Technical Report 834

Justified Generalization: Acquiring Procedures from Examples

Peter Merrett Andreae

MIT Artificial Intelligence Laboratory

This blank page was inserted to preserve pagination.

Justified Generalization:
Acquiring Procedures From Examples

by

Peter Merrett Andreae

B.E.(Hons), University of Canterbury, New Zealand
(1976)

S.M., Massachusetts Institute of Technology
(1981)

Submitted in Partial Fulfillment
of the Requirements for the
Degree of

Doctor of Philosophy

at the
Massachusetts Institute of Technology
January, 1985

© Massachusetts Institute of Technology 1984

Signature of Author .....
Department of Electrical Engineering And Computer Science
September 1, 1984

Certified By ..... Patrick Henry Winston .....
Professor Patrick H. Winston
Thesis Supervisor

Accepted By .....
Professor Arthur Smith
Chairman, Department Committee on Graduate Students

This empty page was substituted for a blank page in the original document.

Justified Generalisation: Acquiring Procedures From Examples

by
Peter M. Andreae

Submitted to the Department of Electrical Engineering and Computer Science
on Dec 28, 1985, in partial fulfillment of the requirements for the degree of
Doctor of Philosophy in Computer Science

Abstract

This thesis describes an implemented system called NODDY for acquiring procedures from examples presented by a teacher. Acquiring procedures from examples involves several different generalization tasks. Generalization is an underconstrained task, and the main issue of machine learning is how to deal with this underconstraint. The thesis presents two principles for constraining generalization on which NODDY is based. The first principle is to exploit domain based constraints. NODDY demonstrates how such constraints can be used both to reduce the space of possible generalizations to manageable size, and how to generate negative examples out of positive examples to further constrain the generalization. The second principle is to avoid spurious generalizations by requiring justification before adopting a generalization. NODDY demonstrates several different ways of justifying a generalization and proposes a way of ordering and searching a space of candidate generalizations based on how much evidence would be required to justify each generalization.

Acquiring procedures also involves three types of constructive generalization: inferring loops (a kind of group), inferring complex relations and state variables, and inferring predicates. NODDY demonstrates three constructive generalization methods for these kinds of generalization.

Thesis Supervisor: Dr. Patrick Winston
Title: Professor of Computer Science

This report describes research done at the Artificial Intelligence Laboratory of the Massachusetts Institute of Technology. Support for the laboratory's artificial intelligence research is provided in part by the Advanced Research Projects Agency of the Department of Defense under Office of Naval Research contract N00014-80-C-0505.

Acknowledgments

There are many people to thank for their part in this work. My thanks especially:

To Patrick Winston, my supervisor, for encouragment, advice, and believing that I had something even when I thought I didn't.

To my father, J.H. Andreae, who got me interested in this problem in the first place, even though I do things differently.

To Dan Brotsky, Bob Hall, Tomas Lozano-Perez and David Kirsh for enlightenment and help at some crucial points.

To my readers, Tomas Lozano-Perez and Bob Berwick for reading a draft at the last minute.

To my office-mates over the past $7\frac{1}{2}$ years, especially Karen Wiekert, Dan Brotsky, Bonnie Dorr, Ken Forbus, and Judy Zinnikas, for help, encouragement, tolerating my quirks, and very enjoyable times.

To the M.I.T. AI Lab, that is tolerant and highly critical at the same time, and to Patrick Winston for running it in such an enlightened way.

And to my wife, Kathleen, for living through it all with me.

To Christopher

Who learns much more and much better than NODDY

And who is much more fun.

Contents

1 Introduction 9
1.1 Scenario 11
1.1.1 Another Example: The TURTLE Procedure 23
1.2 Learning Tasks 27
1.2.1 What the Learner Must Learn 27
1.2.2 What the Learner Learns From 29
1.3 Generalization 33
1.4 An Instructable Robot 38
1.5 Background 39
1.5.1 Concept Learning in Artificial Intelligence 40
1.5.2 Relation to Van Lehn's SIERRA 42
1.5.3 Inductive Inference 43
2 Task, Representation, and Domain 44
2.1 Procedures and Examples 44
2.2 Representation of Procedures 44
2.3 Representation of Patterns and Conditions 48
2.4 Representation of Actions 49
2.5 2-D Robot domain 53
2.5.1 Actions 53
2.5.2 Patterns and Conditions 54
3 Generalizing Procedures 56
3.1 Incremental, Syntactic Acquisition 56
3.2 Generalizing Structured Descriptions 57
3.2.1 Backtracking or Determinism 59
3.2.2 Justification Based Matching 61
3.3 Overview of the Matching Stages in NODDY 62
4 Structure Matching and Generalization 68
4.1 Stage 0: Include 68
4.2 Stage 1: Skeleton 69
4.3 Stage 2: Propagation 73
4.3.1 Generalizations in the Propagation Stage 77
4.4 Stage 3: Grouping 77
4.4.1 Event Generalization in the Grouping Stage 80
4.4.2 Introduction of Loops 83
4.5 Stage 4: Parallel Branches 86
4.6 Stage 5: Nondeterministic Forks 91
5 Action Matching and Generalization 95
5.1 First Level Action Generalization 95
5.1.1 Equality testing 96
5.1.2 Matching and Generalizing with the Action Hierarchy 97
5.1.2.1 Generalizing Actions 97
5.1.2.2 The Action Template Hierarchy 100
5.1.3 Checking Inclusion 103
5.1.4 Finding Generalizations 104

5.2 Second Level Action Generalization . . . 107
5.2.1 Finding the Input/Output Pairs . . . 107
5.2.2 Searching for a Function . . . 108
5.2.3 Expression Building Algorithm . . . 111
5.3 Future Improvements to the Action Matcher . . . 114
6 Condition Matching and Generalization . . . 118
6.1 Ordering the Search Space . . . 119
6.1.1 Patterns and Conditions . . . 119
6.1.2 Complexity Ordering . . . 121
6.1.3 Content of Templates . . . 123
6.2 Matching Patterns and Conditions . . . 125
6.2.1 Matching Condition Nodes: Include . . . 125
6.2.2 Matching Condition Nodes: Conflict . . . 127
6.3 Generalizing Patterns and Conditions . . . 128
6.3.1 First Level Condition Generalization . . . 129
6.3.2 Second Level Condition Generalization . . . 131
7 Generalization Principles . . . 133
7.1 Exploiting Constraints . . . 133
7.1.1 Reducing the Generalization Space . . . 134
7.1.2 Constructing Negative Examples . . . 135
7.1.3 Guiding Structure Matching . . . 136
7.2 Constructive Generalization Methods . . . 136
7.2.1 Inferring Groups in Structured Descriptions . . . 137
7.2.2 Inducing Relations . . . 138
7.2.3 Generalizing Parameterized Objects: Climbing the Template Hierarchy . 139
7.3 Justification . . . 140
7.3.1 An Ordering on Generalizations . . . 141
7.3.2 A Search strategy . . . 142
7.3.3 Sources of Justification . . . 142
7.3.4 Summary . . . 147
8 Conclusion . . . 148
8.1 Summary . . . 148
8.2 Future Directions . . . 149
8.2.1 Justification . . . 149
8.2.2 Extensions to NODDY . . . 150

Figures

Fig. 1-1 The Warehouse Setup for the RETRIEVE procedure. 12
Fig. 1-2 The Goal Procedure: RETRIEVE. 13
Fig. 1-3 The examples presented by the teacher. 14
Fig. 1-4 The First Trace. 15
Fig. 1-5 RETRIEVE-2 17
Fig. 1-6 RETRIEVE-3 18
Fig. 1-7 A fragment of the intermediate stage in forming RETRIEVE-4 19
Fig. 1-8 RETRIEVE-4 20
Fig. 1-9 A fragment of the intermediate stage in forming RETRIEVE-5 21
Fig. 1-10 RETRIEVE-5 22
Fig. 1-11 The TURTLE Procedure 23
Fig. 1-12 The First Two Traces of TURTLE Procedure 24
Fig. 1-13 The Version of TURTLE After Two Traces 25
Fig. 1-14 The Third Trace for Acquiring TURTLE 26
Fig. 2-1 A Simple Procedure 45
Fig. 2-2 Deterministic Branching 47
Fig. 2-3 A Functional Action 51
Fig. 3-1 Adding a Branch 58
Fig. 3-2 Forming a Loop 59
Fig. 3-3 Forming a More General Branch 60
Fig. 3-4 Skeleton and Propagation Stages 63
Fig. 3-5 Introducing a Loop 64
Fig. 3-6 Procedure with Parallel Branches. 65
Fig. 3-7 The Generalized Branch with Functional Actions 66
Fig. 3-8 A Nondeterministic Fork from the RETRIEVE Procedure 67
Fig. 4-1 Overview of the Structure Matcher. 69
Fig. 4-2 Matching Functional Actions 70
Fig. 4-3 Blocking the Stepping 71
Fig. 4-4 Stepping Backwards from Stop Events. 72
Fig. 4-5 Skeleton for RETRIEVE-3 74
Fig. 4-6 Propagation 76
Fig. 4-7 Forming A Group from Pairs 78
Fig. 4-8 Results of Grouping 81
Fig. 4-9 Generalizing Conditions 82
Fig. 4-10 Forming Loops 84
Fig. 4-11 An Odd Loop 85
Fig. 4-12 Parallel Branches 87
Fig. 4-13 Merged Branches 88
Fig. 4-14 Nested Parallel Branches 89
Fig. 4-15 Inferring Functions in Parallel Branches 90
Fig. 4-16 Nondeterministic Fork 92
Fig. 5-1 Matching Functions to Constants 97
Fig. 5-2 The MOVE Template Hierarchy 102
Fig. 5-3 Complications With Guard Generalizations. 106

Fig. 5-4 Abandoning a Condition because of Inconsistent Pairs 109
Fig. 5-5 Table of operators for the Robot Domain. 110
Fig. 5-6 The Expression Building Algorithm at Work. 115
Fig. 6-1 Include and Conflict relations. 120
Fig. 6-2 The Template Ordering for the 2D Robot Domain. 123
Fig. 6-3 Determining Conflicts between Condition Nodes 128

XIV

Chapter 1
Introduction

This thesis addresses a task and an issue. The task is to acquire procedures from examples. The issue is how to constrain generalization in a rich domain. The thesis claims that one should limit generalization by requiring sufficient justification before adopting a generalization. It describes an implemented system called NODDY, based upon this principle of requiring justification, that acquires procedures in a simple robot domain. NODDY also illustrates three new generalization heuristics for finding loops, finding complex relations and generalizing parameterized objects.

The acquisition of procedures is a variation of the more commonly studied task of acquiring concepts from examples. A teacher shows NODDY several examples of how to perform a certain task. NODDY must generalize these examples to come up with a procedure that will cope with a wider range of situations than those for which the teacher gave examples. A significant difference between procedure and concept acquisition arises from the specific constraints on the structure of procedures. NODDY exploits these constraints to guide the matching and generalizing in several ways. A second significant feature is that there are several different kinds of generalization involved in acquiring procedures, each requiring different generalization methods, which must be coordinated and combined.

The task of procedure acquisition has important applications for automation, industrial and otherwise. A major advantage that human workers have over robots and computers is that one can instruct them easily and flexibly, for instance, by demonstrating how to perform some task. Although humans are likely to remain more flexible than machines for a long time, robots and computers would become much more useful if they also were instructable in a flexible manner. There are several kinds of reasoning and generalization that an instructable robot would need to be able to do. NODDY is just a first step towards one part of such a robot.

The prime purpose of NODDY, however, is not to be an instructable robot, but rather to be a testbed for investigating the issues of generalization. Procedure acquisition is a particularly good domain for this because of the variety of kinds of generalization involved.

Generalization arises in a variety of learning and reasoning tasks. It is central to

any kind of acquisition from examples, where the concept, procedure, rule, *etc.*, to be acquired is a generalization of a set of examples. It is important to other learning tasks such as discovering new theories or constructing classification hierarchies which involve constructing general descriptions that explain or structure some data. But it is not confined to learning tasks—it is also essential for any task that involves any kind of matching, if partial matches are considered. If two descriptions match partially, then they have something in common, without being exactly the same. What they have in common is exactly a generalization of the two descriptions, so that finding a partial match is equivalent to finding a generalization. Since matching is necessary for such tasks as analogical reasoning, indexing, data retrieval, and rule based reasoning, these tasks also involve generalization, either implicitly or explicitly.

Generalization is difficult because it is inherently underconstrained—in any rich domain there are many possible descriptions that are valid generalizations of a set of examples. The central problem of generalization is how to choose the "best" generalization from among these candidates. Generalization can therefore be viewed as a search through a space of possible descriptions. From this viewpoint, the problem of generalization is how to structure the space, what order to search it in, and when to halt the search. This thesis will argue that a generalizer must be guided and limited, by considering the *justification* for its choices. It will also argue that the appropriate search order on a space of possible generalizations is based on the justifiability of the generalizations—how much data would be needed to justify the choice of a generalization.

