plan (or the recovery procedures) itself. Also, at the time Srinivas conceived his scheme, many tools currently common were not practical. Two important ones are faster computers and knowledge based development systems. The former allows the creation and maintenance of the failure tree at run time which, apart from speeding up the recovery time, permits early error detection by verifying more conditions as the task is executed. The latter allows a better, more efficient and deeper, representation of data and control flow, in the hope that it will provide robust recovery capabilities.

1.2.3 Object Oriented Programming

The universe in which the system described here operates is divided into objects, with a relatively small number of relations between different objects. These objects are then organised into frames, with the frame slots describing the objects by assigning discrete values to their characteristics. At any point in time, the state of the world is taken to be the

set of the current values of the slots in the object frames. Some of these notions appeared early on in predicate form with STRIPS and recently in [Kak86] and [Genesereth87]. The use of frames eases the management of large numbers of objects, specially with high-level tools such as ART and KEE [Inference87, Ramamoorthy87] which have dedicated LISP constructs to create and maintain such frames and relations between them.

Actions, in turn, are also represented as frames, with slots describing the changes in the world as modifications to be made to object slots. A plan, a robot program can thus be described incrementally as a sequence of action specifications. Other means to describe operations on objects also exist. See for example [Cardelli85] for a formal description of object-oriented programming and data abstraction. The basic idea, concretised by *methods*, is to think of objects as having several facets, one for each type of action that can be applied to it. For any given action, the effect of that action on an object is defined in the object itself. Ginsberg and Smith, in [Ginsberg88] in particular, have also dealt with some problems associated with the description of actions and reasoning about change.

1.3 Motivation

1.3.1 Applications

Srinivas originally developed his system, MEND, to support off-line planning by monitoring task execution by the JPL robot during unsupervised space manipulations. In general, robot error recovery is necessary whenever unforeseen conditions arise and the cost of replanning or repairing errors is high and for all environments that are difficult to characterise. Sheridan [Sheridan86] demonstrated the usefulness of telerobotics in a class of space applications where transmission delays limit the possibility of teleoperation of manipulators from the earth control station, he stressed the fact that local processing, at both ends, can reduce the time required for task completion. Accordingly, operator error prevention and failure analysis are believed to be able to reduce task cost in two ways, namely: shortening completion time and reducing complexity and number of errors

As a matter of fact, several ideas in this thesis are directed towards an implementation with a telerobot system to perform maintenance work on electrical power lines. The problems faced are the desire to keep the operator at a safe distance from the work area, the necessity to complete the task in a short time, and the limited robotics experience of the operator. To a large extent, these constraints are shared by space [NASA85], nuclear [Thunborg86], and underwater [Yoerger87] applications. There is a growing concern for the safety of workers during *energised* power line maintenance, for both overhead and underground systems, and changes are expected in the various work laws to reflect this.

1.3.2 Replanning versus Error Recovery

In presence of error, there are two radically different approaches, the first one is more traditional and involves evaluation of the current state of the world, of the goal state, and planning to achieve the goal state. Planning, in other words, from the error state as if it was a "normal" initial state, to achieve an unchanged goal state — with the added advantages of a better knowledge of the world and, often, a shorter way to the goal state than from the initial state. The latter advantage is specially important to planners relying on difference reduction, also called means-end analysis in [Ernst69], to evaluate the path to the goal state, such difference functions are difficult to implement and are generally most reliable when the difference is small i.e., close to the goal state. The first factor in favor of replanning, a more accurate world model, comes from the manifestation of the error and its subsequent analysis. A more accurate view of the world may uncover short paths to the goal state and, given that a good difference function is available, may yield better recovery strategies since evaluation of plan success is often influenced by the actions composing the plan; the predicates are generally "tuned" to evaluate the given actions.

The second approach to error processing is error recovery, by which a robot plan is *salvaged* by taking appropriate actions to bring the world back into a state from which the original plan can resume. This is fundamentally different from replanning; in fact, replanning can be seen as updating the world model from the real world and restarting

from there, whereas error recovery is more like attempting to resynchronise the actual world with the intended world model and executing the original plan. Error recovery can use *local* replanning as well; attempting to reach the very next state (instead of the goal state) immediately. This obviously favors the use of difference functions since the differences are likely to be small in number and complexity. Figure 1.1 illustrates the differences between replanning, which goes from the error state S_{error} directly to the goal, and error recovery, which goes back to the last state S_n before an error occurred.

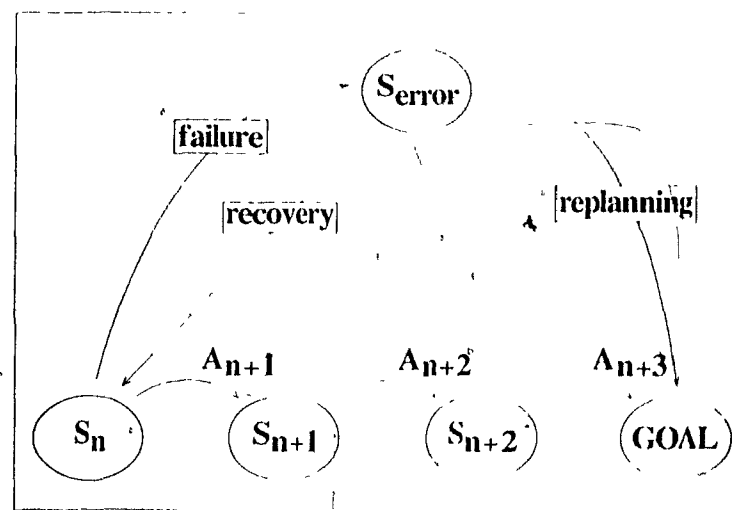


Figure 1.1 Replanning versus Error Recovery

The replanning cost is mainly computational; it requires extensive analysis of sensor values and, depending on the planning "distance" from the error state to the goal state, a great deal of search. For certain applications, a third phase, plan verification, might be required for safety or cost reasons. Error recovery, on the other hand, does not require very deep analysis, since the difference between the current and the desired states is generally small, provided the errors are caught early or that actions are independent. The analysis can thus be narrowed and directed towards a small number of relevant world characteristics. Indeed, error analysis is bound to yield a sub-optimal plan and hence the burden is shifted towards the manipulations, to put the system in a state which is part of the original plan. Furthermore, such an approach is likely to produce a number of redundant or unnecessary steps. Last, trying to retain as much of the original plan as possible can

lead to a dead end, if the plan was wrong in the first place.

From these arguments, it can be argued that replanning is inadequate in complex environments or whenever a quick response is expected, unless good and fast planners are available. In a complex environment, the frame problem is a serious issue; figuring what should be done next is not a simple search problem because the state space is very large. Associated with this, difference functions operating well on widely separated states are still needed. Error recovery and plan salvaging is potentially more efficient in complex environments, provided a reliable overall plan can be constructed. Local analysis and recovery can be used to adjust the plan for minor variations at execution time, retaining the advantages of a carefully designed (human generated) plan. However, since the analysis is only local, error recovery does not necessarily converge to the goal state, if two states are mutually exclusive for instance, and it can be deceptively inefficient. Actually, for a given problem, replanning is increasingly efficient around the goal state, whereas trying to salvage the original plan, if it is correct, is more efficient at the beginning of the task.

1.3.3 What Is ISER?

ISER stands for *Intelligent³ System for Error Recovery*. Its first purpose is to monitor and examine commands issued to the robot system by a program or an operator, at execution time, and validate them on the basis of their possibility of success given the current state of the work cell. Second, ISER attempts to analyse failures that do occur, to provide an explanation for the failure and suggest actions that specifically fix the failure and bring the work cell in a state from which the original plan can proceed. It is not meant to be a planner but it uses a great deal of heuristic knowledge to decide on a course of action without taking into consideration the number of facts a planner normally does. The "knowledge-based" nature of ISER resides in its knowledge of general actions and objects, what is required to use them, and what to do when they are a cause of failure.

³ All modesty apart!

1.4 Thesis Overview

The problem is first described by presenting its context and identifying a typical task for demonstration purposes. At that point, the representation of actions and objects used by ISER is introduced. Chapter 3 describes the world model necessary to keep track of evolution in the work cell. The next chapter classifies errors and their sources and explains the process of error analysis, while chapter 5 details the error recovery algorithm and discusses the notion of the cost of a task with respect to parameters such as the number of steps in the program and the probability of failure. Finally, the conclusion presents the contributions of this work and indicates directions for future research.

Chapter 2

Problem Description and Representation

2.1 Task Description

Following a rapid growth of their power network, many electric power utilities now put the priority on safe and efficient maintenance techniques; due to adverse atmospheric conditions, maintenance work can be tedious or even dangerous. Automation is envisaged to reduce down time, which is extremely expensive, prevent accidents, and increase consistency across the network. The most common operations are the replacement of the cross-arm, the insulator, splicing and connections, etc. [RSI85, Henkener85], all under tensions ranging from 12.5 to 200 kilovolts, with a technician operating the manipulator from a special cabin on the ground [RSI88, Cohen87] or at the end of a boom [IERE87]. At the *Institut de recherche d'Hydro-Québec*, insulator replacement has been selected as a representative task to explore the possibilities of telerobotics in overhead live-line maintenance [Girard88]. This insulator supports the 25 kilovolt conductor and insulates it from the pole. Briefly, this requires the removal of the two tie-wires fastening the conductor to the insulator, lifting the conductor, changing the porcelain, and putting back the conductor, attaching it with new tie-wires

The technique developed involves the use of a few dedicated tools, manipulated by a PUMA 760 industrial robot [Unimation83]. Eventually, a specially designed manipulator will replace the PUMA, since the latter does not provide the strength, morphology,

and insulation characteristics required by live-line work [RS185]. The tools comprise an unwrapping tool to remove the tie wires from the conductor, a wrapping tool to do the opposite, a gripper, a cutter, a range sensor, and a camera. Additionally, an auxiliary arm is available to hold and lift the conductor above the insulator. An experimental setup is used in the laboratory to implement and test robot programs.

2.1.1 The Operator

Ultimately, the system will be operated by a maintenance technician, on the ground, hence the necessity to provide automated routines to speed up the execution of the task and, since linemen are not generally skilled in robotics, the desire to provide help and supervision. Accordingly, the domain of knowledge in the expert system is not in maintenance of live power lines but rather in the operation of complex telerobot systems. The system is expected to prevent "manipulation" errors by the operator, provide analysis of failures, and automatically recover from a certain class of failures.

2.2 Typical Task

In order to illustrate and validate the expert system, a part of the actual task has been chosen: the unwrapping of one tie-wire fastening the conductor to the insulator.

2.2.1 Actions and Associated Errors

The following is a listing of the sequence of actions required to remove one set of tie-wires (there are two), along with associated errors and types classified as one of operational (O), information (I), precondition (P), or constraint (C).

3. Unwrapping of the Right-side Tie-wires

3.1 Localisation and count of the number of turns

3.1.1 Scan conductor surface with proximity sensor

- Robot errors (O)
- Inadequate environment for sensor (P)
- Position error on insulator (I)
- Not enough room for sensor/robot (C)
- Conductor out of reach (P)

3.1.2 Compute number of turns and conductor position

3.2 Unwrapping per se

3.2.1 Mount unwrapping tool

- Robot errors (O)
- Tool mounting errors (O)

3.2.2 Install tool on conductor

- Robot errors (O)
- Position/orientation error on conductor (I)
- Not enough room (C)
- Conductor movement (P)
- Incompatibility between conductor and tool (C)

3.2.3 Turn motor on and slide tool towards insulator

- Robot errors (O)
- Tool errors (O)
- Too much tie-wire to fit in tool (O)
- Tool sticking to conductor (O)
- Position/orientation error on conductor (I)
- Position/number of turns error (I)
- Tie wire in bad shape, cut, etc. (P)
- Not enough room. (C)

3.2.4 Stop motor and verify operation

3.3 Tool removal.

3.3.1 Back off tool and free the tie wires

- Robot errors (O)
- Tool errors (O)
- Tool blocked by the tie-wires (O)
- Wires not entirely unwrapped (P)
- Tool sticking to conductor (O)

3.3.2 Align tool opening

- Tool errors (O)
- Indication error on tool open (I)
- Tool not free from wires (P)
- Section of wire still attached to tool (P)

3.3.3 Unmount tool

- Robot errors (O)
- Tool errors (O)

- Tool opening not aligned with conductor (P)
- Tool not free from wires (P)
- Tool unmounting errors (O)

This list reveals that a number of errors are common to several different actions while only a small portion of the errors are specific to certain actions. This lead to the classification of errors in classes to allow inheritance of error types in the definition of actions.

2.2.2 Level of Representation

In symbolic computing in general, choosing a proper representation for the items being manipulated is essential to the efficiency of the system. In ISER, which has to reason about real objects and actions, this choice will have important repercussions on the possibilities of the system. Objects and actions have to be represented with sufficient detail to facilitate reasoning about errors while being general enough to be applicable to several similar items.