Outline of Thesis

The thesis is organized as follows:

- **Chapter 1** presents a scenario showing informally how NODDY acquires a procedure from examples in a 2-dimensional robot domain. Section 1.2 places NODDY within the spectrum of learning tasks by identifying the key dimensions along which learning tasks vary. Section 1.3 discusses the central problems of generalization and compares the two main approaches. The last section gives a brief review of the background of this thesis in the literature.
- **Chapter 2** defines the procedure acquisition task more precisely and describes NODDY's representation of procedures. Section 2.5 briefly defines the 2-dimensional robot do-

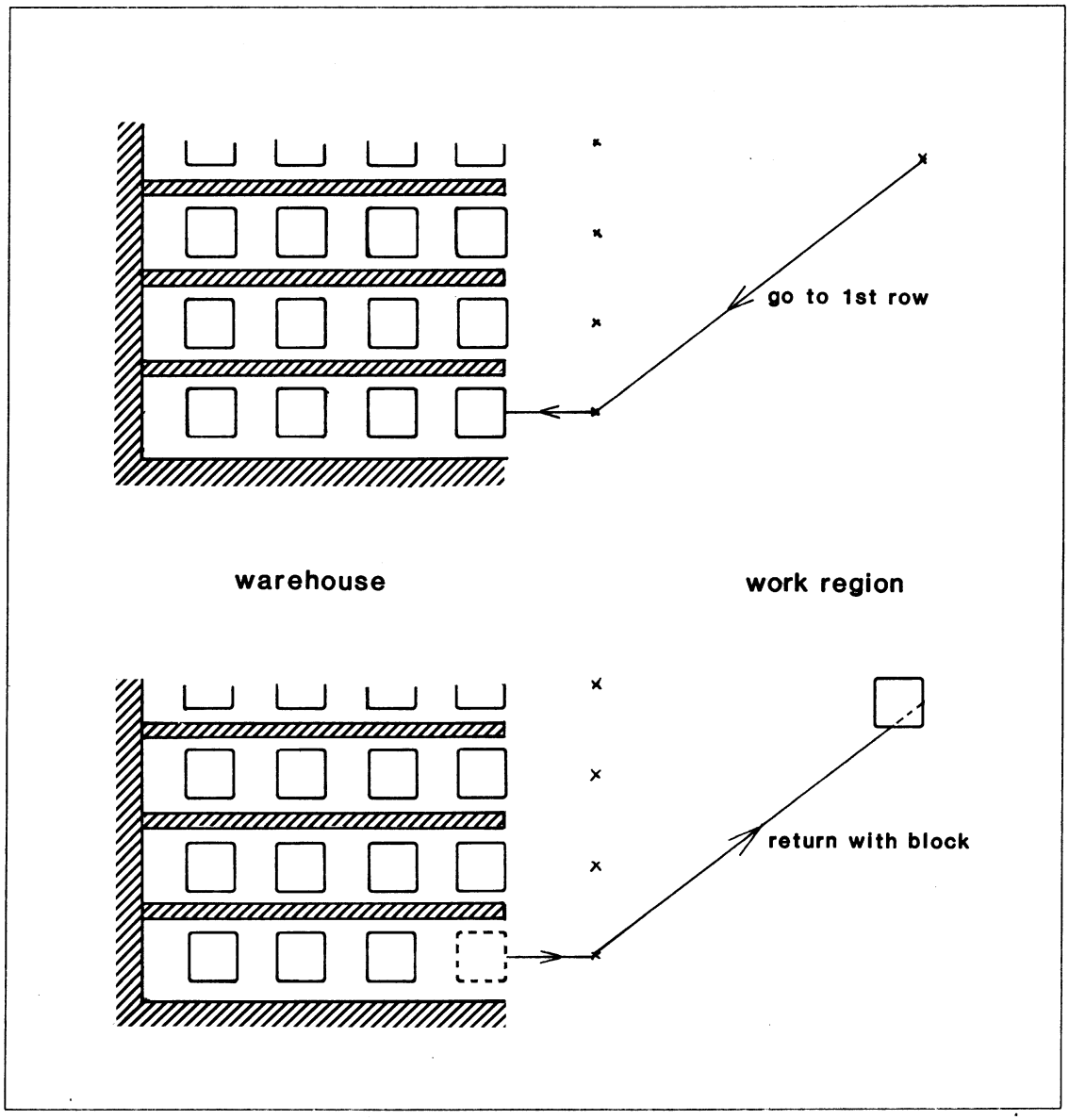
main from which all the examples in the thesis are taken.

- **Chapter 3** gives an overview of NODDY, characterizing the kind of generalization that NODDY does, and using one of the procedures from the scenario to step quickly through the several stages by which NODDY generalizes a procedure to account for a new example.
- **Chapters 4 - 6** present the workings of NODDY in more detail. Chapter 4 describes the matching and generalization of the structure of procedures. Chapters 5 and 6 describe the matching and generalization of the actions and conditions of procedures respectively.
- **Chapter 7** discusses the principles arising from NODDY: the role of domain constraints in constraining generalization, three constructive generalization methods, and the principle of justified generalization.
- **Chapter 8** concludes the thesis with a summary of what has been accomplished and a discussion of the directions for future work in procedure acquisition.

1.1 Scenario

The following scenario describes how NODDY acquires a simple procedure for a 2-dimensional robot domain from a sequence of examples presented by a teacher. Although the example is described somewhat informally, this is an example on which NODDY has been run, and NODDY really does acquire the procedure in the way the scenario describes.

The teacher wants NODDY to acquire a simple procedure, called RETRIEVE, for retrieving a block from a warehouse area. Figure 1-1 shows a warehouse area with blocks arranged in rows of four, and a wall at the bottom and left of the blocks. The procedure RETRIEVE (see figure 1-2) will take the robot from anywhere in the "work area" to the "entry point" of the first row of the warehouse (step 1), along the first row (step 2) until it hits something. If it has hit a block, (as in the example of figure 1-3 *a*), the robot will grasp the block (step 4), move back to the entry point of the row (step 4), then back to the starting position (step 5), where it will let go of the block (step 6) and stop. If, on the other hand, after step 2, the robot has hit the wall rather than a block (as in figure 1-3 *c*, *d*, or *e*), then the procedure will take the robot back to the entry point of the current row (step 3′), up to the entry point of the next row (step 4′), then repeat from step 2 on the

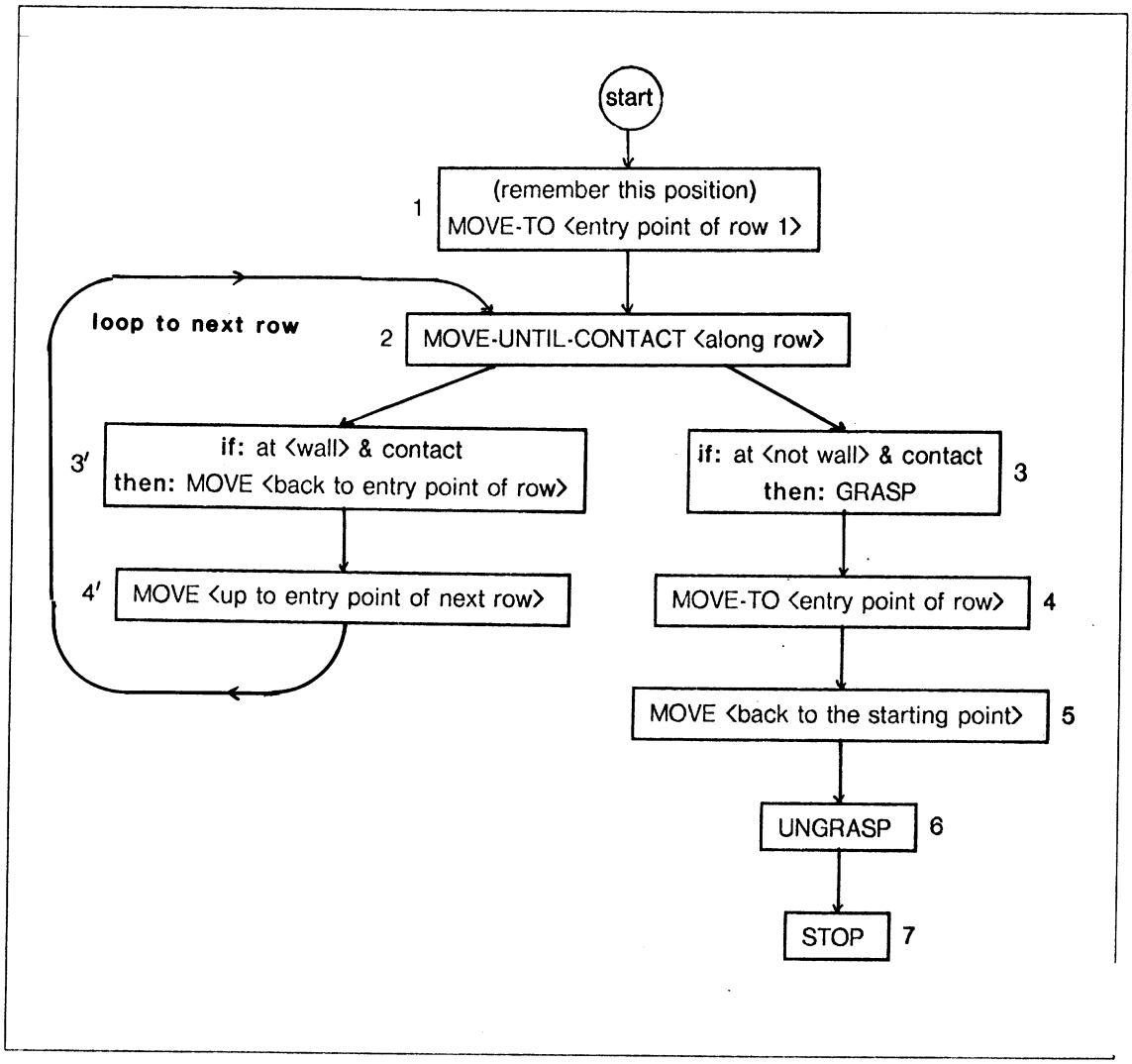


-- -- Figure 1-1 The Warehouse Setup for the RETRIEVE procedure. — —

The robot starts in the work area on the right, then moves to the warehouse and searches the rows, in order. It takes the first block it finds back to where it started.

new row, until the robot finally finds a row with a block. The phrases enclosed in <...> in the figure are informal descriptions of what are actually numbers and expressions. The rest of the figures in the scenario will use these phrases rather than confuse the issues with representation details that belong in chapter 2. The descriptions will be made slightly more precise towards the end of the scenario.

NODDY acquires the RETRIEVE procedure from a sequence of five examples. Figure

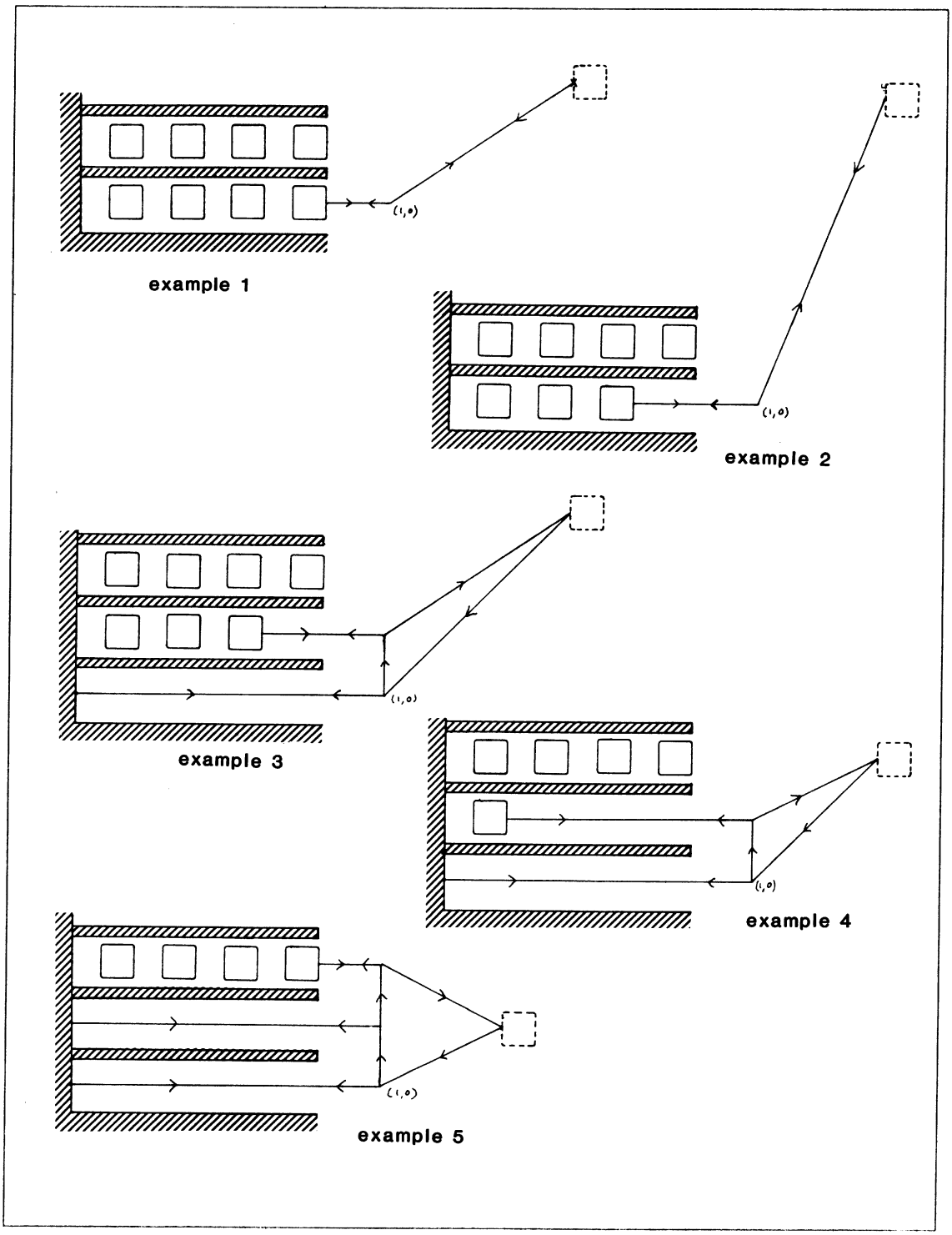


— — Figure 1-2 The Goal Procedure: RETRIEVE. — —

NODDY must acquire this procedure from examples given by the teacher.