All items, objects or actions, are represented as members of one or more classes, each item distinguished by its own characteristics and those inherited from the more general class above it [Stefik85]. Those classes are meant to be as general as possible; to be shared among different applications. Each actual item is an *instance* of an item class, so that no two classes are equal but two instances of the same class are similar to a certain degree. Formally, what differentiates a item class from an instance of that class is the absence of variables in the representation of an instance, whereas certain values in a class can be undetermined. Thus, two different objects in the same class — or one at different points in time — could be differentiated only by their position. Consequently, the world can be represented as the set of all objects in the database, and their current characteristics represent the current state of the world. This representation is static in ISER, that is no movement or continuous action can be represented and hence there is no notion of, for example, a falling object or a moving conveyor belt. Instead, continuous actions would have to be modeled as sequences of discrete events.

Just how much of a change in the world will get represented as a single action (in a way, the quantization level) is directed by the fact that the goal of the system is to discover errors rather than to perfectly represent the state of the world. Hence an "action" should be a sequence of events long enough to be susceptible to produce an error but short enough to limit the number of possible errors to a manageable number. Manageable in the sense that the error can be identified in terms of a fact in the database. Actually, the level of representation of actions is linked to the level of representation of objects as actions modify characteristics of objects and, accordingly, actions are defined in terms of those characteristics. Since actions modify the state of the world, the world model must be updated as actions are executed. This is done by examining the definition of the action in the database; this definition comprises a set of characteristics to be added to or deleted from the database so that the world model is incrementally modified as the robot executes its task. This set of modifications to the database is considered complete and correct for the purposes of error recovery.

Ultimately, then, one must decide on what kind of characteristics must be included and available to define objects and actions on those objects. Indeed, the choice depends heavily on the application, speed required, and processing power available. In this case, and for telerobotics in general, the application demands good interactive response and a high-level interface, but it also benefits from great processing power (a human) and does not require a great deal of autonomy (although it is always an advantage). As the main goal is to assist the user to issue commands to a complex system and to understand decisions, reports, and requests from the system, the representation employed must reflect this and give the operator a qualitative view of the work cell rather than manipulator-oriented dimensions and coordinates.

2.3 Frame Representation

2.3.1 Objects

To reason about robot actions and errors, it is necessary to have a representation of the world; the objects worked upon and the robot itself. This world model is organised in frames, one per object. This type of representation is particularly appropriate since ISER deals with qualitative aspects of the world and does not use confidence values; the elements of the frames (the slots) are then simple parameter/value pairs. In the case where a parameter can take more than one value at a time, many slots can be assigned to the same parameter, with differing values. This is true of several relations between objects, such as *attached-to* which can be used to indicate that an object is linked to many other objects at the same time.

As can be seen in figure 2.1, the object class comprises the *simple-tool* sub-class, indicated by the *kinds* relation. This type of relation between frames allows automatic inheritance (maintained by the ART shell) for specific classes from their parent classes. The *simple-tool* class, for example, comprises the *tool* and *sensor* sub-classes. These distinctions are used in various sections of the system to restrict a search for objects which have certain abilities, to simplify parsing of robot messages, etc.

Simple tools, tools, and sensors are related to generic objects through their *applied-to*, *modifies*, and *measures* slots. Simple tools by themselves cannot do much, they usually serve as an *interface* between tools or sensors and generic objects; a screwdriver is an instance of a simple tool, a robot (an instance of a tool) applied to a screwdriver, which in turn is applied to a screw, can *attach* an object to another and thus modify their respective *attached-to* slots. This is used at the error detection stage to verify the proper usage of tools and sensors, it could also be used for automatic plan generation to determine which sensor is suited to obtain a certain parameter and which tool can perform the required task.

As pointed out before, relations between objects are modeled as slots in each object's frame. While each object is independent, that is no slot is shared with another

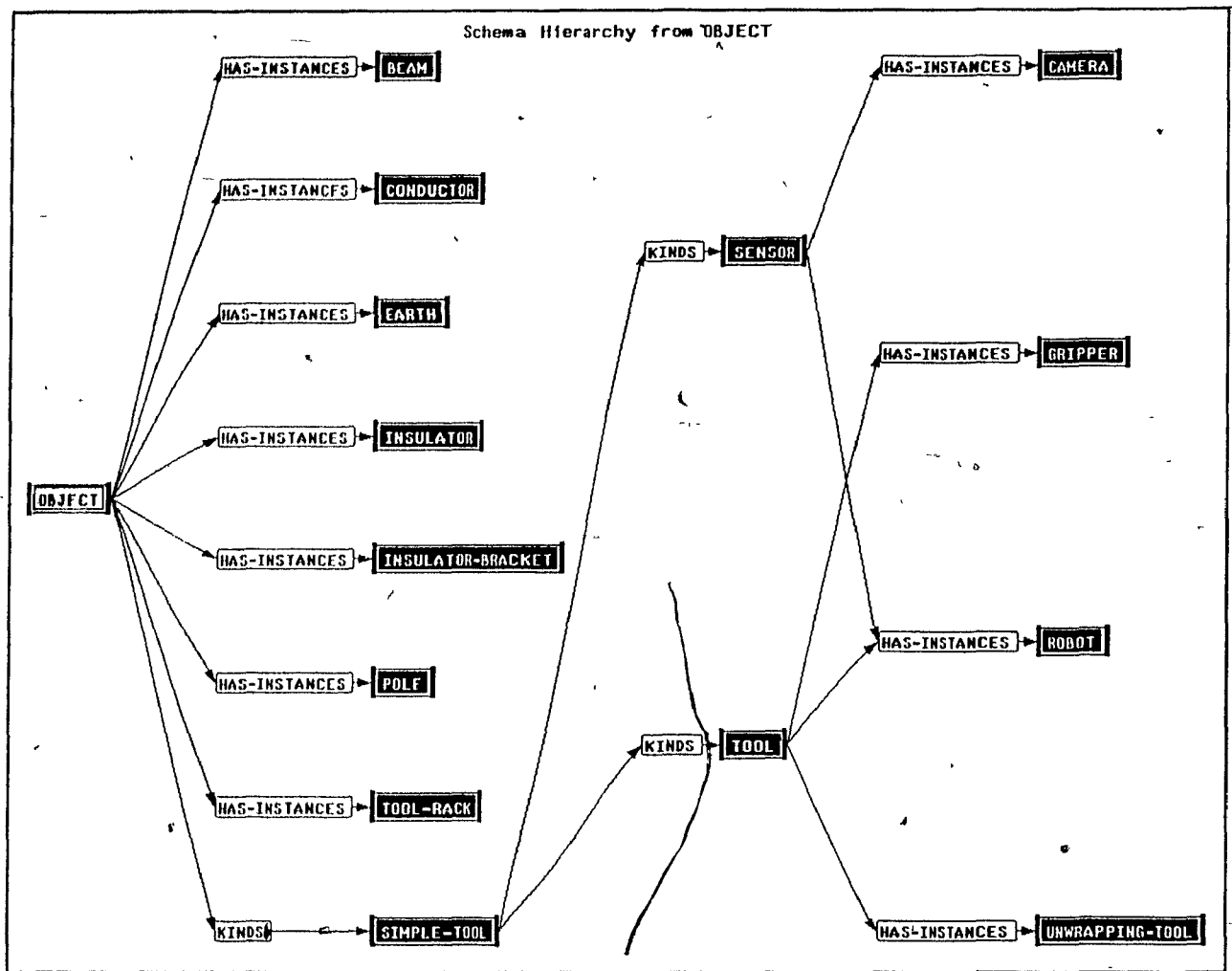


Figure 2.1 Objects classes in ISER

object's slot, the value of a slot often depends on a value from another related object. The mobility of an object, for instance, is recursively defined to be the same as the mobility of the objects to which an object is attached. This definition, including minor variants, is reflected in four rules in the section of the system which keeps the world model up to date

Although ISER can be customised, it is desirable to have a general system capable of satisfying most applications; ideally, only task-specific knowledge should need to be added. Much in the same way that a robot language can build subroutines from primitive actions, a robot task description system must allow the creation of specific objects and actions suited to the task at hand. To prevent chaos and ensure proper integration of the various components of ISER, a formal, strict representation of the objects in the world

is defined here. Surely, new slots can be added to augment the functionality of ISER; in particular to take advantage of ART's data structures in an object-oriented paradigm.

Currently, an object is defined to have at least five *slotnames*:

```
(defschema object "any physical object"
  (dimensions)           ;a three-valued vector in centimeters
  (attached-to)          ;an object
  (abstract-position initial-position)
  (weight)               ;in kilos, approximate value only
  (mobile)               ;yes or no
)
```

This particular choice of slots is directed by the fact that ISER operates at a symbolic level only. Those slots that can contain numbers (dimensions and weight) could actually take qualitative values such as small or heavy since their sole purpose is to verify the compatibility of objects with certain tools; there is a maximum payload that the robot can support and, similarly, there is a maximum object size that the gripper can handle

The abstract-position slot indicates the current symbolic position of the object. As an example, consider the abstract-position of the unwrapping tool; its value when the task begins is tool-rack, this changes to robot while the manipulator is displacing it, and changes again to conductor when the tool is applied to remove the tie wires. Therefore this symbolic position is an object and it can eventually be used with a real-world interpreter that is capable of mapping symbolic description to/from physical coordinates. At that time, a physical-position slot could be added to the standard object schema⁴. In a few cases, the abstract-position can be the object itself, to indicate that this object's position is absolute rather than relative to another object. This is always the case for fixed objects, such as the tool rack, which also have a value of infinite for their weight slot, reflecting the impossibility (for the robot) to move such objects. Besides its physical signification, the infinite weight and self abstract-position

⁴ The literature seems to use the terms *frame* and *schema* interchangeably. Since ART uses *schemata*, we will reserve the use of *frame* for a general item and *schema* for an ART construct

slots serve to stop recursion when evaluating the position of an object through other objects.

The last standard slot, `attached-to`, is multiple-valued, meaning that an object can be attached to many objects at the same time, although it can have only one abstract-position. This raises the problem of determining a unique abstract-position from multiple `attached-to` values. Since ISER has to present a consistent world model to the operator, the ambiguity is resolved by assigning to abstract-position the position of the object most recently attached to the current object. Considering that the `attached-to` slot is an attempt to describe the structure of the objects, this solution generally shows how far the assembly has gone since one usually assembles small parts into more complex objects, at a higher level of abstraction. When a structure is disassembled, the abstract-position slot is set to the largest object in the structure.

2.3.2 Tools

Just as there is a notion of a standard, generic object, the `simple-tool` class has this minimum configuration:

```
(defschema simple-tool "any tool, passive or active"
  (is-a object)
  (applied-to)           ;an object
  (purpose)              ;an action
  (configuration)        ;open, closed, extended, etc.
)
```

The first slot simply states that a `simple-tool` inherits all the characteristics of general objects. The `applied-to` attribute is dynamic and indicates, when the tool is used, which object is being worked on to allow world model update of that object. This attribute is multiple-valued since a tool can be applied to several objects at the same time, to attach them for instance. The `purpose` slot associates an action with the `simple-tool`, the action definition can be looked up to determine the effects of the tool of the conditions necessary prior to its use. The last slotname, the `configuration`, is applicable to most tools; some,

such as screwdrivers, have only one configuration but many, such as grippers (open/close) and cameras (short/long focal length) have multiple states that need to be represented and hence the configuration attribute groups these states that would otherwise need to be represented as a large number of different slots, breaking the generality of the system.

The simple-tool class is divided further into the tool and sensor classes. At this point, the distinction between simple-tools and tools is clarified: a simple-tool cannot accomplish anything by itself: a screwdriver, for instance, needs to be turned one way or another by a tool capable of exerting some force. A gripper, on the other hand, is considered to be a tool since it can pick an object by itself, even though another tool may be required to move the object. In a way, simple-tools can be considered as adaptors between a generic tool and a particular object. For a powered screwdriver, the various sizes and types of bits are considered to be simple-tools and the motor/gear train assembly is the tool. This distinction justifies the existence of the powered slot, which takes a boolean value. This attribute is so important in error handling (in a panic situation, it may be crucial to find any moving part and to stop all motors) and since it is common to all tools, it should not be lumped into the configuration category

```
(defschema tool 'powered, active tool'
  (is-a simple-tool)
  (modifies) ;an object attribute(s)
  (powered) ;boolean, subset of configuration?
)
```

This actually raises the issue of why there should exist simple-tools when all tools could be in the same class, perhaps differentiated through the configuration slot. Recalling the automatic screwdriver example, it should be noted that the various bits are *independent* of the motor: they may get lost, break and they have to be inserted in the tool, etc. The point is they are separate objects and the system may have to reason about them, to replace a broken bit for instance. Furthermore they cannot be disposed of as normal "task" objects; they cannot be assembled into a finished product yet they are used repeatedly. Hence simple-tools cannot be incorporated into the more general class of objects, nor can they be raised to the status of tools since, in general, the robot cannot

apply them directly and they are associated with one and only one tool

Finally, what distinguishes tools from sensors is the fact that the former cannot provide any information on the objects worked on, only sensors can. This is reflected in the `modifies` and `measures` slots for tools and sensors, respectively. These slots point to object attributes to be used for world model update when a tool is applied to an object and in the error recovery phase, to identify which sensor can verify a fact or provide the information necessary for an action to be executed.

```
(defschema sensor 'a measuring tool'
  (is-a simple-tool)
  (measures) ;an object attribute(s)
)
```

2.3.3 Actions

Although not as structured as objects, actions are described with frames. Their hierarchy is not as deep as that of objects; namely, actions do not inherit as many characteristics from other actions as the object classes do (figure 2.2). Furthermore, actions frames are fixed and do not change over time. They are used to define the changes made to the world as execution proceeds, in the form of modification specifications for object slots. For instance, the robot action (`move block-23 position-17`) has at least two effects in the world: the `position` slot of `block-23` must be updated to reflect its displacement, and the `configuration` slot of the robot must also reflect the new arrangement of the robot. This is achieved by means of the `d-modify`, `d-assert`, and `d-retract`⁵ slots in the action frame which states which object slot should be modified and how.