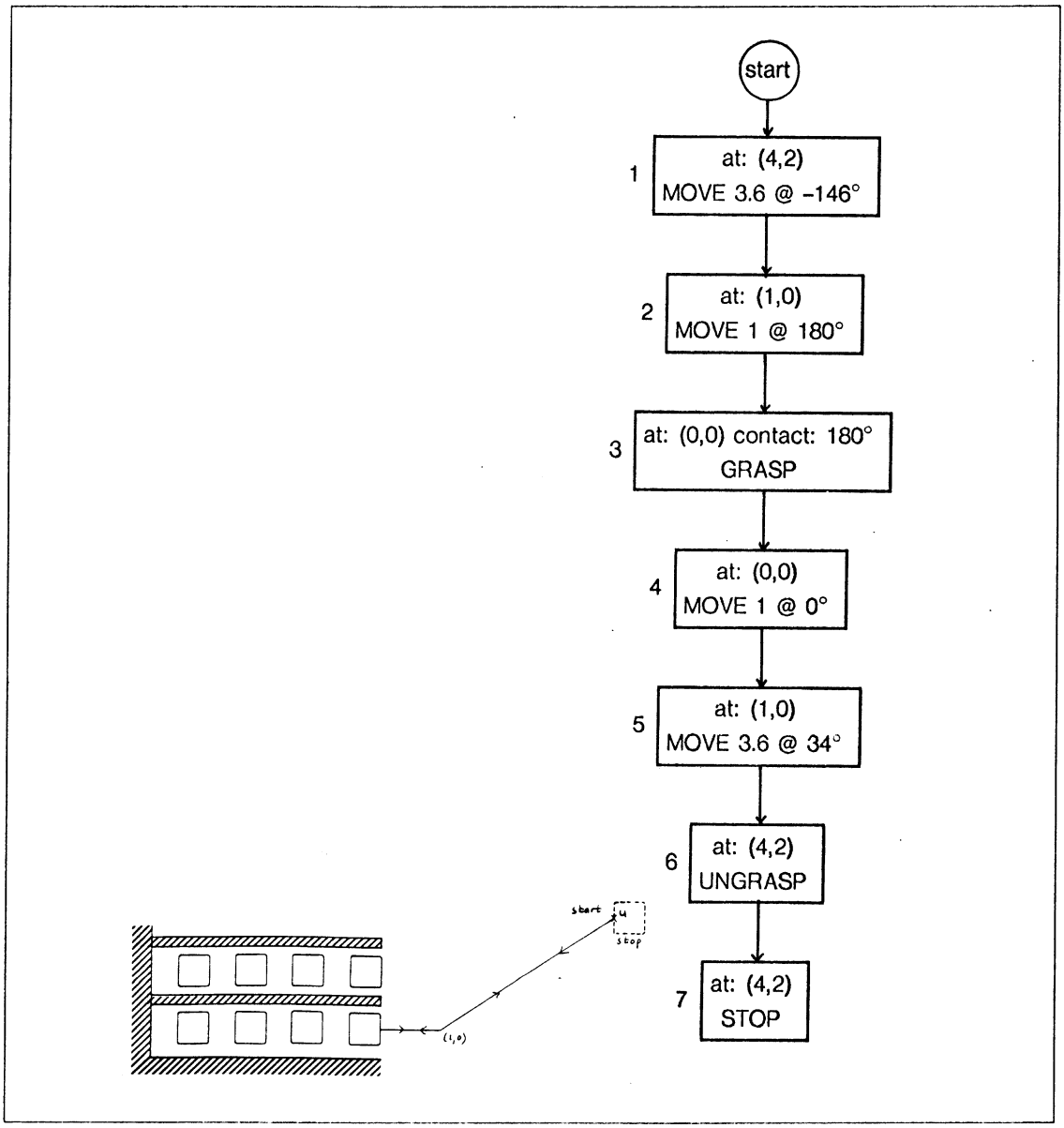
1-3 is a graphical representation of the examples, showing the path that the teacher led the robot through to generate each example. The description of the example actually presented to NODDY is more like that of figure 1-4 — a sequence of patterns (position and force feedback) and actions (MOVE, specified by the distance and direction to move, GRASP, or UNGRASP).

There are a number of features in the RETRIEVE procedure that are just not present in the examples given to NODDY. Therefore, to generalize the examples to obtain RETRIEVE, NODDY cannot just extract components of the examples but must infer these features. For instance, RETRIEVE has a loop and a fork. All the examples are



— — Figure 1-3 The examples presented by the teacher. — —

This is a graphical representation of the sequence of steps the teacher leads the robot through to generate the examples. The actual example given to NODDY is the sequence of actions performed and the feedback pattern at each step.



— — Figure 1-4 The First Trace. — —

This trace is just example (a) of figure 1-3 expressed in a form closer to NODDY's representation of it. The details of the representation will be given in chapter 2, and are not necessary for understanding this section. This is also NODDY's first version of the RETRIEVE procedure since NODDY does not generalize from a single example.

straight line sequences, so NODDY must infer the presence of the loop and the fork itself. The conditions in step 3 and 3′ (which are predicates on positions) are the branching conditions for the fork at step 2. These conditions must be inferred from the particular positions in the examples.

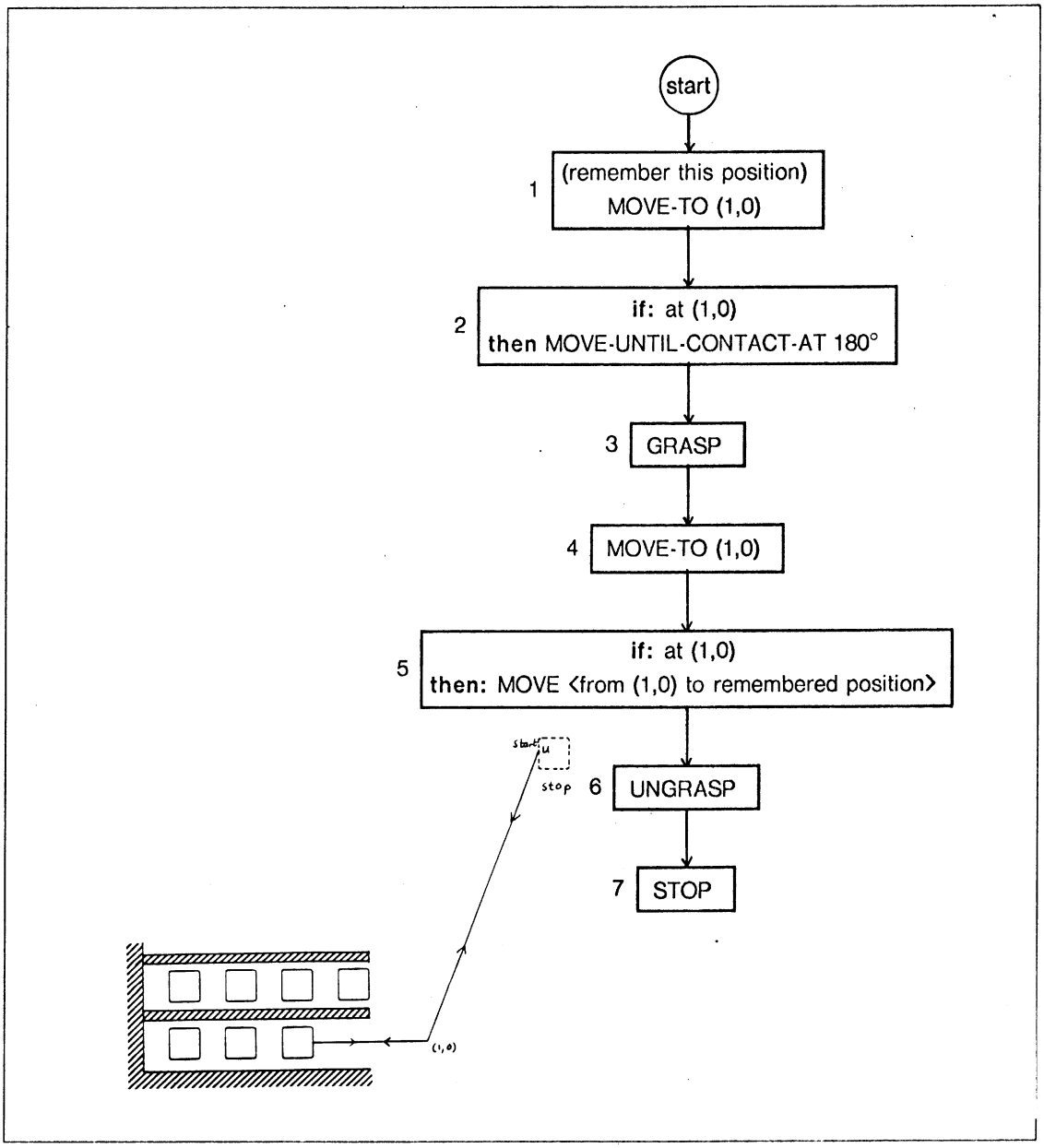
Several of the actions are also generalizations of the actions in the examples. For instance, step 1 has an absolute `MOVE-TO` action, *i.e.*, move to a particular point. This must be generalized from the relative `MOVE`s of the examples—move by a vector relative to the current position. Step 2 has a `MOVE-UNTIL-CONTACT-AT` action, which involves a new stopping condition. The actions of steps 4 and 5 are even more complex, though the complexity is hidden in the descriptions of the parameters. The `MOVE-TO` of step 4 involves moving to a point, but which point depends on which row you are in, and step 5 moves to the initial position at the start of the procedure. In both cases, the parameters of the actions must be expressed as a function of some position, and NODDY must not only determine what the function is, but must also determine which position it is a function of. None of this information is present in the examples.

NODDY does not acquire RETRIEVE all at once. The teacher presents the examples one at a time, and NODDY uses each new example to generalize its current version of the procedure to incorporate the new example.

The first example shows NODDY how to get the first block from the first row. NODDY's first version of RETRIEVE is identical to the first example (see figure 1-4) since NODDY has no basis for any generalization. RETRIEVE-1 is only applicable when the warehouse is completely full and the first block of the first row is present.

In the second example (figure 1-3 b), the teacher shows the robot how to get the second block from the first row. The teacher also starts the robot in a different place. Using this example, NODDY generalizes RETRIEVE-1 to obtain RETRIEVE-2 (figure 1-5). Neither of the first two examples involve the second row, so RETRIEVE-2 does not include the loop of RETRIEVE, but NODDY has generalized the first two actions to `MOVE-TO` $(1, 0)$, (the entry point for the first row), and `MOVE-UNTIL-CONTACT-AT` 180°, respectively. NODDY also infers that step 4 involves moving back to the entry point of the first row and step 5 moves back to the starting position. Because this last action is a function of the position in step 1, NODDY notes in that step that the position should be remembered for later reference. This position is a *state variable* of the procedure. The robot does not need to remember any other other position and force-feedback information.

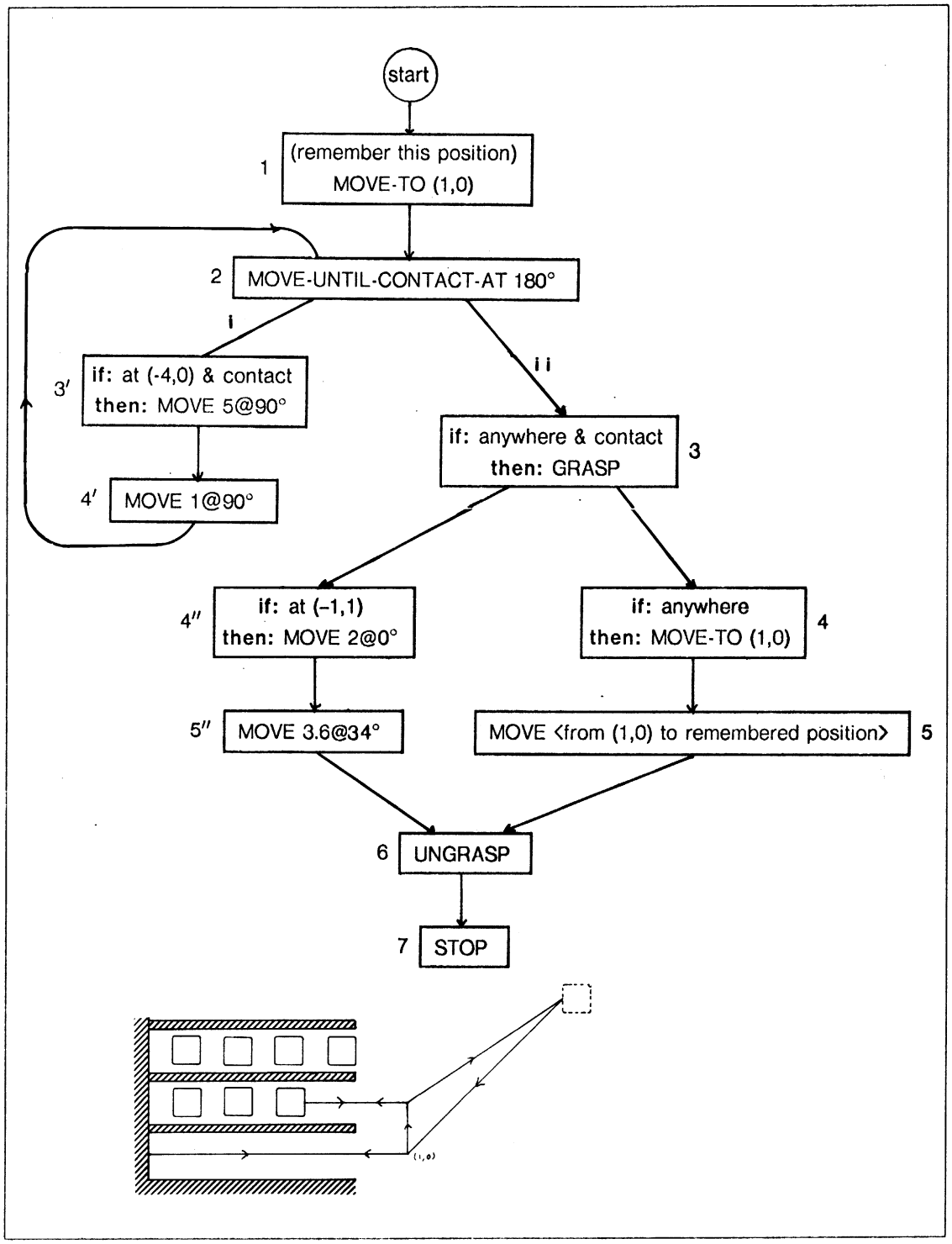
The third example introduces the second row. Given this example, NODDY generates RETRIEVE-3 shown in figure 1-6. This procedure is actually more complex than the goal procedure, RETRIEVE, since it not only has a loop for moving up to the second row, but has an extra branch—steps 4″ and 5″. RETRIEVE-3 has one branch for taking the