Action frames are also used for error detection and recovery; they define pre- and post-conditions that must hold during correct execution of a task. These conditions are facts that must be present in the current state of the world, otherwise an error is raised. Chapter 4 expands further on the notions of error and error detection.

⁵ The *d* stands for *defined* as opposed to a particular instance of a modification to be made

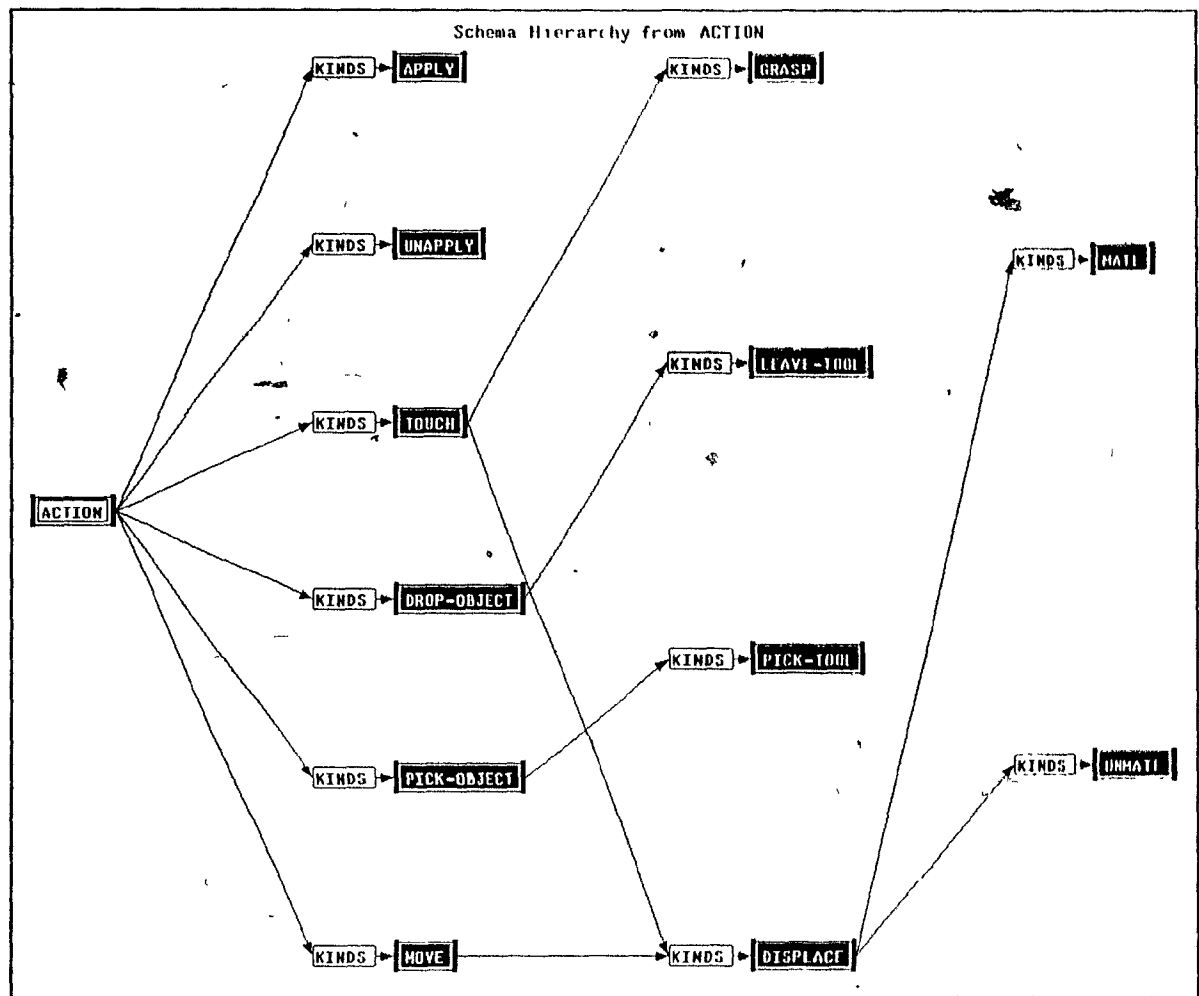


Figure 2.2 Action classes in ISER

In ISER, the generic action has these following elements

```

(defschema action "any action, by any tool"
  (d-effector) ;the actor, the tool performing the action
  (d-error) ;possible error associated with an action or a tool
  (d-assert) ;fact to be asserted after execution of an action
  (d-retract) ;fact to be retracted after execution of an action
  (d-modify) ;fact to be modified after execution of an action
)
  
```

The d-effector slot has the value, if it exists and is unique, of the agent associated with the action. Its use is internal to the high-level parser (section 3.4). Formally, an action is a set of preconditions, post-conditions, new fact specifications, and old fact specifications. The preconditions are represented by the d-error slots while the post-

conditions and new/old fact specifications are equivalent to the d-assert and d-retract slots. The d-modify slot can be modeled as a combination of d-assert and d-retract slots. The fact that post-conditions and modification specifications are represented by the same slots follows from the method by which the world model is updated; it could, for instance, have been based entirely on interpretation of sensor readings. Post-conditions are generally less specific than world update indicators and hence, in this system, post-conditions are derived from such indicators. This is in contrast with what has been done in [Gini85], where special steps are included in the robot program to provide information to monitor the outcome of actions.

The following notation will be used throughout the paper to represent actions:

A : action, a set of C, P, O, N

C : precondition

O : old fact to be retracted

N : new fact to be added

P : post-condition, any O or N

F : any fact, $F \in W$

S : a possible state of the world, $S \subset W$

W : the set of all facts, in any state of the world

and:

$A: C_1, C_2, \dots, C_n, \dots, C_i \quad C_n \in W,$

$O_1, O_2, \dots, O_n, \dots, O_j \quad O_n \in W,$

$N_1, N_2, \dots, N_n, \dots, N_k \quad N_n \in W$

Obviously, the modifications must not be mutually exclusive

$$\forall n \quad \forall m \quad \forall p: (N_n \in S) \wedge (N_m \in S) \wedge (O_p \neq N_n)$$

2.3.4 Where is the Knowledge?

Precisely, two kinds of information are required in ISER: First is the ability to follow the task and represent changes in the work cell as variations in the database and provide a world model to assist the user in determining the system's idea of the world and behavior. This also serves ISER's error recovery by complementing the second type of information required, the pre- and post-conditions to every action

Information on the actual state of the world is provided primarily by the manipulator system which relays the position of the robot and the values of the sensors. This is matched with corresponding slots in object frames, which are then updated accordingly. In many situations, however, there is no information on the new state of certain objects and it must be inferred from the expected outcome of actions. This expected outcome is the second source of knowledge. The pre- and post-conditions for all actions are given explicitly at compile time in the definitions of actions and objects, either directly or through inheritance from other items.

2.3.5 ART, The Automated Reasoning Tool

The discussion of representation of the world would not be complete without a brief description of the facilities provided by ART[†] that are used in ISER. First and foremost, ART provides a database system with pattern matching that allows retrieval of world model facts. Second, it provides a frame organisation of the data called the *schema* system; this organisation allows class definition and inheritance as well as object manipulation. The database can then be modified with pattern-matching rules and an inference engine that provides forward-chaining and, to a certain extent, backward-chaining of the rules as well as priorities on rules and classes of rules. Last, it offers a rich set of graphic primitives to group and display the information in an efficient way. The reader is invited to note that

[†] ART is a registered trademark of Inference Corporation

this is in no way an endorsement of a commercial product, but rather a clarification of its capabilities with regards to the large number of expert system shells available; it is used as a prototyping tool to avoid reinventing the wheel.

2.4 Domain Independence

It is relatively difficult to classify and organise all the tasks that can be accomplished in a robotic cell. The number of high level actions is infinite, yet they can be described in terms of a limited set of primitives, such as those composing the robot language itself. VAL-II, for instance, uses about 30 keywords to describe robot motions [Unimation86]. In the context of error recovery, these primitives can be associated in a relatively straightforward way with errors characteristic of the actions involved, that is every action can only fail in one of several ways. In ISER, this is the role of the d-error slots.

That part of the knowledge is stable and independent of the task or domain of application. To be really useful, however, ISER has to be augmented with a great deal of domain-specific knowledge. Objects in general and specially tools have to be described in terms of standard ISER primitives. Adding a new object requires filling in the standard slot values, adding a new sensor or tool requires a more complete description since not all of their characteristics, such as compatible-with can be derived from sensor observation. Adding a new action requires more analysis, its effects on the world model must be determined as well as its preconditions. This is unnecessary if the new action can be represented by a sequence of predefined ISER actions; in such a case, the new action should be broken down into atomic actions that ISER can handle in the advent of an error. The design of a truly flexible representation, however, is beyond the scope of this dissertation; as John McCarthy pointed out [McCarthy87], "the problem of generality in artificial intelligence is almost as unsolved as ever".

Chapter 3

A Dynamic World Model

The state of the world at any point in time is defined as the set of all object slots. In practice, a small number of slots has to be maintained directly; other slot values can be deduced easily. Positions, for instance, have to be recalculated every time an object is moved and, since positions cannot always be determined exactly, they are likely to introduce uncertainties and errors. Once the position is known, however, determining relations such as attached-to or mobile is deterministic and thus straightforward. Hence, the position of objects and tools must be determined first, along with tool configuration, like open or close for a gripper and extended or retracted for a robotic arm.

3.1 Effects of Actions

When the system is started, the database is initialised, to reflect the state of the work cell, by setting the objects slots to the appropriate values; starting with typical values, the world model is adjusted from sensor readings. After each action, the world model must be updated to reflect the changes in the actual world. Defining the program as a sequence of action frames, effects of actions on the state of the world can be monitored incrementally. This is what was done in STRIPS, which included a set of facts to be added or deleted from the set of well-formed formulae representing the state of the world. Similarly, actions in ISER have special slots whose values define facts altered by the execution of the action. These d-modify slots contain patterns to be instantiated at run-time by the

high-level parser (section 3.4). The d-assert and d-retract slots complete the function of d-modify. As mentioned earlier in section 2.3.3, the very nature of ISER makes the modification patterns (the d-modify slots) equivalent to post-conditions because the expected outcome of actions is used to construct and maintain the world model. The term post-condition is really meaningful only in the context of error recovery.

Since there is no planner available to organise and sequence the actions, it is imperative to use the proper representation for the actions. In particular, the preconditions must match the modification patterns if actions are to be chained. Conceptually, any action can follow any other action, provided the preconditions are met, and for this to be also true using ISER's representation, there are restrictions to be followed. The number of preconditions must be small and there must exist at least one action with a matching modification pattern for every current precondition. Unmatched preconditions are actually constraint errors, which the robot cannot resolve. Hence, *the database of actions is self-contained*.

Each modification pattern must be matched by a precondition in another action. Furthermore, the intent of the action must be clear and independent of preceding actions; for instance, mate and unscrew are used instead of move down and rotate counterclockwise. Actions must be represented at a level low enough that the modifications are observable. If the only possible evaluator is human, it should be expected that only the human will be able to diagnose and solve an error. Hence, the level of representation is a function of the resolution of the available sensors—sensors taken in a very broad sense, to include any program available to interpret them. In general then, one cannot have "The insulator is changed" as a post-condition since such a fact is too complex to evaluate. The modification patterns must also be achievable; the predicates used must match the actions in the database. The previous example, The insulator is changed, would not qualify since, in general, there is no such general action as change-insulator. Actions must also be simple to allow for the preconditions to be orthogonal, independent of each other. This to make sure that the order in which the preconditions are established

is irrelevant.