— — Figure 1-5 RETRIEVE-2 — —

NODDY's version of the procedure after two examples. It will work for any block in the first row and any starting position. NODDY has inferred that it should remember the starting position and that the MOVE of event 5 is a function of the remembered position. NODDY has also inferred that the MOVE of event 2 is a `MOVE-UNTIL-CONTACT-AT`.

block out of the **first** row and returning to the starting position (steps 4 and 5, taken from RETRIEVE-2) and another branch for taking the block out of the **second** row and returning to the starting position (steps 4″ and 5″, taken from the third example). NODDY has not seen enough about the second row yet, to be able to find the general way of

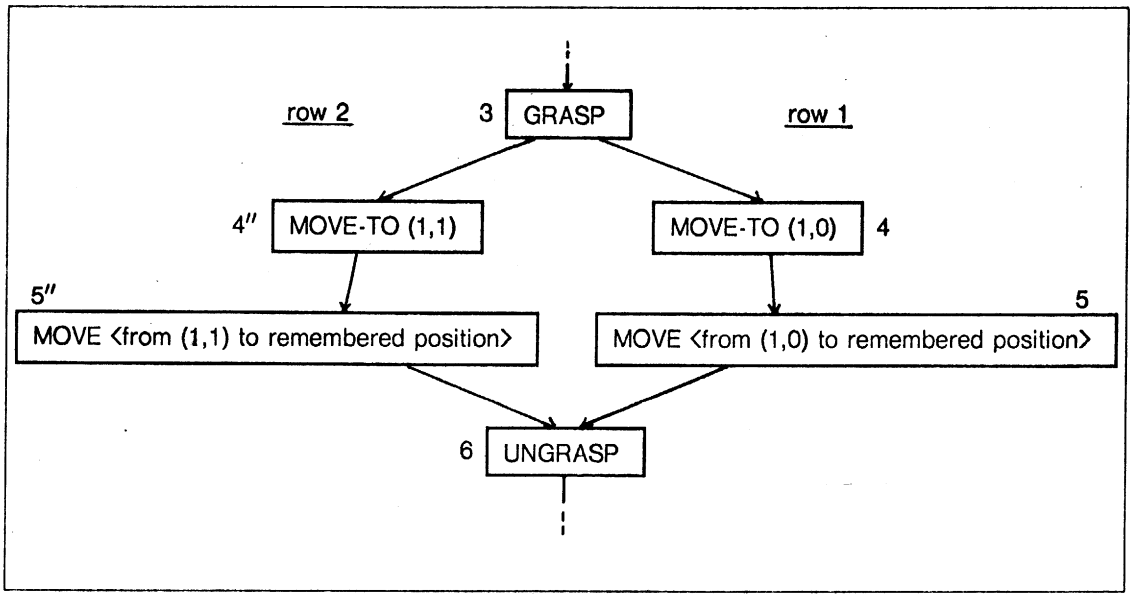


— — Figure 1-6 RETRIEVE-3 — —

NODDY has inferred a preliminary version of the loop to move up to the next row, but has not had enough examples to find the general way of returning from a row with the block and back to the start; hence the extra branch of events 4″ and 5″.

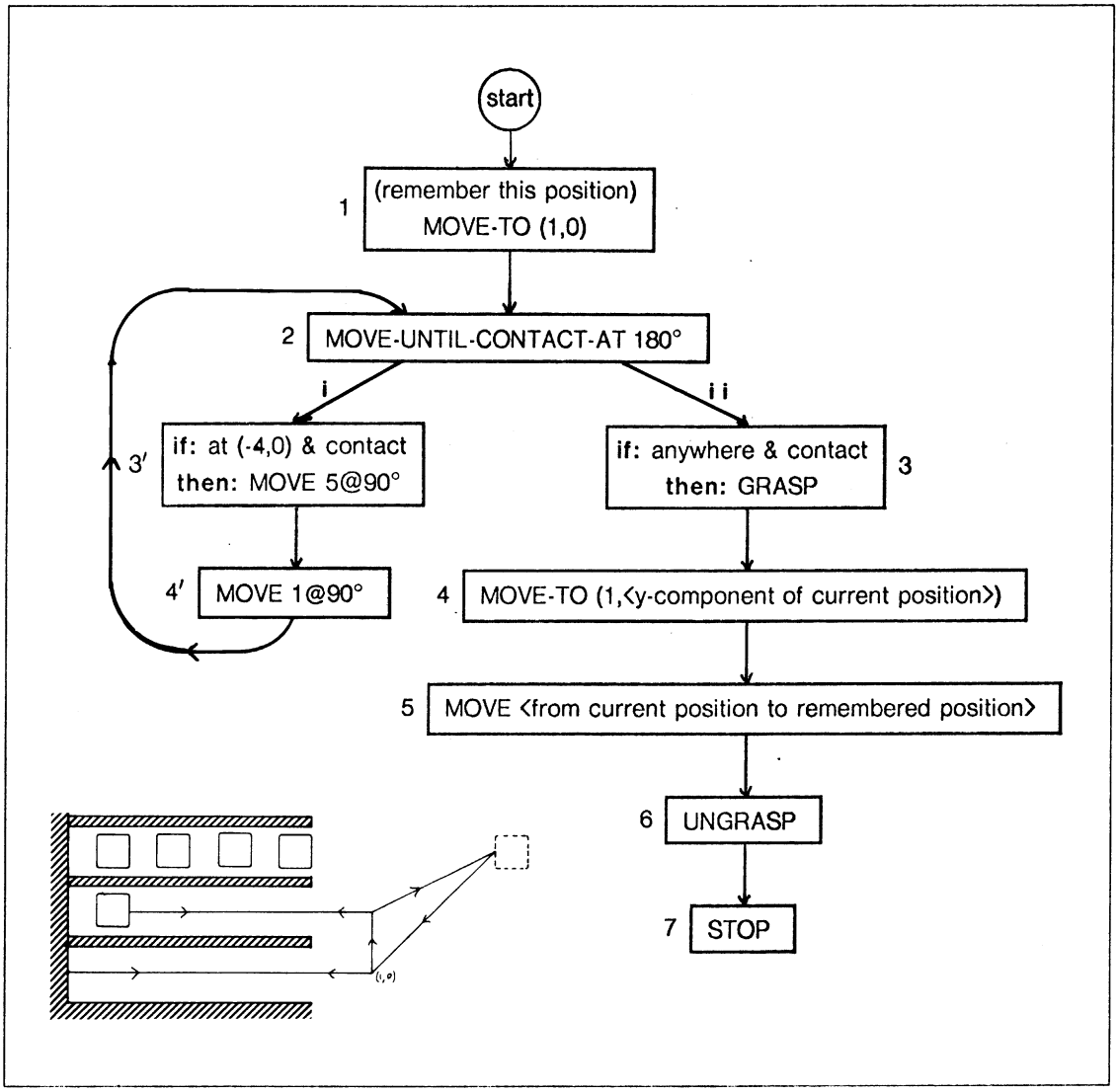
taking a block out of any row. This extra complexity will disappear with the next example when NODDY has been given a bit more information. However, NODDY has acquired a preliminary version of the loop which takes the robot up to the next row. It noticed that in the new example, the robot moved to the left two times, and each time ended up in contact with something. NODDY inferred that the procedure must loop back to the same `MOVE-UNTIL-CONTACT-AT` (step 2). The loop is still preliminary because the condition for taking the left fork after step 2 (*i.e.*, the condition of step 3′) is specific to the first row—being at the position $(-4, 0)$, (the wall is the column $x = -4$), since the example only traversed that part of the loop once.

NODDY also notes that the fork after step 2 must be executed carefully. The condition on step 3′ is a special case of the condition on step 3. NODDY therefore notes on RETRIEVE-3 that the robot must first test the applicability of the more specific condition (step 3′) and only when it is not satisfied, test the condition of step 3. NODDY notes the same thing for steps 4″ and 4. In figure 1-6, the order in which the conditions should be tested is noted on the arrows leading to the conditions.



— — Figure 1-7 A fragment of the intermediate stage in forming RETRIEVE-4 — —
NODDY has started to generalize RETRIEVE-3 to incorporate example 4, and has generalized the left branch of the "return from row" part of the procedure so that it can cope with a block anywhere in row 2.

The fourth example (figure 1-3 d) is another example of finding a block in the second



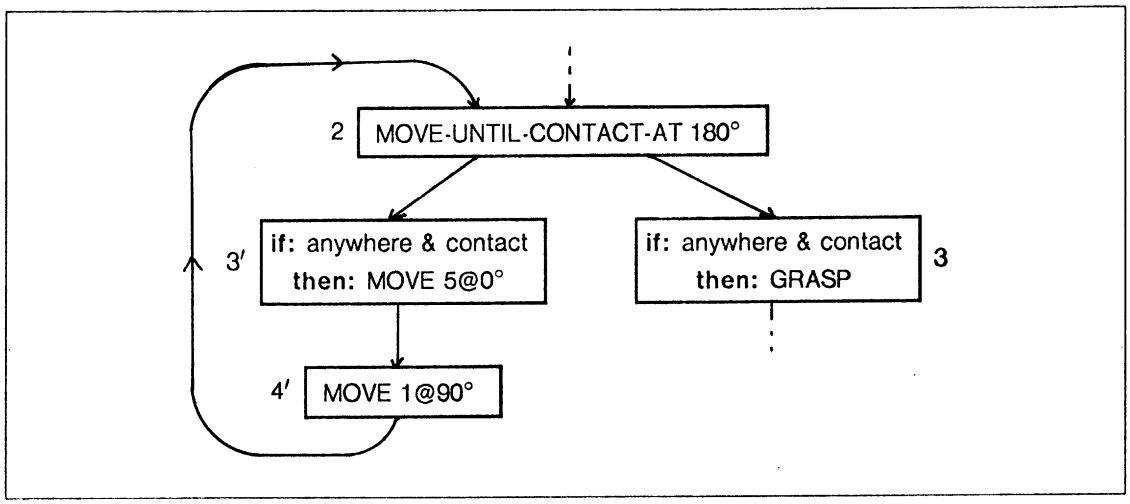
— — Figure 1-8 RETRIEVE-4 — —

NODDY can now grasp a block and return to the starting position from any row. In steps 4 and 5, NODDY has found the way the actions depend on the current position. The only deficit is the overly specific condition for entering the loop.

row. This example follows RETRIEVE-3 exactly until step 4″, so that no change needs to be made to the first part of RETRIEVE-3. Taking the block back to the entry point and back to the (different) starting position is slightly different, however. NODDY is able to use this new example to acquire a general way of returning with a block from the second row (generalizing steps 4″ and 5″ of RETRIEVE-3). Figure 1-7 shows the relevant fragment of the intermediate procedure that NODDY creates with this generalization. NODDY then infers the general way of returning with a block from any row (merging

steps 4″ and 5″ with steps 4 and 5). Figure 1-8 shows the new procedure RETRIEVE-4 after merging the two branches in the intermediate procedure of figure 1-7. The loop is still specific to moving up to the second row because the condition for entering step 3′ has not changed from RETRIEVE-3.

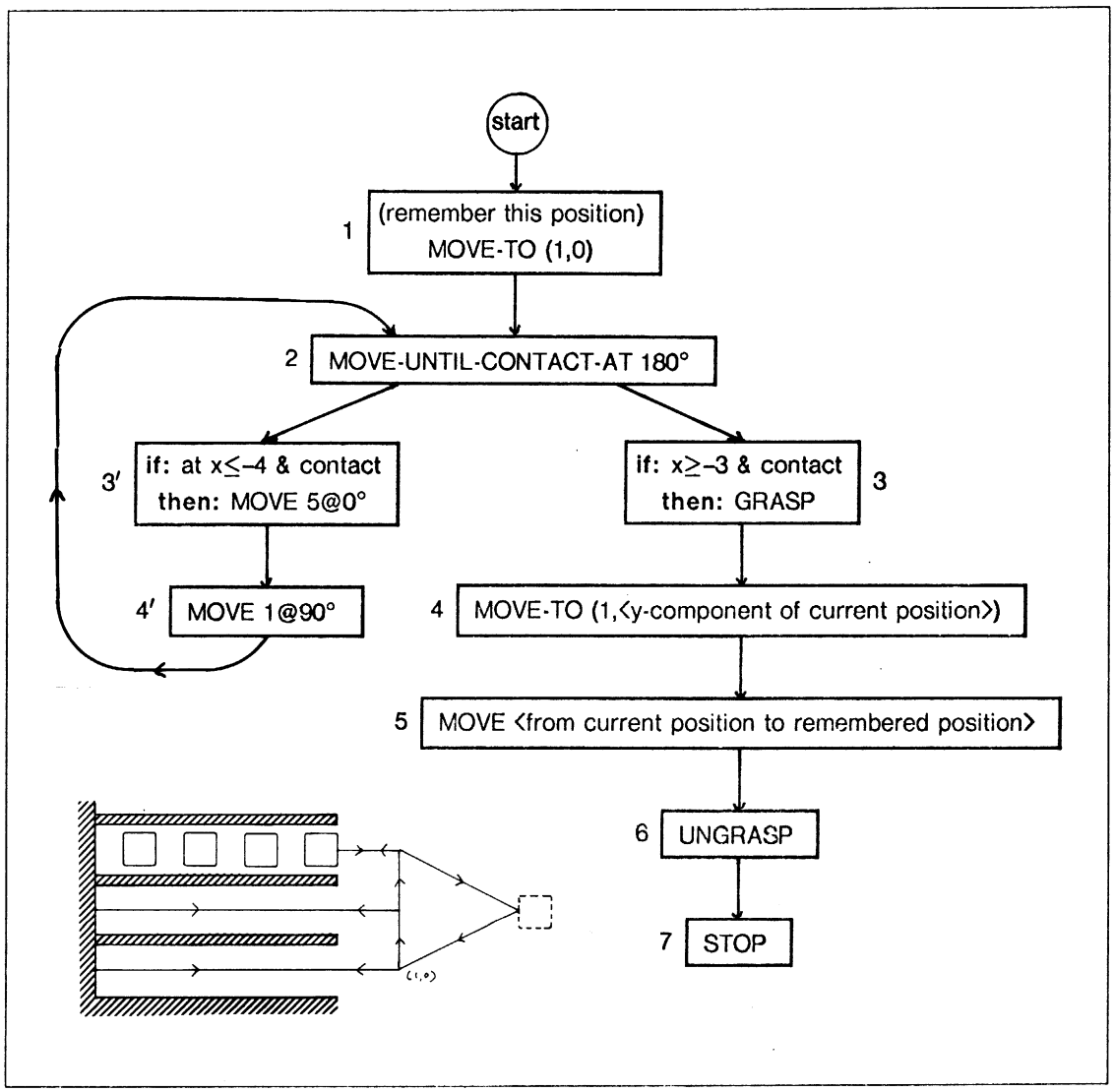
However, the steps for grasping a block, moving to the entry point of the row, and returning to the starting position will apply to any row, not just rows 1 and 2, even though those are the only rows that NODDY has seen.



— — Figure 1-9 A fragment of the intermediate stage in forming RETRIEVE-5 — —

The problem that NODDY notices is that the conditions on steps 3 and 3′ are now overly general, and NODDY must find conditions that will distinguish the two branches.

The final example (figure 1-3 e), involves moving from the first row to the second row, and then from the second row to the third row. NODDY recognizes that steps 2 – 8 of the example go round the loop of RETRIEVE-4 twice, and uses the two iterations to generalize the loop. First, NODDY generalizes the conditions on steps 3′ and 4′ without considering the rest of the procedure. The relevant fragment of the resulting procedure is shown in figure 1-9. There is a problem because the conditions of steps 3 and 3′ are identical. Therefore, after step 2, the robot could never tell whether to take the right branch and GRASP or the left branch and move to the next row. NODDY notices this indeterminacy, realizes that it has overgeneralized the conditions and searches for alternative conditions that distinguish the two branches. Figure 1-10 shows the final procedure, with these distinguishing conditions. This procedure is identical to the goal procedure of figure 1-2 (except that some of the informal descriptions of the actions and conditions are spelled



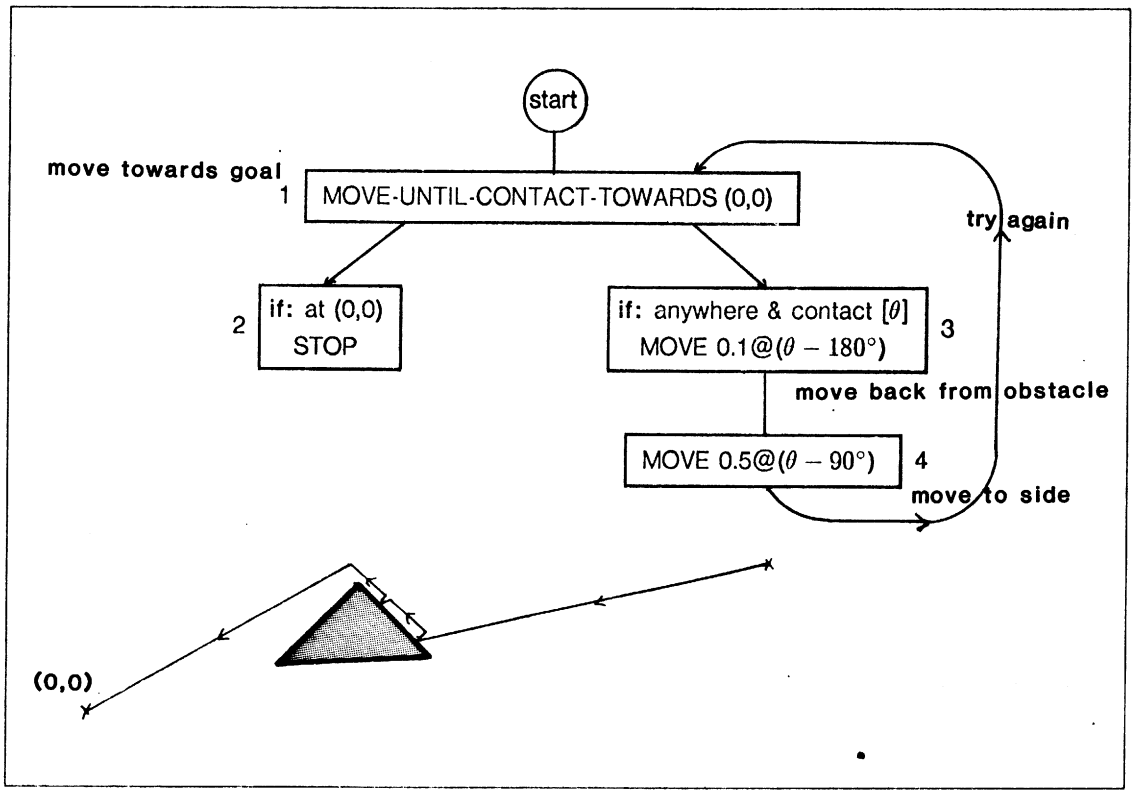
— — Figure 1-10 RETRIEVE-5 — —

The final procedure, which is the same as the goal procedure. From the five examples, NODDY has learned how to find the first block, whatever row it is in, and take it back to the starting point.

out in more detail). This is what the teacher was intending to teach, so NODDY has successfully acquired the procedure, with its loop, branch conditions, generalized actions, and internal state variable, from the five examples presented by the teacher.

1.1.1 Another Example: The TURTLE Procedure

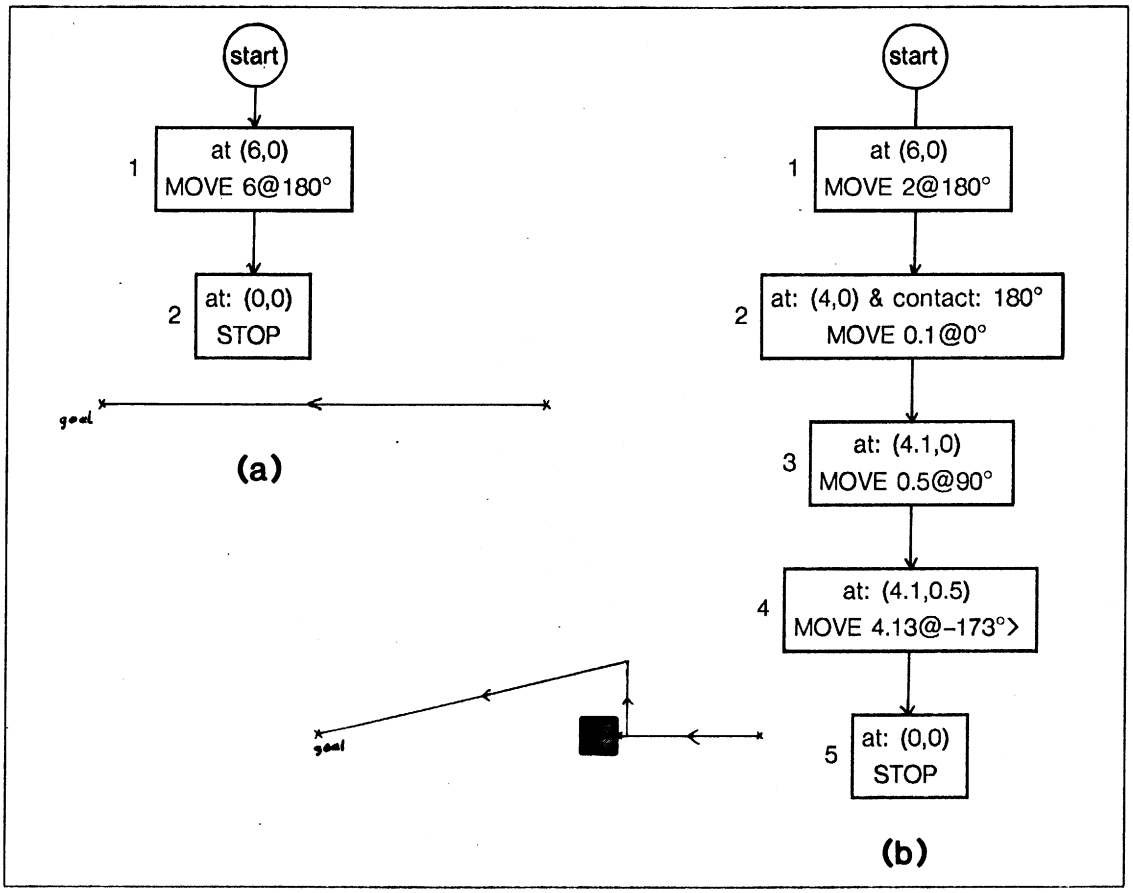
The following chapters use the RETRIEVE procedure described above to illustrate NODDY's operation. It will be useful to have an additional example to illustrate certain features of NODDY, so this section briefly describes the TURTLE procedure and the traces from which NODDY can acquire it.



— — Figure 1-11 The TURTLE Procedure — —

The TURTLE procedure instructs the robot to move towards the goal until it is either at the goal or has hit an obstacle. If it hits an obstacle, the robot backs off, moves to the side, and tries again.

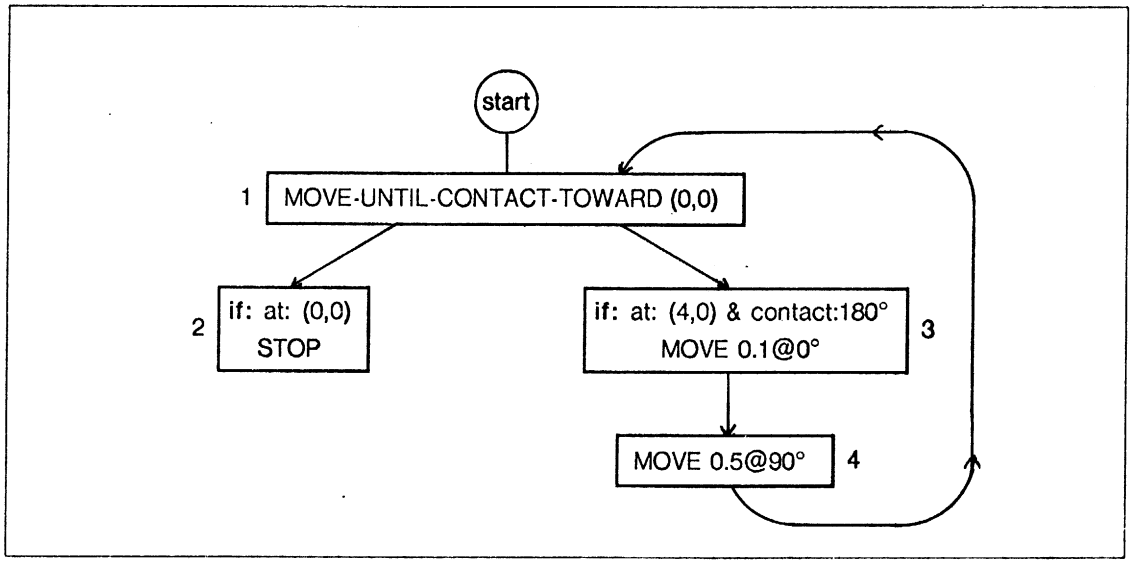
The TURTLE procedure, shown in figure 1-11, is a very simple procedure for getting to a goal position (0,0), and circumnavigating and obstacles along the way. The robot moves toward the goal (event $E1$) until it is either at the goal ($E2$) or in contact with some obstacle ($E3$). If it has hit an obstacle, it moves back perpendicular to the obstacle a step of 0.1 ($E3$); moves parallel to the obstacle a step of 0.5 ($E4$), then heads for the goal again (loops back to $E1$). It will repeat this as many times as necessary to get around the obstacle, and can cope with multiple obstacles, and obstacles with multiple faces.