To summarise, the plan must be representable as a set of successive calls to robot program subroutines. All displacements should be relative to objects, to be able to infer their intent. In fact, ISER could be considered to be a front-end to a robot programming language, and a succession of action frames would constitute an ISER program. The preconditions and modification patterns must be at the same level of complexity and match. The set of preconditions and modification patterns must then be closed, the only openings allowed being the constraint errors. The conditions must in general be expressed in terms of world model objects and symbolic coordinates. They must not be expressed in terms of preceding actions, in other words the sequence of robot actions passed to ISER must constitute a *local Markov program* [Nilsson80]. The preconditions must be independent of each other so that the order in which they are made true is irrelevant.

Using the notation introduced in section 2.3.3, $C_i = P_j = P_k$, for any i, j, k , namely, all preconditions and modification patterns are world facts (in the database) and, conversely, all world facts are susceptible to be conditions.

$$\langle \text{world-fact} \rangle := \langle \text{object-slotname} \rangle \left(\langle \text{object} \rangle \langle \text{object-slotvalue} \rangle \right)$$

3.2 Use of Sensors

Sensors are used to confirm — or infirm — ISER's idea of the world. Sensor values serve as reference points from which the world model is extrapolated. This information is also used to verify and eliminate post-conditions or, as described in the next chapter, to prove that an error has occurred. The process of maintaining the world model from action definition is completely independent from the integration of measured values in the model and, consequently, sensor input is allowed to occur at any point during execution of the task. This permits, for instance, use of unexpected robot messages as information to be integrated in the world model.

3.3 Action Trace

The interface between ISER and the real world is through the robot controller. In one direction, the controller sends a copy of the executed action to ISER, as well as sensor output. In case of error, ISER sends back commands to be executed by the manipulator or the sensors. The sequence of executed actions is necessary for the purpose of error recovery, to trace back the failures to their source. This sequence is therefore kept as a list of actions, each prepended with its sequence number. Every time an action is added to the action trace list, ISER's world model is incrementally updated from the new action. These actions are not analysed by the controller but they must correspond to those described at compile time in the database. It is the responsibility of the high-level parser to analyse the robot controller's messages to identify the type of action, the effector, the subject of the action, etc. Later, the output of the parser is used by other modules in ISER to update the world model and perform error detection and analysis.

3.4 High-level Parser

The most general form of condition includes an action name, a state specification, and the condition description itself.

$$C(A, S) = F$$

where

A : pointer to an action description

S : state specification

F : fact or set of facts in the database

Such a form is overly general and in practice, one of two schemes is used: In the first one, the state specification is null, matching all states, and the condition becomes a set of facts (actually, any observable events). In the second scheme, there is no condition description (F is null) and the state specification matches only those states for which the

action is applicable. Indeed, the two representations are mathematically equivalent but, conceptually, the first representation, where S is null, is readily applicable when conditions can be represented as a small set of facts, and the second is more convenient for planners dealing with multiple states. Since there is no notion of multiple states in ISER and that only the current state is considered, S is made null and hence the conditions reduce to this general form:

$$(d-error \langle error-type \rangle (\langle action \rangle \langle condition \rangle))$$

Concretely, this means that errors, as defined as facts in the database, include an error type, such as information or operational error, a specific action for which the error is defined, and a condition that, when present in the current state of the world, raises an error of the specified type. The condition can contain certain variables to allow errors to be defined at compile time with sufficient generality to be applicable in any situation. It is the role of the high-level parser to instantiate such variables at run-time, reflecting the context in which the actions and failures take place. The high level parser is a front end to ISER for the robot controller; the controller issues statements describing the current robot actions for the purpose of maintaining a trace of the actions in ISER. The parser is responsible for converting actions of the form:

(action parameter-1 parameter-2)

into database entries specifying *explicitly* what the agent (the tool) is, along with which objects are being acted on and how. Appendix B describes this process in greater detail. Conceptually, the parser allows actions to have parameters and to instantiate these parameters at execution time. In the current implementation, a parameter can be one of the four parser-variables; the effector of the current action, a wildcard that can stand for any value (used mainly for retraction of sets of facts), and subject or argument, which are interpreted according to the current action.

Error processing in ISER is divided into three parts; error detection and analysis come first, followed by the error recovery itself, described in the next chapter. The purpose of error detection is to stop execution of the robot program before damage occurs to the work cell or the manipulator and to provide indication of the failure type and current state of the world to the second phase, the error analysis. Error analysis is essentially a computational process; from the symptoms of the failure given by the error detection and the action trace, error analysis is used to determine the original source of the error.

4.1 Error Types

There are two ways in which an error can be detected; the first one, easiest to detect, is in the form of an error message from the robot controller. This type of error is generally given in terms of low-level primitives representative of the configuration of the manipulator and capabilities of the controller. Typical errors of this type in VAL II are:

- *Stopped due to servoing error*
- *Hand closed too far*
- *Hardware defect*
- *[Fatal] Out of range* Jt <joint>
- *Motor stalled* Jt <joint>
- *High Acceleration* Jt <joint>
- *Envelope error* Jt <joint>

While VAL II defines over 400 error messages, related to hardware failures, system limitations (disk space, communications, etc.), programming or run time errors (arithmetic overflow), and, like the above list, manipulation errors, ISER actually deals only with the latter. The term *manipulation error* denotes here the class of errors caused by limitations of the controller or the particular robot configuration; in a broad sense, they are due to departures of the actual system from an ideal one. Actually, ideal manipulators do not exist and are used only as an abstraction in the blocks world; they are not subject to gravity and other physical constraints such as manipulator configuration and length. After detection of a manipulator error, it is analysed to infer as much as possible of the current state of the work cell, the position of the robot, and the nature of the error (i.e., collision, excess weight, out of range, etc.). It can be more difficult to interpret, though, *which* action failed and triggered the error message.

A second type of error occurs when a post-condition (a local goal) is not satisfied upon completion of an action, the error is raised when there is a discrepancy between the observed state of the work cell and the internal world model. Of course, this type of error is most convenient because it is expressed in terms of sub-goals and can thus be used to ensure the desired accomplishment of the task, if the plan is regarded as a succession of states, then a correct sequence of successfully achieved sub-goals guarantees plan success. Alternatively, if an evaluator function is available, it can be used to assess the success of the task by looking only at the final state to verify that it matches the goal. Again, as mentioned in the introduction, such an evaluator is impractical for real, complex, tasks because its cost is prohibitively high in resources and computational time.

The first type of error, coming from the robot system, is mapped, after being analysed, as sub-goal error; the error message is simply considered to be a manifestation of the error, much as if there was an implied post-condition stating that there should not be any robot error at any time. The error message from the robot is thus treated as additional information about the world and is added as a proven fact in the world model.

Hardware failures and programming mistakes cannot be resolved without, res

pectively, redundancy of tools or high-level reasoning to untangle the programmer's intent. Apart from fault-tolerant computing, there does not appear to exist any work to support the idea of non-fatal hardware failures. On the other hand, there is some progress done in the area of automatic software understanding and debugging; HACKER, for instance, has been developed to study learning by debugging "almost-right" plans [Sussman75]. Another example is the preprocessor used to generate the augmented program in Gini's implementation [Gini85]; the augmented program is intended to be more complete and thus better than the original program

Partly because the world is not entirely observable and partly because the world model is imperfect and incomplete, not all errors are detected. This may be due to the absence of an appropriate sensor for a particular state, because the sensor is incorrect, because it would take too long to measure a certain parameter, or even due to the sheer complexity of the current state. Thus, errors are likely to go unnoticed until a subsequent action triggers a *manifestation* of the error. The philosophy in ISER, inherited from the original Srinivas system, is to let those errors occur. This allows the robot task to proceed at high speed, performing costly analysis only in case of error, although in practice, unverified preconditions are not discarded but are rather marked as possible errors to speed up future analysis. Having mapped all types of failures as the absence of any precondition necessary for the next step to proceed, and given that undetected failures are allowed to occur, errors are defined to be preconditions *proven* to be false.

4.2 Error Sources

Failures can have many sources; this section discusses several categories of errors and what can be done to circumvent them.

Even before recovery is performed, sensors are important to verify success of an operation from within the user program, in which a manipulation can be repeated until successful, or to measure some parameter before or during a manipulation. Evidently,

sensor inaccuracy is a primary cause of error, whether the parameter is beyond the sensor operating range, the sensor is misused, or even failed completely. Sensor failures cause severe problems in ISER because sensors are believed, and not consistently verified. However, they are verified to a certain extent during error analysis and recovery when actions — even sensing actions — are verified; if another sensor exists to corroborate the value of the first one, it can be used to provide limited sensor integration [Shafer86].

Insufficient use of sensing can have the same effects as sensor failure but is not so severe because a cause for the error can be established. Lack of sensing becomes evident when certain characteristics or parameters of the work cell cannot be guaranteed; the program incorrectly believes the world is in a certain state when it is not. Brooks' system [Brooks82] is a direct attempt to eliminate such programming errors.

Tools can err in two ways, they can be inaccurate (such as a servoing error in the manipulator) or they can totally fail to achieve some function. Inaccuracy can be thought of as a quantitative error, while total failure is a qualitative error. The means to evaluate these errors can be similar but recovery procedures differ, it could be advantageous to have total failure since ISER's recovery process is geared more towards reattempting failed actions rather than correcting (replanning) the current state of the work cell. Total failure, on the other hand, may imply damages that can be difficult to repair and since inaccuracy usually results in more controlled situations, where the actual state is only *quantitatively* different from the expected one, the latter could be easier to resolve.

Due to the limited nature of the world model in ISER, external agents and lack of integrity of objects in the work cell are likely to introduce unpredicted states within the normal sequence of actions. As a matter of fact, ISER assumes there is only one agent for change, the robot, and gravity, wind, etc. are never considered. As a result, if the objects are fragile or the assembly is unstable and subject to "asynchronous" modifications, unexpected errors will occur and will incorrectly be imputed to the failure of a previous action. Although the diagnosis is incorrect, the recovery procedure will probably solve the problem and reach the necessary state by redoing the "failed" action.

Programming errors are not detected explicitly since one important assumption in ISER is that the plan is correct and errors can arise only from action failure rather than from incorrect or incomplete sequences of actions. Since it relies so much on the world model, ISER will only find errors that have a physical manifestation. It will eventually find the *effect* of a logical or programming error without reasoning about it and hence cannot resolve logical errors, unless they are at a level low enough to be modeled within an action. Formally, it cannot identify an error as the result of a *particular sequence* of actions, ISER can only discover that a certain state was not reached by a *certain (one)* action. In other words, ISER believes the robot is faulty, not the program.

The last source of error lies in ISER itself, the world model depends for the most part on the post-conditions associated to the action definitions. Such definition errors are potentially fatal because an action could never achieve some of its post-conditions and cause ISER to believe there is a permanent failure of some sort. This condition can be detected to some extent when all preconditions have been verified but the action still fails. In general, if the world model is different from the actual world, it can come from an actual action failure (a genuine error), from an incorrect piece of information given by a faulty sensor when in fact the real world is perfectly fine, or from an incorrect action definition in ISER. Since ISER does not reason about itself, it can be misled by the last case and by a faulty sensor if no other sensor is available to verify it.

4.3 Run-time Error Detection

There are numerous advantages in detecting errors as soon as they occur, since ISER resolves errors mainly by reattempting failed actions, the sooner the error is detected, the closer it is to the state propitious to the execution of the "patch" (chapter 5 discusses this in greater detail). Similarly, error analysis is simpler because the effect of the failure can be directly observed. Further, the chances of propagation of the error are diminished and the probability of damage is lowered. To this end, the post-conditions of every action should be verified entirely but it is impractical to do so. Some conditions, however, are

quickly verified and can be used to monitor the task progress. This idea originated in the Srinivas system and is central to the error analysis procedure of ISER.

One notable difference with the Srinivas theory is the way verified preconditions are handled; such facts are removed from the database and only unverified preconditions are kept as possible reasons for (future) failures. In the context of error detection, the distinction between preconditions and post-conditions becomes clear, post conditions can be used to verify action success even though not all post-conditions are necessary, preconditions are all necessary and are verified by matching them with corresponding previous post-conditions. If no post-condition is found for a current precondition, a warning is issued to the operator; this can be caused by an incomplete world model, by a programming error or, similarly, by an invalid request from the operator. If a corresponding post-condition is found but it was not verified, the precondition is also kept as true but unverified. Finally, if the corresponding post-condition exists and is verified, the precondition is eliminated because it will never be used again.

The above procedure is similar in spirit to the construction of the failure tree in the original system except that ISER makes use of the pattern matching capabilities of ART to virtually build the tree as the plan is executed; although they are not explicitly linked, ART can easily match the preconditions of a given action with the post conditions of previous actions.