— — Figure 1-12 The First Two Traces of TURTLE Procedure — —

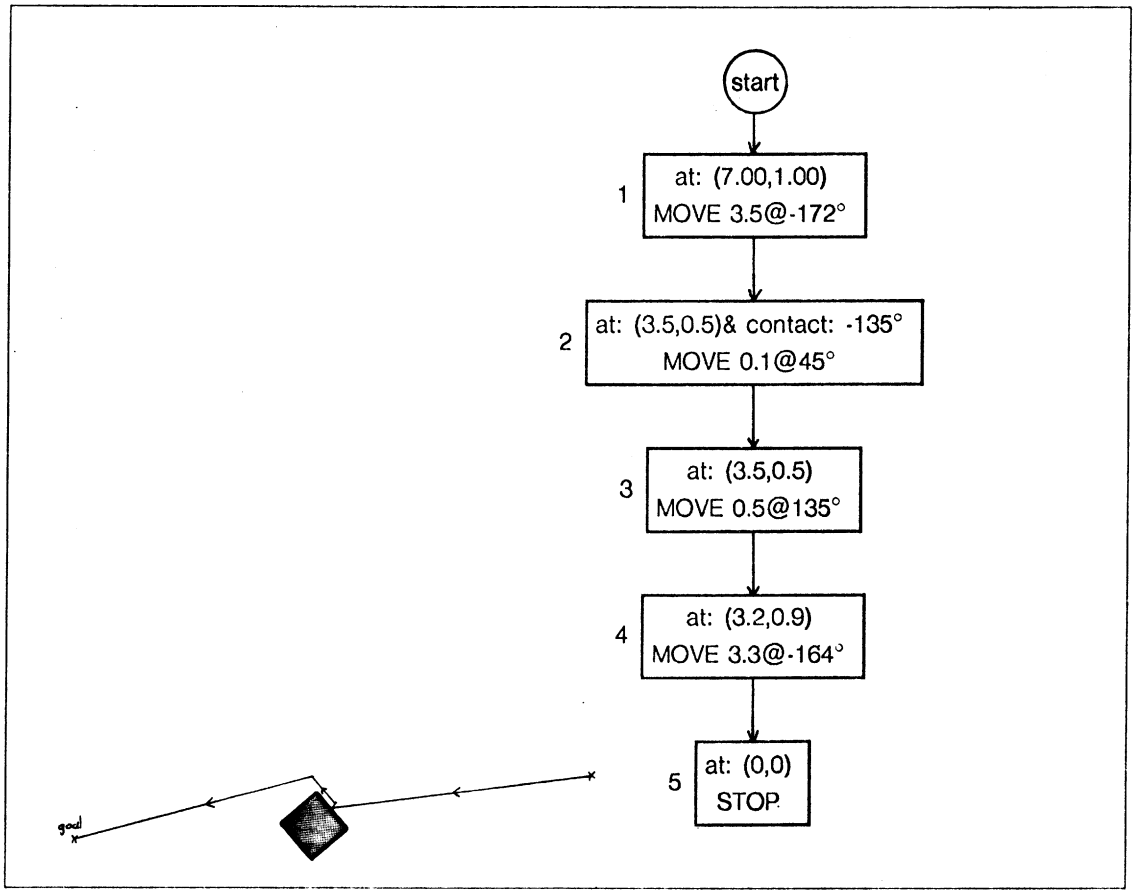
(a) When there is no obstacle, the robot goes straight to the goal position. (b) When there is a small obstacle oriented vertically, the robot hits the obstacle, backs off horizontally, moves along the side of the obstacle, then heads for the goal again

NODDY can acquire this procedure from just three traces. The first trace (figure 1-12-a) has no obstacle, so the robot goes straight to the goal. The second trace (figure 1-12-b) has one small obstacle. From these two traces, NODDY acquires the procedure in figure 1-13. NODDY has inferred that the first move should move towards the goal, whatever the current position. The loop in this procedure enables the robot to step around any object with a vertical face, no matter how many steps are needed. The third trace has an obstacle oriented at 45°, and from it, NODDY is able to generalize the actions in the loop to back away from the obstacle and step along the face, no matter what its orientation. This is the complete TURTLE procedure.



— — Figure 1-13 The Version of TURTLE After Two Traces — —

NODDY has inferred the loop, and can now handle any number of any sized obstacles, as long as the face that the robot hits is vertical. For example, it could handle the case shown in the bottom left of the figure.



— — Figure 1-14 The Third Trace for Acquiring TURTLE — —

With a small obstacle oriented at 45°, the robot does the same as in the second trace, except that it backs off and moves to the side in a different direction. With this third example, NODDY is able to generalize the version of TURTLE in figure 1-13 to the fully general procedure of figure .

1.2 Learning Tasks

"Learning" is a very vague term, and there many different problems and tasks that could be classified as learning tasks. This section places NODDY within the broad area of learning tasks by identifying a number of the dimensions along which such tasks vary and giving the NODDY's position along each dimension.

A learning task can be characterized in terms of the given input and the desired output—what the learner must learn from, and what the learner must acquire as a result. The following two subsections address the nature of the output and input respectively.

1.2.1 What the Learner Must Learn

Parameters, Database Entries, Concepts:

The first dimension is the nature of what the learner must acquire. Early work in machine learning, such as Samuel's Checker player [Samuel 1959] and perceptrons [Minsky and Papert 1969], was concerned with acquiring the values of parameters of a predetermined structure, whether a set of rules or a vector of features. The major limitation of this approach was that the weights were much less significant than the choice of rules or features, so that the more interesting problem of acquiring the rules or features themselves was not addressed.

A significant part of human learning is the acquisition of new facts that can then be retrieved later. No-one is willing to say that a computer that simply stores entered facts into a data base is really doing interesting learning, though there are some significant machine learning problems involved with having a machine construct its own indexing and retrieval system based on the facts that have been presented to it.

NODDY is concerned with the third class of learning tasks—acquiring concepts, where "concept" is used in the sense of a structured description that involves parts, properties and/or relations expressing some internal structure. Most of the AI machine learning work in the last 15 years has been concerned with acquiring concepts. A key issue in acquiring concepts is the representation language in which the concepts are expressed, since this contains implicit or explicit constraints on what concepts could be acquired. A later paragraph will distinguish several important subclasses of concepts on the basis of the constraints on the representation language.

Concepts, Rules, Procedures, Plans:

There are a variety of kinds of structured descriptions that a learning machine could acquire. All of them can be viewed simply as descriptions in terms of nodes with properties and relations between the nodes, but each subclass generally has certain constraints on what is a valid description which should be exploited by the learner. Concepts, as used in the general sense of unspecific structured descriptions, have very few constraints, and do not assume any special properties of any of the relations. The representation for rules, on the other hand, has a special relation between the set of preconditions and the set of consequents of a rule. A rule acquirer should exploit this special relation, and any other particular properties of rules.

Procedures, which NODDY acquires, are even more constrained than rules because of their control structure which imposes a dominant sequential structure on the components of the procedure.

Groups and Relations:

An important characterization of structured descriptions is whether the descriptions the learner acquires involve explicit groups and/or relations that are not present in whatever the learner learns from. Inducing groups is a particularly difficult induction task because the boundaries of the group are "invisible objects" that the learner must create out of nothing rather than select from the data. Inducing relations is similarly difficult because the learner must not only discover a relation, but also determine which nodes might be related.

The procedures that NODDY acquires have both groups and relations. An iterative loop, which describes many iterations by a single "typical iteration" and an exit condition, is a special kind of group. Procedures may also have internal state, represented, for example, by variables, and actions that depend on the state. This dependency is a relation between the variable and the action that the learner must acquire. In NODDY's case, there are no variables present in the data, and NODDY must find both the variable and the dependency.

Numerical Parameters *versus* Nominals:

A class of structured descriptions can also be characterized by the nature of the components out of which the description is constructed. The components may be parameterized objects, objects from a hierarchically structured set, objects from an unstructured set, or even

structured descriptions themselves. The nature of the generalization required to acquire the goal descriptions from the examples depends upon the type of the components.

The procedures that NODDY deals with are structured descriptions whose components are parameterized objects. Without parameterized components, several key features of procedures (generalized actions and dependencies between actions and patterns) become impossible, and procedure acquisition becomes relatively trivial. This is another interesting feature of procedure acquisition because there has been less work done on generalization of this type of description than on descriptions of the other kinds.

Single Theory *versus* Multiple Theories:

Another characterization of the output of a learner is whether it should produce a single theory or a set of alternative theories (where a theory is a particular structured description which accounts for the data). Mitchell's version space method [Mitchell, 1982, 1983] is specifically designed to produce the set of all theories that are consistent with the data seen so far, represented in the most concise manner possible. Winston's arch learner, on the other hand, kept only one description of the theory being acquired.

There are good reasons for either choice. A basic design decision underlying NODDY is to insist on a single theory (the description of a procedure, in NODDY's case) that accounts for the data, and wait for more or better examples if there is not a single best theory. There are two motivations behind this design decision. The first is that if the representation language for the theories is rich, there may be very many alternative theories that satisfy the data and it will be computationally expensive, both in terms of search and storage space, to deal with all the alternatives. The second motivation is one of the generalization principles outlined in the next section and addressed more fully in chapter 7, namely, that several alternative theories are no theory at all. NODDY works on the assumption that if there is not one unambiguous theory to explain the examples, then it has not seen sufficient examples to uncover the underlying regularities, and it should wait for more examples before adopting a theory.

1.2.2 What the Learner Learns From

The input to the learner includes both the knowledge implicit or explicit in the learner before it even starts to learn anything, and the data from which it explicitly learns. We must consider the nature of this data, and where it comes from.

Teacher Driven *versus* Self Driven:

The second dimension along which learning systems vary is the source and nature of the input data. The most important distinction is between those that assume a cooperative teacher is providing examples and those that generate their own examples in some way. The learner's task is much simpler if a teacher provides it with preclassified examples in a pedagogically good order. One of the powerful ideas in Winston's early concept acquisition program was that the teacher should provide "near misses" along with the positive examples—descriptions of structures that were almost, but not quite, examples of the concept being acquired. Michalski's CLUSTER algorithm [Michalski, 1980, 1983] must be given a set of descriptions, but it will find its own classification for them and a description for each class. Lenat's AM and Eurisko [Lenat, 1982a, 1982b] and Mitchell's LEX [Mitchell, 1983] generate their own examples, AM by applying modification heuristics to the descriptions it has already constructed, and LEX by using a very simple, but general, problem solver to solve problems, and analyzing the problem solver's solution. However, at least for learners such as LEX, generating the examples is quite separable from the problem of acquiring the concepts from those examples, and can legitimately be viewed more as a problem solving task than a learning task, even though the it is solving a problem in order to acquire new knowledge.

NODDY is currently teacher driven, because the purpose of NODDY is to investigate issues in generalization rather than problem solving. The teacher must hand NODDY a sequence of examples of the procedure. NODDY will learn much faster if the sequence introduces just a few new aspects of the procedure in each example, and NODDY may not be able to acquire the procedure at all if the order is very perverse.

However, procedure acquisition is particularly suited to automatic generation of examples, because running the procedure in a modified environment will produce a new trace, and a promising avenue for future work is to reduce the role of the teacher by enabling NODDY to generate its own examples. If NODDY were told the goal of the procedure it is trying to acquire, it could classify any traces it generated as positive or negative instances of the procedure and then use them to improve the procedure.

Instances, Situations, or Traces:

If the learner operates on examples of what it must learn (whether the examples are provided by a teacher or self generated), the relation between the examples and the concept to be learned will depend on the nature of the concept. For instance, an example

of a description of a table will be a description of a particular table, which will generally contain all the structure of the general table, but more specific values for the properties of the parts of the table. An example of a rule may be just a description of an entire situation in which the consequences of the rule are valid, which will include many irrelevant parts.

One possible example of a procedure would be an overly specific procedure—a procedure with all the structure of the procedure to be acquired (such as loops and forks) but with overly specific conditions and actions. The most natural example of a procedure, on the other hand, is a trace of the execution of the procedure in a particular situation. Traces are "natural" because a human teacher can easily generate a trace of a procedure by actually performing the task (or leading a robot through the task) and recording the steps he actually took. The teacher does not even need to have an explicit formulation of the procedure in order to generate a trace. Humans often teach procedures by "showing how to do it," rather than giving instructions.

A significant feature of traces as examples is that they do not have any of the interesting structure of the procedure—they have no branching, no loops, and no explicit dependencies. Acquiring procedures is therefore a particularly interesting acquisition task because the learner must construct all of these structural components of the procedure rather than just extracting them from the examples.

Currently, NODDY acquires procedures from from a sequence of traces presented by the teacher. NODDY will also accept procedures as examples—traces with additional structure such as loops—but it does not require them and all of the examples used in the rest of the thesis are traces without any additional structure.

Domain Specific Knowledge *versus* Domain Independent Knowledge:

The second part of a learner's input is the knowledge it uses to interpret and analyze the data it is given. This may be explicit in a data base, or implicit in the workings of the system. It may also be specific to the particular domain (for example, symbolic integration, or two dimensional geometry), or general heuristics that are domain independent (such as the "drop condition" heuristic of [Michalski 1983]). If the learner is dependent on domain specific knowledge, it will be more difficult to apply it to different domains, particularly if the domain specific knowledge is implicit in the learner's structure.

Knowledge about the structure of procedures and how they can be generalized is implicit in NODDY's algorithm. NODDY is therefore restricted to acquiring procedures

rather than acquiring rules, structural descriptions, or plans. Although the principles underlying NODDY could be applied to the acquisition of other kinds of concepts, NODDY itself could not. NODDY, has currently been applied to a two dimensional robot domain and uses some knowledge specific to this domain. However, one design principle for NODDY has been to keep this domain specific knowledge to a minimum and to make it explicit. As a consequence, one could use NODDY to acquire procedures in any domain by replacing the robot domain knowledge. Later chapters describe the form that this domain specific knowledge must take.

Incremental *versus* Non-incremental Acquisition:

One can also classify the input data to a learner that learns from examples on the basis of whether the examples are presented as a complete set or incrementally. If presented as a group, the learner can analyze all the examples at once and may also assume that no other examples are necessary. The learner is usually expected to produce the "best" theory of the data, where "best" is defined according to some optimality criteria.

In incremental acquisition, the learner is expected to produce a theory after each new example is presented, revising the theory when it is inconsistent with the next example. In this case, the learner is not expected to produce the best theory, but rather to evolve the theory "gracefully" to accommodate the new examples. VanLehn [1983] uses a modification of incremental acquisition in which the teacher presents the examples as a sequence of groups of examples. The teacher guarantees that the theory needs to be changed in only one way to accommodate all the examples in each group. This overcomes the ambiguities that may arise if the learner may only analyze one example each step.

NODDY is designed solely for incremental acquisition. The teacher presents the example traces one at a time, and NODDY modifies its current version of the procedure to incorporate the new example.

Positive Examples Only *versus* Positive Examples and Near Misses:

The final classification of the input data is determined by whether the data only includes positive examples of the concept/procedure/rule/*etc.*, or whether it also includes negative examples. One can show quite readily that in the general case, it is not possible to acquire concepts from positive data only (see [Gold 1967]), because there are too many possible theories consistent with the data and no way of distinguishing between them. However, if there are strong constraints on the class of theories under consideration, it may be possible

to acquire a theory without any negative examples (see [Angluin and Smith, 1982] for set of grammars that can be acquired from positive data only).

The procedures that NODDY acquires do have a strong constraint—that they be deterministic. This constraint is sufficiently strong that NODDY is able to acquire many procedures from traces of valid executions of the goal procedure (*i.e.*, positive examples only). It turns out that the determinacy constraint does not place any constraints directly on the conditional tests in the procedure, and therefore the conditions cannot be acquired without negative examples. However, NODDY can exploit the determinacy constraint to generate positive and negative examples of the conditions from the traces (which are positive examples of the procedure as a whole).

1.3 Generalization

The task of a generalizer is to take a set of example descriptions in some representation language L_E, and construct a description (or set of descriptions) in a representation language L_G that is a *generalization* of the examples. A description G is a generalization of another description S if G *covers* S—G describes everything that S describes and more. If the set of examples includes both positive and negative examples, then the generalizer must construct a description that covers all the positive examples but excludes all the negative examples.

We can view generalization as a search through the space of all possible descriptions in the language L_G for a description or set of descriptions that account for the examples([Mitchell 1982]). Clearly, the more expressive L_G is, the larger the space that must be searched, and the more difficult the generalization task. It is very desirable, therefore, to constrain or *bias* L_G to make the search space is as small as possible without eliminating generalizations that may be needed to explain some sets of examples [Mitchell 1983]. The constraints on the task domain provide a basis for this bias.

The two defining characteristics of a generalizer are the *order* in which it searches the space, and the *stopping criteria* for either deciding on a generalization or abandoning the search. One can classify generalizers into two broad classes according to their criteria for ordering and halting the search. Both approaches are based on a version of the principle of least commitment—make as conservative a choice as possible so that you will not need to retract it later. The first approach is that of *least generality*—order the space according to generality, and stop the search at the most specific description that is a generalization

of the examples. This is most appropriate for a data driven search using generalization heuristics to construct more general descriptions out of more specific descriptions. The second approach is that of *least complexity*—order the space according to a complexity measure and stop the search at the simplest description that is a generalization of the examples. This approach is particularly appropriate for a top-down search through a predetermined ordering on the descriptions. The rest of this section will describe these two approaches, discuss the disadvantages of each, and present the unifying approach upon which NODDY is based.

Least Generality

The simplest form of the least generality approach is to choose the most specific candidate that satisfies the examples when there is just one maximally specific candidate, and wait for better examples when there are several maximally specific candidates (*e.g.*, [Winston, 1970]). This method works only when L_G can be constrained so as to ensure that there can be just one maximally specific generalization, which is not the case in many domains.

A more sophisticated form of the least generality approach is to keep the complete set of maximally specific candidates instead of insisting on a single candidate, since choosing one over the others would mean making more commitment than necessary. If there are negative as well as positive examples, then the least generality approach would also keep the set of maximally general generalizations that exclude the negative examples (the least commitment to what the concept is *not*). The two sets together constitute a *version space* ([Mitchell, 1982]), which is a concise representation of the complete set of all the generalizations consistent with the examples—any generalization that is a generalization of one of the maximally specific generalizations and a specialization of all of the maximally general generalizations is consistent with all the examples. The two sets effectively define a region of generalization space by specifying the lower and upper bounds on the region. A version space represents the least possible commitment since it covers all possible generalizations that are consistent with the examples. With sufficient examples, the version space will narrow down to just a single generalization, as long as the space is not too dense.

The least generality approach is particularly good for incremental acquisition since each new version space for the concept can be an incremental generalization of the previous version space so that what has been learned from previous examples can be used in accounting for the new examples.

There are a number of well known generalization heuristics that can be used for the least generality approach. Each heuristic is applicable to different kinds of descriptions. The set of heuristics chosen for a given generalization task will determine both the target representation language (L_G), and the generality ordering on the generalizations.

Michalski [1983] gives a comprehensive list of the best known generalization heuristics and also classifies them into two classes—selective and constructive methods. A generalization method is *selective* if it only uses the descriptive elements in the example descriptions (*i.e.*, $L_G = L_E$); and *constructive* if it introduces new descriptive elements that are not present in the basic descriptions (*i.e.*, $L_G \supset L_E$).

Least Complexity

If a generalization task is such that L_G cannot be constrained or biased to the point where a least generality approach is applicable, (for example, when disjunctive descriptions must be considered), one must use the least complexity approach. This approach searches the space in order of complexity and stops at the simplest description that accounts for the data.

The search method in this approach is usually a generate and test method in which the generalizer constructs a sequence of descriptions in order of increasing complexity, testing each description against the data, and halting at the first one that satisfies the data.

There are many possible complexity orderings for a a given generalization space. An important issue is how one can choose an appropriate ordering. Michalski [1983] discusses some of the effects of choosing one ordering over another within one particular representation language and concludes that the choice of the complexity ordering will depend, at least in part, upon the task to which the generalizations will be put. That is, the generalizations are being acquired for some purpose, and this purpose governs the desired characteristics of the resulting generalizations. This flexibility is an advantage that the least complexity approach has over the least generality approach.

Disadvantages

The least generality approach, even in its version space form, suffers from the problem of requiring a constrained representation language: if L_G is a very expressive language, the version space will be too large to be useful. As long as L_G does not include disjunction, then the upper and lower bound sets may be too large to represent or manipulate (for example, the set of all curves in the plane that go through a set of example points). If

L_G includes disjunction, the search space become very dense and the maximally specific generalization of a set of examples is just the disjunction of the set of positive examples This disjunction is just a list of the examples, which is not a generalization at all. In fact, the least generality approach fails completely if disjunctions are permitted, because the upper and lower bounds of the space will never converge, no matter how many examples are presented. Therefore, the least generality approach is only appropriate when L_G can be severely constrained or biased to keep the space of all possible generalizations small.

An important disadvantage of the least complexity approach is that it does not behave well for incremental acquisition. When a new example is presented that does not satisfy the current generalization, the complexity ordering gives no guarantee that the current generalization will be of any use in constructing a new generalization that covers the new example, since a small change in the extension of the description does not necessarily correspond to a small change in its complexity. Therefore, incremental acquisition can turn into a sequence of applications of a non-incremental acquisition method to a growing set of examples, which both subverts the purpose of incremental acquisition and is likely to be very expensive.

There is an additional problem with both the least generality and the least complexity approaches: the stopping criterion never abandons the search for a generalization unless no generalization exists at all. This is fine if there is a guarantee that there will be a useful generalization of the set of examples, but this is not always the case. Especially in matching tasks such as searching a data base for a description that is similar to one just input by the user, part of the task may be to determine whether an acceptable generalization of the examples exists or not. This determination may be very expensive, or not halt at all unless the generalizer can abandon the search at some point.

Justifiability

This thesis claims that generality and complexity are just two instances of a more fundamental ordering criterion—justifiability—and that designing a generalizer on the basis of justifiability can overcome the disadvantages of both the least generality and the least complexity approaches.

A generalization of a set of descriptions can be viewed as a theory of the set. A proposed theory is unjustified if there are many other theories that are equally good. For example, one generalization of a group of people would be "objects with noses". However, there are many other equally good theories relating to eyes, toes, ears *etc.*, so there is no

justification for picking one over the others.

Even if there is just one theory that is better than any others, there may still be no justification for adopting it if there is always a theory of that form that would account for any such set of data. For example, if the data consists of three points on the plane, then there must be just one quadratic that fits them. If we have prior knowledge that these point are examples of some polynomial curve, then we would be justified in guessing that the quadratic that fits them is that curve—a straight line would not fit them, and there are an infinite number of higher order polynomials that fit the points.

But if we do not know whether the points are examples of a polynomial curve, or even whether they have any property in common, then we would not be justified in adopting the quadratic as a theory of the points since some quadratic could be fitted to *any* three points by appropriately choosing the coefficients. It is only when we are given (say) six data points that all lie on a quadratic that we would feel justified in believing that quadratic really accounts for the data, since it is impossible to tailor a quadratic so as to fit an arbitrary set of six points.

These observations can be summarized in the following principles:

- Do not choose any generalization if there are several equally good candidates.
- Do not choose a generalization if it is so tailorable that it could have explained any set of data of the current form.
- Order and search the space of generalizations according to "justifiability".
- Obtain some measure of confidence that there is a generalization of the data to be found, and abandon the search when the generalizations cannot be justified by the data.

The justifiability ordering will depend on the task domain and the nature of the examples. The more justification a particular description would require in order to adopt it, the later it should be in the ordering. To choose a justifiability ordering, one should take into consideration both generality and complexity, because both may contribute to the ordering. Determining an appropriate justifiability ordering is the central design problem in constructing a generalizer for a given task.

There are several different sources for justification. The justification for the quadratic theory above depended on the relation between the number of data points and the degrees

of freedom (the number of coefficients) of the theory. As long as the number of data points was significantly larger than the degrees of freedom, we were justified in adopting the theory.

A second source of justification is a guarantee from some "oracle" (for example, a benign teacher) that a set of descriptions do have something in common. On the basis of this knowledge, the generalizer may be justified in adopting a generalization, even when the generalization has more degrees of freedom than can be justified on the basis of the number of data points alone.

In the absence of an oracle, the generalizer may also derive justification for a generalization of a set of objects from the context of the objects—if the objects are very different, but each occur in the same context, then the generalizer has justification for searching harder for a generalization then if they occurred in different contexts.

The next four chapters describe NODDY's generalization methods, which are based on these principles. Building on the example of NODDY, chapter 7 will discuss the principles more fully.

1.4 An Instructable Robot

The major problem with using robots at present is the difficulty of getting them to perform some task. There are currently two alternatives: a teacher can lead a robot through a sequence of steps that the robot will then remember and repeat indefinitely; or a programmer can write a program in some computer or robot programming language which the robot will then execute. The first method is limited to very simple tasks of minimal complexity. In particular, this method can only introduce the simplest sort of tests or conditionals, since these must be specified as single steps. (see [Grossman, 1977; Grossman and Summers, 1984]). The second method is not limited, but it is a difficult and expensive operation, and is only feasible for tasks that will be repeated many times. A robot that avoided both of these problems would be a very desirable tool.

The ideal instructable robot could be instructed in several different ways so that the instructor could use whichever method of instruction was the most appropriate to a given task. One avenue would be to program the robot in a "task level" programming language in which the instructor could specify the task and the goal of the robot, and the robot would then create its own program to achieve that goal. Another avenue would be to lead the robot through the task in several different situations from which the robot would

generalize to create its program. These two methods could be combined so that the instructor could describe the task and goal and give an example, and the robot would use the task description to guide its generalization of the example and would use the example to interpret the task specialization. This combination would relieve the instructor of having to provide a complete task description or a complete set of examples. The robot might also provide the instructor with a description of its program in a suitable robot programming language and allow the instructor to edit the program (while preventing him/her from putting errors into the program).