4.4 Post-mortem Analysis

The tree can then be constructed, explored, and pruned according to the following principles:

- Look for distinctive features of a failure to verify if the failure is possible
- If no reason for the failure of an action can be found, then it can be assumed that the action succeeded and thus there is no support for a failure due to that action (i.e., a successful action cannot cause another action to fail)

This process is, of course, applied recursively until, hopefully, only one possible source of error subsists. If more than one possible source of error is found, the system asks the operator to resolve the ambiguity

4.4.1 The Failure Tree

Once an error is detected, the crucial step is to determine the cause of the error, this process was called failure reason analysis by Srinivas [Srinivas78]. Although the error analysis in ISER does not yield an explicit failure tree as the original technique did, it is easy to match the preconditions of a given action with the post-conditions of previous actions. The process of finding the cause of the error is reduced to a search problem, starting at the unsuccessful action, for a previous action whose failure to establish the necessary preconditions for the current action was not detected. Before this search takes place, during the actual execution of the task, default reasoning is used to a great extent to prune the search, all actions are assumed to have succeeded unless proven otherwise. This allows on-line, real-time reduction of the search space: if an action is proven successful, its post-conditions will not have to be verified further and hence can be eliminated from the database.

The tree is expanded towards the failure reason by matching unresolved preconditions with earlier post-conditions. This constitutes a chain of actions during which an error might have occurred. The actual search is recursive: if an action fails when its local goal (a post-condition) is not met, this is because some precondition(s) of the current action was not realised by a previous action which also failed. The termination condition for the recursion is when no cause for the failure of the current action can be found, in that case the failure is attributed to either the action itself, to an unverified information, or to a constraint error.

The tree is pruned if there is no evidence to support the possibility of failure and is pruned further using the method of failure "signature" (what Srinivas calls distinctive

features of a failure) to eliminate possible reasons of failure based on traces left by failures. Collisions, for instance, leave a *Motor stalled* fact in the database, whereas large position errors do not.

4.4.2 Ultimate Source of Error

ISER considers only one error at a time or, more precisely, believes that only one action failed originally and caused subsequent actions to fail. Although in practice this may be false, all failures can be treated under this assumption if they are processed in sequence. A few criteria distinguish the first unsuccessful action from all others; if it failed first, all preceding actions established their post-conditions, by definition. Accordingly, the failed action had all its preconditions established, since all previous post-conditions are valid. The ultimate source of error is thus the first action which failed to establish its post-conditions. The absence of one of these later caused another action to fail and so on until a manifestation of the error was detected. The first error is important because it is futile to attempt to continue execution of the original plan, knowing that a condition is missing, and therefore it must be resolved first.

4.5 Complexity Analysis

It is typical of many search and planning problems to suffer from combinatorial explosion, the number of possible outcomes grows exponentially with the number of steps explored. In this section, an attempt is made to quantify the depth and breadth of the search for the ultimate source of error.

First of all, the complexity of the search depends heavily on the average number of preconditions per action, or actually, on the average number of unverified preconditions. Each unverified precondition must be matched against post-conditions of preceding actions which, in turn, must also be verified recursively. The upper limit on the number of actions

that must be traversed this way is the number of executed actions and the lower limit is, of course, the number of preconditions of the failed action is the error is caught immediately.

Presently, the number of preconditions in the definitions of ISER actions is artificially low, from 3 to 6, as this is all that was needed for computer simulation, but a more realistic number would be more around 10 to 15, depending on the level of representation and generality of the system. Of that number, only about 10% to 20% can be expected to be verified and eliminated. The search can be limited in other ways, though, by segmenting the task into "unrelated" parts; if it known that a section of the program was completed successfully, the unverified post-conditions can be marked as verified and their dependent preconditions can be eliminated. It is then reasonable to imagine that such unrelated parts be limited to less than a hundred steps, which is a reasonable upper limit, given that not all these conditions will have to be independently verified with sensors.

Last, the number of conditions to be verified depends heavily on the "serial" nature of the task, if a large number of actions must be executed in a given order (building a tower, for instance), the search will be limited in depth to the first action proven successful. This follows from the principle that the success of an action guarantees the success of its predecessors. On the other hand, if many actions are independent and can be executed in any order, the task is highly parallel and the branching factor of the failure tree is higher. This generally leads to larger search spaces.

5.1 Error Recovery Algorithm

Upon completion of the failure reason analysis, the first action that failed will be identified as the ultimate source of error. The principle behind ISER's error recovery is that the success of the plan does not depend on the order in which the preconditions are established. It then follows that, if a precondition is missing, the precondition can be established at any moment before the failed action can be attempted again. The general idea is to establish the conditions necessary for the *controlled* execution of the failed action i.e., with additional feedback. This controlled execution is considered a patch for the original plan. Since errors are not necessarily detected as they occur, the state of the world at the time the error is detected can be very different from the state in which the "patch" can be applied. Instead of direct replanning from the current state to the "recovery" state, the manipulations that took place after the failed action are analysed to guide planning and gradually bring the state of the world towards the recovery state by undoing actions, in reverse order. This procedure is outlined below:

```
patch = failed action  
top = last executed action
```

```
;;; Undo actions that have post-conditions similar to preconditions  
;;; of the failed action. Such post-conditions have the same parameters  
;;; but different values compared to the preconditions.
```

```

for (i from top to patch)
    if (post-condition (i) = precondition (patch))
        undo_with_sensing (i)
        remember_action_was_undone (i)
    endif
endfor

;;; If there are preconditions still unresolved, search for any
;;; known action that is susceptible to achieve it.

forall (precondition-error (patch))
    find_action (i)
    such that (post-condition (i) = precondition-error (patch))
    execute_with_sensing (i)
endforall

;;; At this point, if there are preconditions missing, preventing
;;; application of the patch, abandon ship.

if (precondition-error (patch))
    exit (''Cannot establish preconditions to apply the patch'')
endif

;;; Because this action failed once, take good care when executing it.

execute_with_sensing (patch)
if (error-detected)
    exit (''Cannot redo the failed action'')
endif

;;; Redo carefully the actions that were undone in the first step.

for (i from patch to top)
    if (action_was_undone (i) or
        (post-condition (i) = precondition (j) and
         j is in range (i, top]))
        execute (i)
    endif
endfor

;;; At this point, we are at the state we should have been
;;; had there been no error.

exit (''Success'')

```

Of course, since little is known about the actual state of the world after an error, the actions must be carefully undone in the first part of the recovery procedure, with

as much sensing as practical.

5.1.1 Constraining the Search Space

Error recovery is thus regarded as search for an action or a sequence of actions susceptible to "patch" the plan in order to proceed towards the original goal. This search is expensive in processing time, however, and actually too expensive to be acceptable in an interactive robot environment such as telerobotics. One must then look for ways to reduce the recovery delays. Given that failure reason analysis has identified the ultimate source of the error, the failure of an action due to an information or an operational error, the simplest patch is to attempt to re-execute the failed action with, possibly, a better understanding of the current state of the world. In order to redo this action, its preconditions must be verified, this time with the use of sensors since the state of the world cannot be inferred from the world model after an error. Assuming the error analysis is correct, the failed action is the first source of error and therefore its preconditions were once established correctly. Hence, the preconditions of the failed action missing from the current state of the world must have been undone by steps subsequent to the failed action. The search space can then be reduced to the set of actions that took place between the failed action and the action where the error was detected, searching for actions which have post-conditions similar to the preconditions of the failed action. These *similar* post-conditions are an indication that one aspect of the world that was present and necessary for the failed action to succeed has been altered after the failed action was attempted. They are similar because they are expressed with the same parameters, with different values. If such actions are found, they can easily be undone if *inverse actions* exist for these, namely if the *inverse-action slot* is non-null. Otherwise, the search space must be expanded to the set of all actions in the database, as shown by figure 5.1. When an error occurs, ISER first searches for inverse actions for each of the previous steps.

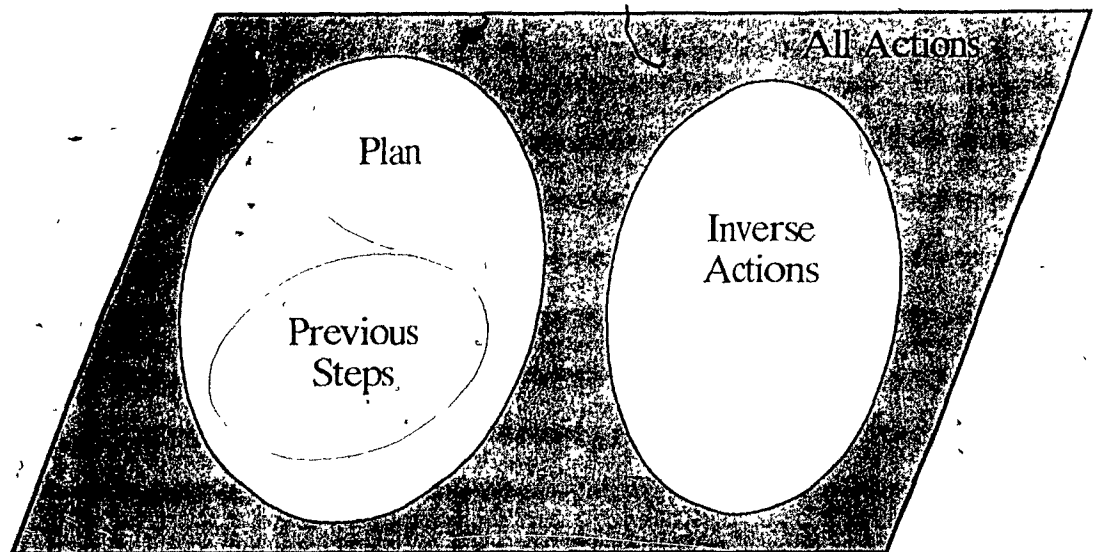


Figure 5.1 The Action Search Space

Figure 5.2 describes the error recovery procedure by comparing the normal sequence of states, S_0 to S_5 , to the sequence of states S_0 to S_{5e} created by an error in the execution of A_2 , detected at S_{5e} . In this example, the error is detected at run time after execution of A_5 and the analysis finds that the error occurred at A_2 .

Suppose: Error occurs at A_2
 The error is detected at S_{5e}
 S_{4e} and S_{3e} conflict with S_1
 S_{2e} and S_{5e} do not conflict with S_1

The recovery then proceeds as follows: Undo A_4, A_3 to achieve S_1
 Redo A_2 (with sensing)
 Redo A_3, A_4 , and A_5 (without sensing)

Note that A_2 and A_5 are not explicitly undone. This is because actions in ISER are atomic (refer to sections 2.3.3 and 2.4) and therefore partial success is never recognized. To cope with the fact that the outcome of the failure of A_5 is unpredictable, it is considered only if S_{5e} interferes with the undoing of A_3 and A_5 by conflicting with their preconditions. Also note that A_3, A_4 , and A_5 are re-executed without special sensing. This follows from the rationale of the original plan; if it was worth attempting A_3 to A_5 without verifying S_2 , it is certainly valid to attempt execution now that S_2 has been established explicitly.

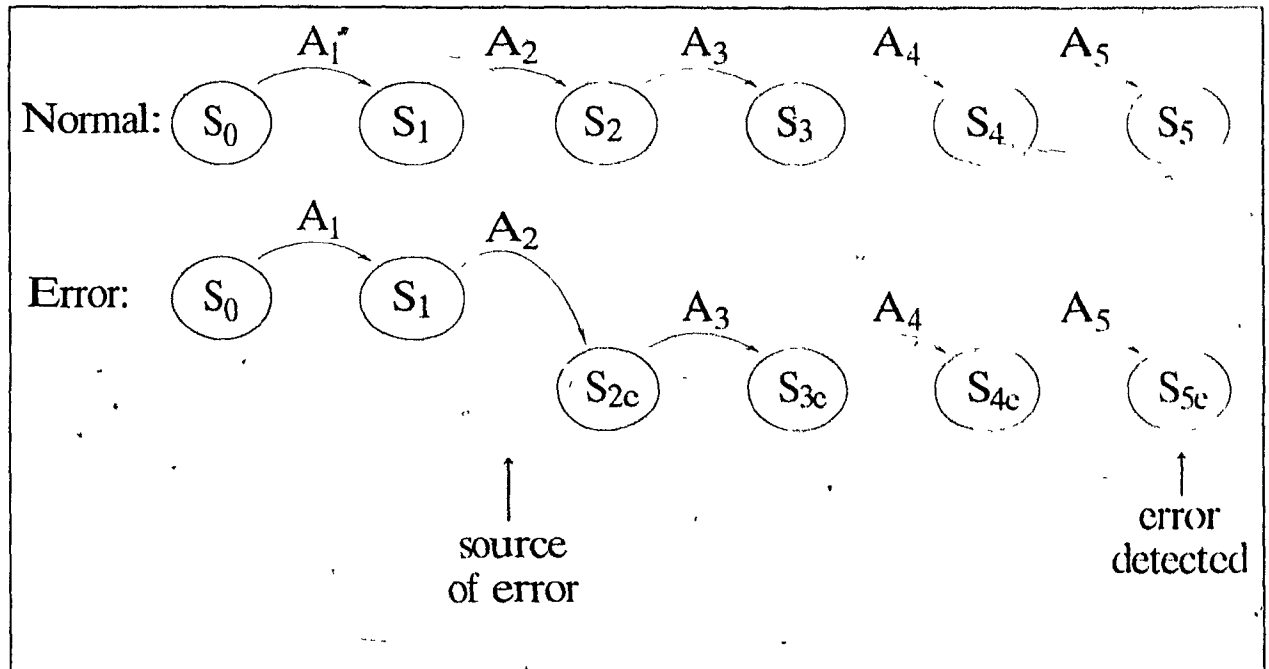


Figure 5.2 Error Recovery

5.2 Recovery Propagation

The class of errors that will be trapped and corrected by ISER has already been discussed. This section deals with how error correction at the lowest level can propagate to repair high-level plans.

As ISER aims to solve errors occurring in the course of an *instance* of the execution of a task, often these errors are due to some particular arrangement of the work cell or some coincidence of events. Indeed, such errors are very common but their treatment does not constitute the core of most tasks; that is, error detection is seldom seen as a goal of the task planner. Assuming then that the task planner has dealt with the general organisation of the program, ISER has the responsibility of coping with variations of the real world to attain the desired goal with the given plan. The recovery plans should be as short and reliable as possible, however, since an error in the execution of a recovery plan stops ISER. This is based on the assumption that failure of a recovery plan indicates a possible recurrence of the error that triggered the execution of the recovery plan in the first

place.

The form of the recovery plans is not different from that of a top level robot plan. Recovery plans are weaker, however, because interaction between steps is never considered; there is no meta-knowledge as there could be in a carefully thought plan. As recovery plans blindly attempt to satisfy preconditions, recovery failure then indicates that some precondition for an action in the recovery plan is missing. Since, in recovery mode, all actions are checked and all preconditions are verified, this means that there is an unachievable state in the recovery plan. If the recovery plan is conceptually valid, the failure to establish the state is a constraint error, beyond the robot's capabilities. If the recovery plan is invalid, for instance if the unachievable state is unnecessary, then there is a flaw in ISER itself, in its database as a required precondition missing from an action frame, in the robot system repeatedly failing to operate, or in a sensor returning an incorrect value. These last four failure types cannot be dealt with at the level of ISER, which would suffer from some form of neurosis⁶, but rather at some higher level. In telerobotics, the operator can evaluate where the fault is by comparing his assessment of the situation with the status report from ISER.

Indeed, upon failure of a recovery plan, the standard error analysis procedure can be applied quickly to narrow down the possible reason for failure to:

- missing precondition (constraint error or sensor failure)
- missing post-condition (constraint error, robot or sensor failure, or incorrect database)

Of course, a flaw in ISER itself can manifest itself in any of a number of ways, all causing self-induced neurosis.

5.3 Example

To illustrate the error recovery algorithm and the recovery procedure in general,

⁶ Either its perception or interpretation of the world is altered or its body, the robot, refuses to operate. Neurosis is an abuse of language, but it depicts well the system behavior.

the following example is typical of many runs of the robot program developed in Hydro-Québec's laboratories, to realise the task outlined in chapter 2. It demonstrates what occurs when an imprecision in the measure of the position of the conductor leads to a collision between the unwrapping tool and the conductor when the tool is mated with the conductor. The original plan is:

```
(d-actrace 1 (move tool-rack))
(d-actrace 2 (pick-tool sonar))
(d-actrace 3 (displace sonar above-conductor))
(d-actrace 4 (apply sonar conductor-position))
(d-actrace 5 (leave-tool sonar))
(d-actrace 6 (pick-tool unwrapping-tool))
(d-actrace 7 (displace unwrapping-tool conductor))
(d-actrace 8 (mate unwrapping-tool conductor))
(d-actrace 9 (apply unwrapping-tool tie-wires))
(d-actrace 10 (apply unwrapping-tool conductor))
(d-actrace 11 (apply robot unwrapping-tool))
(d-actrace 12 (unapply robot))
(d-actrace 13 (unapply unwrapping-tool))
(d-actrace 14 (unmate unwrapping-tool conductor))
(d-actrace 15 (leave-tool unwrapping-tool))
(d-actrace 16 (move cradle))
(d-actrace 17 (stop robot))
(d-actrace 18 (goal))
```

Running it yields a message from the robot controller, which reads, after interpretation by ISER:

```
(robot-error 8 (mobile robot no))
```

Which means that at least one joint motor stalled at step 8. Error analysis yields the following list of preconditions:

(precondition (mobile conductor no))	:given!
(constraint (compatible robot tool))	:ok
(constraint (compatible tool conductor))	:ok
(constraint (mobile tool yes))	:ok, it moved
(constraint (can-reach tool conductor))	:ok
(precondition (mobile robot yes))	:ok, it moved
(needs-information (position conductor))	:not verified

⇒ the only possible cause of failure is an incorrect (position conductor).

Which action was supposed to get (position conductor)?

(d-actrace 4 (apply sonar conductor-position))

There is only one reason why action 4 could fail, imprecision in the sonar. Actually, it could also fail if the sonar measured the position of something else, but we can safely assume the approximate position of the conductor is known (from the operator, say). Hence, there is no action prior to 4 that could have failed.

ISER then proceeds with local planning to recover from the error:

Attempt to redo action 4 with sensing. Actually, action 4 is a sensing operation and hence is a special case, performing "sensing with sensing" is much like sensor integration. The user could be asked for a sensor that could verify a previous sensing operation, but as a first approximation, the original sensor is used to repeat the measurement. This in no way changes the course of the recovery procedure, except for a change in the sensor used to patch the plan

Starting from action 8, trace back for actions with post conditions matching preconditions of:

(apply sonar conductor-position)

these are:

- (holds robot sonar)
- static constraints that have been verified already (compatibilities, etc.)

It may be argued that (d-actrace 6 (pick-tool unwrapping-tool)) violates this, but it would require too much reasoning to figure directly that holding the unwrapping tool precludes holding the sonar. Instead, ISER discovers that

(d-actrace 5 (leave-tool sonar))

violates (holds robot sonar). This is easy to undo, the inverse action is (pick-tool sonar). The preconditions of (pick-tool sonar) have to be validated

(leave-tool whatever*)
(move tool-rack)

i.e., the unwrapping-tool

and finally.

(pick-tool sonar)

The preconditions of (apply sonar conductor-position) are now validated.

We can then define the recovery plan:

```
(defplan 'patch'
  (leave-tool *whatever*)           ;i.e., the unwrapping-tool
  (move tool-rack)
  (pick-tool sonar)
  (execute-with-sensing (apply sonar conductor-position))
  (leave-tool sonar)               ;from list of "undones"
  (pick-tool unwrapping-tool)     ;from list of "undones"
  (displace unwrapping-tool conductor) ;necessary for next action
  (mate unwrapping-tool conductor) ;because it failed
)
```

This plan can then be sent to the controller to be executed by the robot. Conceptually, it consists of the following steps:

- 1° Undo actions to get to a state in which the failed action can be attempted.
- 2° Redo the failed action
- 3° Redo the actions undone
- 4° If any action not in the list of "undones" has post-conditions matching preconditions of one of the undones, redo it anyway in case it was also undone.

5.4 Recovery Cost

Allowing non-catastrophic failures to occur, in the hope that the manifestation of the failure will provide useful clues to recover from the failure and continue execution of the original plan, makes sense only if the total cost of the failure and recovery is low. To verify this, one must address the problem of the trade-off between:

Full sensing.	high computational cost
	sensor complexity
	multiple displacement of sensors
	high reliability and safety
No sensing + recovery.	possibly optimal use of hardware
	low reliability, danger

possibility of learning

ISER is less useful with full sensing; called upon discovery of error, it assumes that errors are due to incorrect information or operation. The best use of ISER lies somewhere in between, because there is a cost in time, resources, and perhaps in material (damages to the objects in the work area). All these costs form the recovery cost, if it is high, the plan should include sensing and be careful to prevent recovery. Conversely, if the recovery cost is small, it may be more efficient to let the robot program run at higher speed and lower complexity.

The recovery cost C_R is a function of.

- the time to compute the patch
- number and cost of the manipulations
- amount of sensing required
- damages caused by the error

For a given plan consisting of the sequence of manipulations:

$$m_1, m_2, m_3, \dots, m_q$$

and properly interspersed sensing operations:

$$s_1, s_2, s_3, \dots, s_r$$

The cost of the program is then the sum of the manipulation cost C_M and of the sensing cost C_S . To a first approximation, all manipulations can be assumed to have the same cost (gear wear, for instance) and so can the sensing operations. Hence, the program or plan cost C_P is proportional to the number of actions. For a given run, the total or task cost C_T is then the program cost plus the recovery cost:

$$C_T = C_P + C_R \quad (1)$$

While ISER itself has no control over the top level planner, which defines the program and hence the program cost, is it desirable to produce an estimate of the expected

recovery cost, to reduce the overall average task cost. Indeed, C_T cannot be eliminated by any trade-off in program versus recovery since both contain terms in manipulation and sensing. If one refines (1) to include a measure of the probability of error

$$C_T = C_P + \rho C_R \quad (2)$$

where:

$$0 \leq \rho \leq 1$$

Then, starting from a complete "perfect" plan which accounts for every possibility of failure and prevents them, errors are impossible and $\rho = 0$. On the other hand, if one allows errors to occur ($\rho > 0$) by removing some of the manipulation and sensing operations from the perfect plan, C_P decreases while the C_R term is more important. The probability of error raises as less checking is performed and, consequently, ρ is directly proportional to the number of operations left out from the perfect plan. Ultimately, one wants to minimize C_T by minimizing C_P while keeping ρ low enough to limit the effect of C_R .

Refining the evaluation of C_R , one can argue that C_R also increases with ρ as C_P is decreased. Using the model of the perfect plan, this is explained by the fact that, if error detection and correction steps are eliminated from the perfect plan and an error occurs, it is likely that the removed steps were necessary and will have to be done, perhaps in a modified form, in the error recovery process and thus raise the value of C_R . To approximate, again, C_R can be divided into two components: the cost of error analysis and of undoing failed steps, the error complexity (E_C), and the cost of doing steps omitted from the perfect plan, the "error detection" cost (C_D). Rewriting (2), one obtains

$$C_T = C_P + \rho(E_C + C_D) \quad (3)$$

where:

$$C_D + C_P = C_{P_p} \quad \text{the cost of the perfect plan}$$

The error complexity is largely related to the amount of search required to find the ultimate source of error. If many checks are performed during plan execution, the error analysis is bounded, errors are detected sooner, and the number of steps to undo is reduced, all factors contributing to reduce the cost associated with the error complexity. E_C can thus be estimated from the following assumptions.

- a) On the average, failures will occur half way between two "checks" in program execution
- b) The number of operations to verify after an error is that portion of the perfect plan that would fall between the last check and the action during which the error was detected.
- c) The number of steps to undo is that portion of the actual plan that occurred since the last check
- d) If actions must be executed in a given order, the error analysis cost is decreased since success of an action guarantees success of previous actions
- e) If actions can be executed in any order, the cost of undoing steps decreases as the stack of actions is shallower

The error complexity is divided into two terms, the portion due to the extent of the analysis required and the part due to the number of steps that have to be undone. From the above assumptions, and given that the actual plan is a selection of steps from the perfect plan, the relationship between the actual plan and the perfect plan can be depicted as in figure 5.3 below.

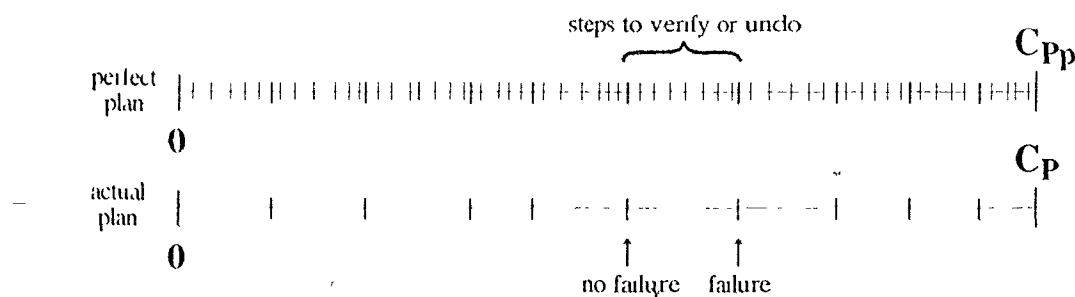


Figure 5.3 Perfect and Actual plans

Assuming an even distribution of "checks" over the actual plan, the analysis cost is proportional to the number of steps in the perfect plan over the actual number of steps:

$$\text{analysis cost} \propto \frac{C_{Pp}}{C_P}$$

From *d* above, and using *S* as a measure of how many steps must be executed in a given order, to denote the "serial" nature of the task, the analysis cost can be refined as:

$$\text{analysis cost} \propto \frac{1}{S} \frac{C_{Pp}}{C_P}$$

This is consistent with the fact that *all* conditions must be verified if they can be established in any order. Finally, from *a*:

$$\text{analysis cost} = \frac{1}{2} \frac{1}{S} \frac{C_{Pp}}{C_P}$$

The cost of undoing steps can be similarly derived from *a*, *c*, and *e*:

$$\text{undoing cost} = \frac{1}{2} S \frac{C_{Pp}}{C_P}$$

Hence, the error complexity evaluates to.

$$\begin{aligned} E_C &= \frac{C_{Pp}}{C_P} \frac{1}{2} \frac{1}{S} + \frac{C_{Pp}}{C_P} \frac{1}{2} S \\ &= \frac{1}{2} \frac{C_{Pp}}{C_P} \left(\frac{1}{S} + S \right) \approx \frac{1}{2} \frac{C_{Pp}}{C_P} S \end{aligned}$$

Rewriting (3), we obtain this final form to represent the approximate cost of a given task:

$$C_T = C_P + \rho C_D + \rho E_C + \rho E_S \quad (4)$$

$$C_P + C_D = C_{Pp}$$

$$\rho = F \frac{C_D}{C_{Pp}}$$

$$E_C = \frac{1}{2} \frac{C_{Pp}}{C_P} S$$

where

F, S : constants depending on the task and domain

C_T : task cost

C_P : program (plan) cost = number of actions ($q + r$) in plan

C_D : error detection cost = $C_{Pp} - C_P$

E_C : error complexity

E_S : error severity

ρ : probability of error

C_{Pp} : cost of perfect plan = estimate of number of actions
to ensure proper execution

A new term, the error severity (E_S), has been added to artificially raise the cost of certain errors for safety or cost purposes (e.g., prevent shorting phases of power lines). Strictly speaking, this would not be required if error analysis and world modeling were complete and sufficient to describe all effects of actions. For example, if the planner understood enough of the real world to model the implications of a short circuit between the phases (power outage, conductor damage, robot replacement, etc.) it would put a high value of error complexity for these steps.

F is constant and depends on the task/domain. In the blocks world, for instance, there are no operational nor information errors. Using a generic "perfect plan" which includes sensing steps, one can optimize it by removing these redundant steps. In the blocks world, $F = 0$, and the task cost equation is reduced to

$$C_T = C_P$$

Conversely, in harsh environments, sustaining heavy winds for instance, errors are more likely to occur if sensing is reduced; objects can move and fall, for example. This is not related to the complexity of errors; displacements of objects is easy to resolve and fix. Therefore, F is a measure of the necessity to perform sensing and verification of actions

and an indication of how delicate the task is. It is actually defined to be the percentage of actions that will cause a failure if they are not executed.

Independently of the probability of error, error complexity is a function of the plan and domain of application. If the plan consists of several actions that must be executed in a certain order, the errors are more complex to analyze and resolve since an error early in the execution of the plan can have repercussions on many actions afterwards. On the other hand, if the actions can be executed in any order (in parallel), errors can be fixed easily. By analogy with the blocks world, one can see that building a tower will yield a higher error complexity than laying out a row of blocks. Therefore, S is defined to be the ratio of the number of steps that must be executed in succession over the number of steps that can be executed in any order or in parallel. This figure is actually related to the shape of the failure tree, if it is high and narrow, the search can be very deep and S is high, if, on the other hand, the failure tree is wide but short, S is low.

The following figure illustrates typical program costs for C_P varying from 1 to 100, for $C_{Pp} = 100$, $E_S = 0$, and various values of F and S .

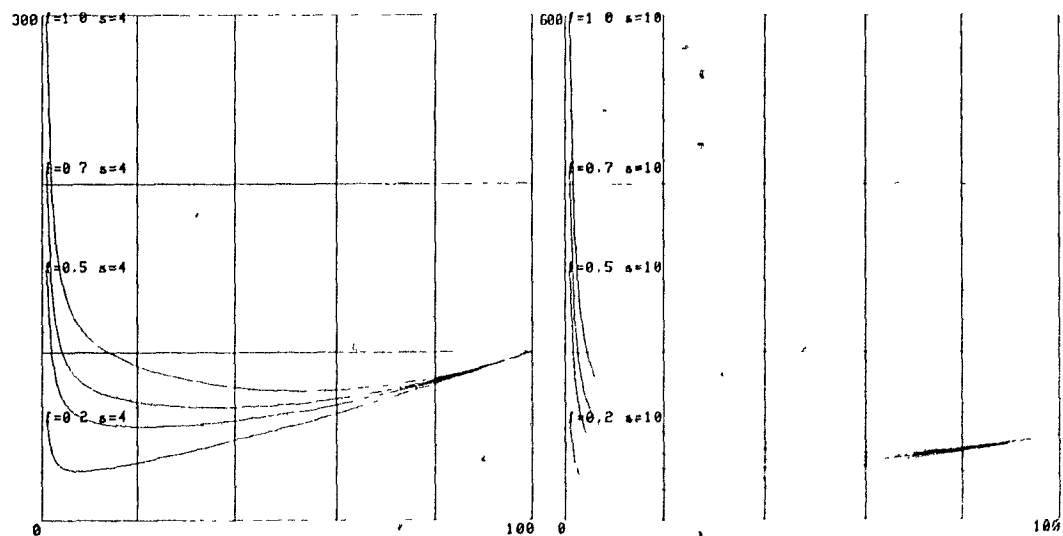


Figure 5.4 Program Costs as function of the Number of Steps

This shows that, as the probability of error decreases, it is worth executing less actions and performing less checking and sensing. As expected, the benefit decreases in

the case where more actions have to be executed in order (note the scale difference between the two figures). Also expected is the fact that all curves join at the point where all actions are executed (a perfect plan) and no error occurs. The actual decrease in task cost should be more dramatic (it varies between 25% and 60%) but this analysis does not take into account the very high cost of executing some parts of a perfect plan. In reality, not all actions carry the same cost and the cost of the perfect plan is prohibitively high, raising the right end of the curve and thus increasing the relative benefit of performing tasks with less sensing.

6.1 Summary and Discussion

The goal of telerobotics is to accelerate completion of tasks performed by a human operator remotely controlling a manipulator. It quickly becomes evident that a computer interface is necessary to translate the operator's intentions into robot instructions and, conversely, to translate robot coordinates into positions and relations. This thesis has addressed a part of this goal: the problem of robot error recovery in a poorly characterised environment. Such a context imposes several constraints, the most severe being the possibility of interruption of the original plan by the operator. Consequently, *on-line* intelligence must supplement pre-execution planning for such unexpected interruptions. Furthermore, the differences between the actual world and the expected world upon which the original plan is based cause actions to fail in ways difficult or expensive to predict.

ISER has been designed to tackle these two problems, asynchronous interruption and randomness of the work environment, by using a world model to monitor the execution of the task as a sequence of actions rather than as the achievement of a top-level goal. Success is then measured for each action as they are executed, allowing changes to the original plan without affecting the general performance of the system. In order to do this, ISER includes definitions of all the actions that can be executed by the robot system; these definitions include restrictions on the state in which the action can be executed and the

expected outcome of the action. If the restrictions are not satisfied, an error is signaled and the robot system is stopped to prevent further damage. At this point, the failure is analysed to discover the source of the error. The central idea in ISER is to use local planning to reduce the complexity of the planning required to recover from the error; a great confidence is put on the original plan and only as a last resort will ISER attempt to build a simple, local, recovery plan to replace it. Actually, ISER attempts to cope with variations in the real world to follow the original plan.

Just as its world model adapts to the actual state of the work cell, ISER is relatively independent of the context and could be used in a wide range of applications. New objects and actions can be added as variants or specialisations of those already in place to customise the world model to the particular context.

6.2 Contributions

ISER expands on the idea of error recovery and operator-assisted manipulator by providing run-time assistance and failure recovery. This is in contrast with common methods involving a planner to build a new plan from the failure state to the goal state; in ISER rather, local planning is used to patch the original plan and salvage it. Consequently, the plan can be repaired with less resources than those required by a full-fledged planner. As it attempts to verify all preconditions before execution of actions, ISER also provides error prevention in the form of messages warning the operator about the possibility of future failures.

As a side effect, ISER required the creation of a high-level parser to interpret robot messages and determine which objects are directly affected by an action. The parser is required to raise the level of abstraction from what the robot system uses to what a qualitative system can understand. This representation and the associated world model, both still incomplete, could be used to elaborate a more complete system to reason about object relations and thus maintain a world model to present information to the operator in a

clear and flexible manner. The way the information is organised is also of interest; instead of having a separate database for the world model, objects carry their own definitions of the changes they produce in the world, as well as the restrictions on the states in which they can be used or applied. The related notion of action classes is viable, but it seems more promising to incorporate action definitions in objects, going from frame-based representation to true object-oriented programming.

6.3 Future Work

The definitions of actions as separate items in the database is an artifact of the original version of ART which did not support object-oriented programming, as the current one (3.1) does. Describing actions as attributes of objects (methods) would greatly simplify expansion and customisation of the database. Limited action attributes already exist for some objects, but they must be processed explicitly by special rules. Using methods, world model update would be performed simply by sending a message to the objects involved indicating which action was executed. Determining which objects are involved is already done by the high-level parser.

As ISER handles errors, the task planner can take advantage of this to generate more general plans. Further, the plans should include sensing operations to aid in error detection. As a matter of fact, the analysis of the task cost can be used to evaluate how much processing and verification must be included in the original plan and how much can be left off to be executed only in case of error, in order to minimise the average cost of the task. Since ISER analyses the sources of error and finds ways around those errors, a learning system could also complement ISER and pass this information back to the planner in order to generate more efficient plans. Actually, from the planner point of view, ISER serves as an extension of the robot capabilities and presents a constant model independent of the robot and environment.

Experimenting with the system has unveiled new avenues in interactive robot task specification, as all objects can be represented by a set of characteristic/value pairs.

the actions can be defined by the operator by indicating the final desired values (compare for instance figures A.3 and A.4). This would require a sophisticated planner if many such values can be changed at once by the operator but it can be implemented easily for small changes in the world model, allowing interactive object-oriented programming of the robot. This is a great advantage since telerobots are really meant to be used by personnel unfamiliar with robots and great experience in the domain of application. Ultimately, one wants to render the robot as transparent as possible; the manipulator is a tool, not an end by itself. ISER aims at that by compensating for the environment, but its interface to the robot system is not sufficiently versatile to accommodate any robot system. Several schemes have been developed to achieve this, one of which models the robot system as an operating system, allocating resources, interfacing, etc [Carayannis88]. Again, the central idea being an abstraction of the robot and its work space to present the operator a *functional* view of his tool.

Appendix A. Sample ISER Session

Certain ideas expressed in this thesis have been implemented and explored with a simulated task. The sequence of actions, the sensor values, and various errors were defined in a file and ISER simply stepped through them to simulate sequential execution by a manipulator. This appendix demonstrates ISER's behaviour through a few different scenarios.

First is a simple warning displayed when ISER finds that the next action is in contradiction with the current state of the world. It reflects the fact that the manipulator cannot handle two tools at the same time (the unwrapping tool and a gripper), and hence the request to install the gripper (by an unexperienced operator, for instance) is likely to produce an error. It is not forbidden, however, to be consistent with ISER's philosophy that it is generally better to let execution continue, unless it can be proven that a failure has occurred. In this particular case, one can imagine that the unwrapping tool was dropped by accident, perhaps causing damage to the tool mount detector, and the intent of the operator is to use the gripper to recuperate it. In figure A.1, the ISER Status window lists the current action and the next one. The UNWRAPPING TOOL window shows that ISER believes the tool is attached to (mounted on) the robot, and the ISER Warnings window indicates what makes ISER believe that the next action will fail. All this information was deduced from the standard action, tool, and objects definitions.

Figure A.1 Typical ISER display

<p>COMMAND WINDOW</p> <p>=> execute next action</p> <p>=> execute next action</p> <p>=> execute next action</p> <p>=> █</p>		<p>ISER COMMANDS</p> <p>execute next action</p> <p>restart</p> <p>examine object</p> <p>hide object</p> <p>enter action</p> <p>expose ART menu</p> <p>hide ART menu</p>
<p>ISER Warnings</p> <p>The (EMPTY ROBOT YES) precondition of action 3 was not met.</p>		<p>ISER Status</p> <p>Tracing action 2</p> <p>--> (PICK-TOOL UNWRAPPING-TOOL)</p> <p>Next action</p> <p>(PICK-TOOL GRIPPER)</p>
<p>ROBOT</p> <p>Position UNWRAPPING-TOOL</p> <p>Mobile YES</p>	<p>UNWRAPPING TOOL</p> <p>Position TOOL-RACK-POSITION</p> <p>Mobile YES</p> <p>Attached to ROBOT</p>	<p>CONDUCTOR</p> <p>Position CONDUCTOR-POSITION</p> <p>Mobile NO</p> <p>Attached to INSULATOR</p>

Figure A.2 lists the choices available to the operator, who can enter and manipulate the sequence of actions to be executed and select which objects from the world model will be displayed on the screen.

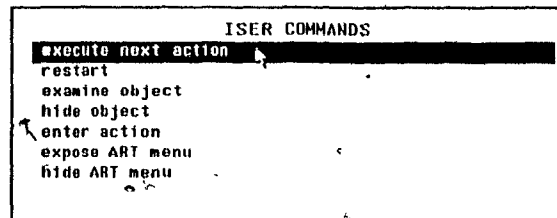


Figure A.2 The ISER menu

Figures A.3 and A.4 give a simple example of the effects of actions on the world model and how this reflects on the display. They illustrate the differences in the characteristics of the conductor before the task is executed and after the tie wires are removed, leaving the conductor unattached and mobile, still at its original position.

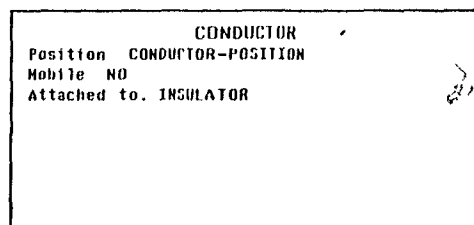


Figure A.3 The conductor object, initial state.

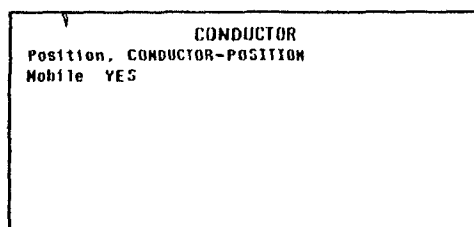
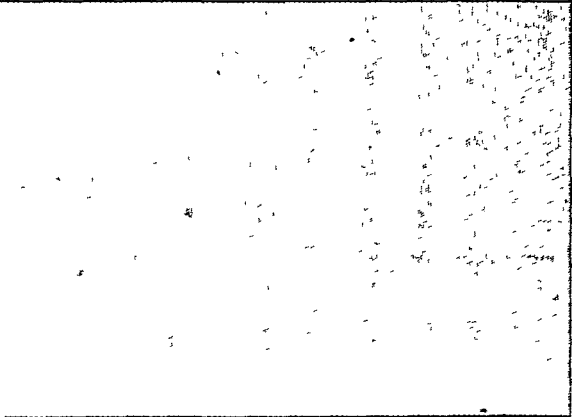


Figure A.4 The conductor object, goal state

The last figure depicts a typical display immediately after an error has been detected, namely a joint stalled while removing the unwrapping tool from the conductor. This could be caused, for instance, by misalignment of the slot allowing the unwrapping tool to slide in and out (mating/unmating) on the conductor. At this point, the status window indicates that ISER is performing error analysis and a special Error Analysis window lists the current deductions. The warning window also points out that the failed action is necessary for the next one and should thus be fixed before continuing. Applying the process of error analysis described in chapter 4 will reveal the necessary preconditions for this action and proceed to verify and establish them.

Figure A.5 Error analysis state

<p>COMMAND WINDOW</p> <pre>=> => execute next action => execute next action => execute next action => execute next action => => execute next action</pre>		<p>ISER COMMANDS</p> <pre>execute next action restart examine object hide object enter action expose ART menu hide ART menu</pre>
<p>ISER Warnings</p> <pre>Action 18 failed to achieve the goal (MOBILE ROBOT NO). The (MOBILE ROBOT YES) precondition of action 11 was not met</pre>		
<p>Error Analysis</p> <pre>Robot error at action 18 (MOBILE ROBOT NO) Goal error at action 18 (MOBILE ROBOT NO)</pre>		
<p>ISER Status</p> <pre>Analysing error (MOBILE ROBOT NO) of action 18 --> (UNMATE UNWRAPPING-TOOL CONDUCTOR) Next action. (LEAVE-TOOL UNWRAPPING-TOOL)</pre>		
<p>ROBOT</p> <pre>Position CONDUCTOR Mobile NO</pre>	<p>UNWRAPPING TOOL.</p> <pre>Position CONDUCTOR-POSITION Mobile NO Attached to ROBOT</pre>	<p>CONDUCTOR</p> <pre>Position. CONDUCTOR-POSITION Mobile YES</pre>

Appendix B. The High Level Parser

The high level parser is the interface between the robot system, which operates on physical objects, and ISER itself, which operates on classes of objects. To be able to specify pre- and post-conditions for action classes, actions must be parameterised and variables must be used in action descriptions. The parser receives action specifications from the robot and then instantiates conditions in the database from the robot values and the general action definitions. This process is best explained by considering the set of acceptable action specifications as a robot programming language (it is, conceptually, at the same level) and to use the following Backus-Naur form (BNF) to represent it:

$$\begin{aligned}\langle \textit{actrace} \rangle &::= (\textit{actrace} \langle \textit{positive-integer} \rangle \langle \textit{action-def} \rangle) \\ \langle \textit{action-def} \rangle &::= (\langle \textit{action} \rangle \{ \langle \textit{subject} \rangle \} \langle \textit{argument} \rangle) \mid \\ &\quad (\langle \textit{action} \rangle \langle \textit{effector} \rangle) \{ \langle \textit{subject} \rangle \} \mid \\ &\quad (\textit{goal}) \\ \langle \textit{action} \rangle &::= \textit{move} \mid \textit{pick-tool} \mid \textit{displace} \mid \textit{mate} \mid \textit{apply} \mid \\ &\quad \textit{unapply} \mid \textit{unmate} \mid \textit{leave-tool} \mid \textit{stop} \\ \langle \textit{argument} \rangle &::= \langle \textit{subject} \rangle \mid \langle \textit{abstract-position} \rangle \\ \langle \textit{subject} \rangle &::= \langle \textit{object} \rangle \\ \langle \textit{abstract-position} \rangle &::= \langle \textit{object} \rangle \\ \langle \textit{positive-integer} \rangle &::= 1, 2, 3, \dots\end{aligned}$$

For every action received from the robot controller, the parser looks up in the definition of the action to determine if it has default values and then explicitly posts its instantiated parameters

References

- Brooks82** Brooks, R. A. , "Symbolic Error Analysis and Robot Planning", *International Journal of Robotics Research*, Vol 1, No 4, Winter 1982
- Carayannis88** Carayannis, G , "A Generic Run-time Environment for a Robotic Work cell", Ph D Dissertation, McGill University, Montreal, Canada, June 1988.
- Cardelli88** Cardelli, L and Wegner, P , "On Understanding Types, Data Abstraction, and Polymorphism", *ACM Computing Surveys*, Vol. 17, No. 4, December 1985, pp. 471-522
- Cohen87** Cohen, J. , "Live-Line Repair with TOMCAT", *EPRI Journal*, Vol. 12, no 5, July/August 1987, pp 14-19
- Donald86** Donald, B. R , "Robot Planning with Uncertainty in the Geometric Models of the Robot and Environment: A Formal Framework for Error Detection and Recovery ", *Proc. IEEE Int. Conf. on Robotics and Automation 1986*, pp 1588-1593, 1986.
- Ernst69** Ernst, G. W. and Newell, A , "GPS: A Case Study in Generality and Problem Solving", Academic Press, New York, NY, 1969
- Fikes71** Fikes, R. E. and Nilsson, N. J , "STRIPS: A New Approach to the Application of Theorem Proving to Problem Solving ", *Artificial Intelligence*, Vol 2 (1971), pp. 189-208.
- Genesereth87** Genesereth, M. R and Nilsson, N. J , "Logical Foundations of Artificial Intelligence", Morgan Kaufman Publishers Inc , Los Altos, CA, 1987
- Gini85** Gini, M. et al. , "The Role of Knowledge in the Architecture of a Robust Robot Control", *Proc. IEEE Int. Conf on Robotics and Automation 1985*, pp. 561-567, 1985.
- Ginsberg88** Ginsberg, M. L and Smith, D. E , "Reasoning about Action I: A Possible Worlds Approach", *Artificial Intelligence*, Vol 35 (1988), pp 165-195.
- Girard88** Girard, P. , "La robotique en distribution", Technical report no. IREQ-4167C, *Institut de recherche d'Hydro-Québec*, Varennes, QC, Canada, April 1988.
- Hayes-Roth83** Hayes-Roth, F. , "Using Proofs and Refutations to Learn from Experience", in *Machine Learning: An AI Approach* Michalski, Carbonnell, and Mitchell, eds. Tioga Publishing Co , Palo Alto, CA 1983.
- Henkener85** Henkener, J. A , "Study of a Component Evaluation Unit for Remote maintenance of Transmission Lines" Interim report for RP1497-1, prepared by Southwest Research Institute, Electric Power Research Institute report no EL-4188, August 1985.

- IERE87** IERE of Japan, "New Applications of Electronics in Power Facilities of Japan". Report Submitted to 16th General Meeting of IERE, December 1987.
- Inference87** Inference Corp. , "ART Reference Manual". Inference Corp., Los Angeles, CA. Version 3.0, January 1987.
- Kaemmerer87** Kaemmerer, W. F., Allard, J. R. , "An Automated Technique for Providing Moment-by-Moment Advice Concerning the Operation of a Process", *Proc. AAAI-87 Sixth National Conference on Artificial Intelligence*, pp. 809-813, 1987.
- Kak86** A. C. Kak et al., "A Knowledge-Based Robotic Assembly Cell", *IEEE Expert*, Vol. 1 No. 1, pp. 63-83, Spring 86.
- Lee84** Lee, M. H., Hardy, N. W., and Barnes, D. P. , "Research into automatic error recovery", *Proc. I. Mech. E. Colloquium on U. K. Robotics Research*, Paper C463/84, London, 1984.
- Moya86** Moya, M. M. and Davidson, W. M. , "Sensor-Driven, Fault-Tolerant Control of a Maintenance Robot", *Proc. IEEE Int. Conf. on Robotics and Automation 1986*, pp. 428-434, 1986
- Myers86** Myers, W. , "Introduction to Expert Systems", *IEEE Expert*, Vol. 1 no. 1, Spring 1986, pp. 100-109
- NASA85** NASA Task Force [directed by D. R. Criswell], "Robotics for the United States Space Space Station", *Robotics*, Vol. 1 no 4, December 85, pp. 205-222.
- Nilsson80** Nilsson, N. J. , "Principles of Artificial Intelligence", Tioga Publishing Co., Palo Alto, CA. 1980.
- RSI85** Robotic Systems International, Ltd. , "Application of Robotics to Distribution Systems", Canadian Electrical Association report no. 190 D 392, Montreal, Canada, December 1985.
- RSI88** Robotic Systems International, Ltd. , "Application of Robotics to Distribution Systems, SKYARM Functional Specifications", Canadian Electrical Association report no. 190 D 392, Montreal, Canada, revision 3, March 1988.
- Ramamoorthy87** Ramamoorthy, C. V , Shekhar, S., Garg, V. , "Software Development Support for AI Programs", *IEEE Computer*, Vol. 20 no. 1, January 1987, pp. 30-40.
- Shafer86** Shafer, S. A., Stentz, A , Thorpe, C. E. , "An Architecture for Sensor Fusion in a Mobile Robot", *Proc. IEEE Int. Conf. on Robotics and Automation 1986*, pp. 2002-2011, 1986.
- Sheridan86** Sheridan, T. B. , "Human Supervisory Control of Robot Systems", *Proc. IEEE Int. Conf. on Robotics and Automation 1986*, pp. 808-812, 1986
- Smith86** Smith, R. E. and Gini, M. , "Robot Tracking and Control Issues in an Intelligent Error Recovery System", *Proc. IEEE Int. Conf. on Robotics and Automation 1986*,

pp. 1070-1075, April 1986.

Srinivas77 Srinivas, S. , "Error Recovery in Robots Systems", Ph. D. Thesis, California Institute of Technology, 1977.

Srinivas78 Srinivas, S. , "Error Recovery in Robots through Failure reason Analysis", *Proc. National Computer Conference*, pp. 275-282, 1978.

Stefik85 Stefik, M. and Bobrow, D. G. , "Object-Oriented Programming: Themes and Variations", *The AI Magazine*, Vol. 6, no. 4, Winter 85. pp. 40-62.

Sussman75 Sussman, G. J. , "A Computer Model of Skill Acquisition", American Elsevier, New York, NY, 1975.

Thunborg86 Thunborg, S. , "A Remote Maintenance Robot System for a Pulsed Nuclear Reactor", *Proc. IEEE Int. Conf. on Robotics and Automation 1986*, pp. 442-447, April 1986.

Unimation83 Unimation Inc. , "Puma Mark II Robot, Equipment and Programming Manual", Unimation Inc., CT., August 1983.

Unimation86 Unimation Inc. , "User's Guide to VAL II", Unimation Inc., CT. Version 2.0, February 1986.

Will85 Will, R.W. , "TRICCS: A Proposed Teleoperator/Robot Integrated Command and Control System for Space Applications", *NASA Technical Memorandum*, No. 87577, Langley Research Center, July 1985.

Yoerger87 Yoerger, D. R., Slotine, J-J. E. , "Supervisory Control Architecture for Underwater Teleoperation", *Proc. IEEE Int. Conf. on Robotics and Automation 1987*, pp. 2068-2073, 1987.