Such an instructable robot would involve many different components. NODDY is but a preliminary step towards one of these components. The different components would include:

- Planning: constructing, analyzing, and debugging plans (reasoning about goals, subgoals, prerequisites, *etc.*.) *e.g.*, [Sussman, 1975; Sacerdoti 1974]
- Knowledge about the Robot world and robot actions. *e.g.*, [Mason 1979]
- Reasoning about physical objects in the real world (collisions, support, insertions, *etc.*). *e.g.*, [Hayes, 1979; Forbus, 1984;]
- Path planning (a specialized form of planning). *e.g.*, [Lozano-Perez, 1981]
- Constructing, manipulating, and editing programs. *e.g.*, [Rich, 1981]
- Generalization from traces (*e.g.*, NODDY).
- Generalization by analyzing an example. *e.g.*, [Mitchell, 1983]

1.5 Background

There are three distinct areas related to the work in this thesis. The first is the work in artificial intelligence on learning from examples; the second is the work on inductive inference; and the third is the work on instructing robots. Most of the work in artificial intelligence has been on the problem of acquiring concepts from examples. There have been a variety of different approaches, summarized best in three survey articles [Michalski, 1983; Mitchell, 1982; Deitterich, 1982]. Winston's thesis [Winston 1970] laid the groundwork for acquisition of concepts from examples, and this thesis adopts Winston's incremental acquisition task, but considers a more complex domain than the simple block-structures world.

The closest work in the artificial intelligence area is that of Van Lehn in his recent thesis on learning the subtraction procedures, and Latombe's work on acquiring robot procedures. Van Lehn addresses the same question of how to acquire procedures from examples as this thesis, though Van Lehn is primarily interested in understanding how children learn and constraints on how best to teach them, whereas this thesis is primarily concerned with the problem of generalization. Section 1.5.2 will discuss the relation between his work and this thesis in greater detail.

The problem of acquiring procedures from examples was inspired by J.H. Andreae's work on interactive, domain independent learning, implemented in the PURR-PUSS system ([Andreae, 1977; Andreae and Andreae 1977; Andreae 1972-1984]). Both NODDY and PURR-PUSS acquire procedural knowledge from traces (although NODDY does not attempt to learn interactively), but NODDY uses a completely different representation and acquisition method. In particular, PURR-PUSS does no explicit generalization, and its implicit generalization is very limited.

1.5.1 Concept Learning in Artificial Intelligence

Michalski [1983] gives a list of basic generalization heuristics, which are an improved and extended version of the heuristics that Winston's Arch learner used. The use of these is implicit in NODDY, particularly the *extend range* heuristic. Michalski [Michalski ??] also addresses the problem of choosing between candidate generalizations. His generalizations are always expressed as a disjunction of (almost) conjunctive terms. He allows the user to choose between several different measures of complexity, such as the number of disjuncts or the coverage of the largest term, and biases the search though the space of generalizations to minimize the user's complexity measure. This, however, does not address the question of whether one measure is more useful than another for generalization. Another major difference between Michalski's work and NODDY, apart from the difference in domains, is that NODDY adopts an incremental approach, whereas Michalski assumes that all examples are given at the start, and finds the best generalization of all examples at once.

Mitchell [1983] also addresses the problem of choosing between generalizations. His solution, which works well for generalization spaces that are not too wide or too dense, is not to choose—his version space keeps a specification of all possible generalizations that are consistent with the examples and non examples that have been seen so far. Each new example may reduce the version space, until there is just one option. Section 1.3

discussed the problems faced by version spaces when the descriptions include disjunctions. Mitchell is able to cope with disjunctions only by restricting disjunctions to the top level of the expression, assuming a bias towards the minimum number of disjuncts and keeping a separate version space for each disjunct needed. This essentially separates the generalizer into one part that does disjunction-free generalization and another "meta-level" that handles disjunction. Disjunction is not the only device that causes the version space method to fail—any representation that allows a very wide generalization space and many possible generalizations of a given description will have the same effect. Dealing with a larger generalization space than version spaces can handle is a major goal of NODDY.

Dufay and Latombe [Latombe, 1981; Dufay and Latombe, 1983; Dufay, 1983] addressed the problem of acquiring procedures from traces in a particular robot domain. The basic algorithm was a simplified form of the propagation stage in NODDY, except that the propagation started only at the start and end of the procedure and trace. It was therefore able to acquire a procedure with a single loop. However, all the variables and branching conditions were supplied within the traces, so that the algorithm did not address all the other generalization issues that NODDY does, especially, the integrating of different kinds of generalization.

Langley [1981a, 1981b; Langley *et. al.*, 1983] approaches the problem of finding functional relations between numerical variables. His system (BACON) must be handed sets of values for several "observed" variables, and it will find a function relating the variables. It therefore does constructive generalization of ordered sets of numerical values. One component of NODDY also finds functional dependencies between variables. However, the approaches are quite different in a number of ways, and there are things that NODDY can do that Bacon cannot, and *vice versa.* BACON uses a set of heuristics for incrementally constructing expressions, making intermediate terms out of pairs of variables. The set of heuristics are quite domain dependent, and each new domain [BACON1, BACON2, BACON3,...] involved developing new heuristics, even though all the domains were essentially numerical. NODDY's algorithm is domain independent, and only requires a small amount of type information on the operators out of which it can construct expressions. Some of BACON's heuristics are quite specialized, and can therefore find some relations that NODDY cannot.

Berwick [1982] constructed a system (LParsifal) for acquiring grammar rules for English from positive examples by exploiting the tight constraints on English as expressed in a

particular parser ([Marcus, 1980]). The constraints were expressed in the representation language for the grammar rules: each rule could have only one of 4 possible actions (without parameters) and the condition of a rule was a vector of conjunctions of nominals which were very simple generalizations of the state of the "window" of the parser at the time the action was taken. If there was no rule that applied at any step of parsing an example, LParsifal could create a new rule whose action was whichever one of the possible actions was applicable, and whose condition was the state of the parser's window. If the example were ambiguous—two actions were applicable—LParsifal would ignore the example. If two rules in the same rule group had the same action, LParsifal created a new rule by generalizing the conditions of the rules using the drop-condition heuristic. This acquisition method works because English is constrained such that there are enough unambiguous examples, and English grammar can be expressed in terms of rules having a "local" scope.

1.5.2 Relation to Van Lehn's SIERRA

Van Lehn [1983] constructed a system (SIERRA) that learned to perform multicolumn subtraction from the same example sequences that school children learn from. This task is similar to NODDY's task in that they both acquire a procedure from examples of the execution of the procedure.

A major difference between Van Lehn's work and this thesis is that Van Lehn was primarily concerned with understanding human learning, so that SIERRA was designed and evaluated on the basis of how it conformed to, and explained, children's learning behaviour, particularly the types of mistakes that they made and did not make. NODDY, on the other hand, is concerned with the issue of generalization in machine learning. The thrusts of the theses are therefore very different.

The most obvious difference between SIERRA and NODDY is the different representation for procedures. NODDY represents procedures as elements of a restricted class of flow charts. SIERRA represents procedures by applicative And/Or Graphs. This difference has many ramifications. For example, NODDY represents iteration explicitly as loops, whereas SIERRA represents iteration implicitly as recursive calls to a parent node of the graph in SIERRA. NODDY's procedures have explicit state variables and the data flow is distinct from the flow of control. SIERRA's procedures, on the other hand, do not have state variables, and the data flow is parallel with the flow of control.

Van Lehn argues for these design decisions very convincingly on the basis of the data from children's learning. The arguments do not seem to carry across to the robot procedure tasks, however. A crucial differences between SIERRA's subtraction task and NODDY's learning task is the nature of the pattern. A key feature of the arithmetic domain which governed the design of Sierra is that all information necessary to make a decision is always present in the current pattern—actions only add to the pattern and nothing ever goes away. The data flow in Sierra's procedures is the focus of attention within this constantly visible pattern. The patterns of the robot domain, on the other hand, are ephemeral—the pattern changes upon every action, and there is no way to recover a previous pattern. The data flow in NODDY's procedures is the pattern values that must be remembered because later actions depend on them. Finding and constructing these dependencies is an important part of the generalization process that NODDY performs.

1.5.3 Inductive Inference

Angluin and Smith [1982] present an extensive review of the literature on inductive inference. The goals of most of the work on inductive inference are related to artificial intelligence work on learning, but the methods and approaches are very different. The issues that NODDY addresses do not seem to be addressed by the inductive inference work. We therefore refer the reader to Angluin and Smith's excellent paper rather than present any review.

Chapter 2

Task, Representation, and Domain

2.1 Procedures and Examples

NODDY's task is to take a sequence of examples of a procedure and infer the procedure from them. To teach NODDY a procedure, a teacher generates a set of examples or *traces* of the procedure by executing the procedure several times and recording the sequence of actions performed, along with any sensory input or feedback (the *patterns*) from the world resulting from the actions. Each trace is a linear sequence of actions and patterns.

If the procedure ignores the sensory feedback or there is no feedback in a given domain, then the procedure must generate exactly the same actions under all circumstances, so that the traces will all have the same sequence of actions, and the procedure can be trivially inferred from the examples since it is exactly the sequence of actions in any of the traces. This is the way many industrial robots are currently "taught". In this rote "learning" the robot is led by the trainer through a sequence of actions which it then repeats endlessly without variation. Though useful, this technique is severely limited in the range of procedures that can be taught.

The procedure acquisition task only becomes interesting when the procedures do take into account the sensory feedback or pattern and specify different actions at different times depending on the pattern. Such procedures may involve conditional branching, iterative loops, or other dependencies between the patterns and the actions. Each trace will then be different, resulting from a different path through the procedure, and the procedure acquisition task is to find a procedure that is a generalization of all the traces. This procedure will have constructs such as conditional branching or iterative loops which are not present in any of the traces and must therefore be inferred from the traces.

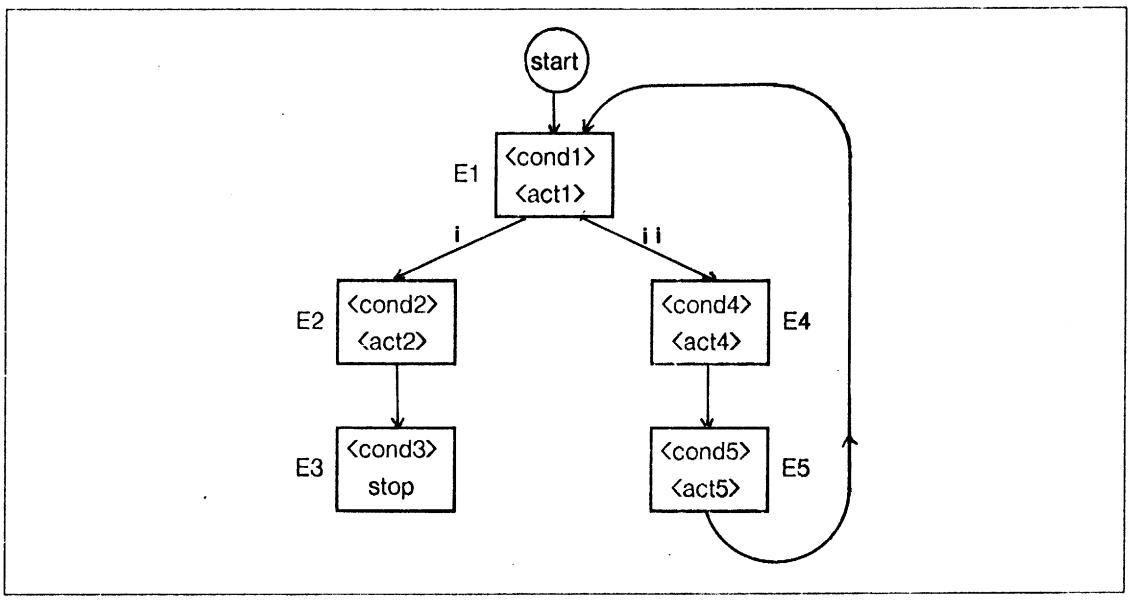
Such procedures must contain several kinds of information: the descriptions of the actions to perform, the "flow of control" specifying the order in which the actions must be performed, the conditions at any choice points in the procedure, and other dependencies between the actions and the feedback patterns.

The following three sections define these procedures more precisely by describing the representation of procedures used in NODDY. Section 2.5 describes a simple robot domain

which supports interesting procedures with these characteristics. NODDY has been tested in this domain and all the examples in the thesis are taken from this domain.

2.2 Representation of Procedures

NODDY's representation of procedures is based on flow diagrams, and all the diagrams of procedures in this thesis are shown in flow diagram form. A flow diagram consists of boxes connected by arrows representing the flow of control. In general, flow diagram boxes may contain many different types of information, but the boxes of NODDY's procedures all contain a description of a *condition* and a specialization for an *action.* Such a flow diagram box with a condition and an action will be referred to as an *event.* One of the events of a procedure is identified as the *start event,* and one or more as *stop events.* Each event, except the stop events, has one or more arrows, or *sequence links,* proceeding from it. If there is more than one sequence link proceeding from an event, the links may be labeled with priorities which define a partial ordering on the links. The *succeeders* of an event A are the events pointed to by sequence links out of A and the *preceeders* of A are the events at the base of links that point to A.



— — Figure 2-1 A Simple Procedure — —

A procedure with five events, and a single loop. Control will pass into the loop (events E4 and E5) only when the pattern resulting from action <act1> does not match <cond2> (which is marked with a higher priority) and does match <cond4>.

For example, figure 2-1 shows a simple procedure with five events. There are two sequence links proceeding from event **E1**; the one pointing to **E2** is labeled with a higher priority than the one pointing to **E4.**

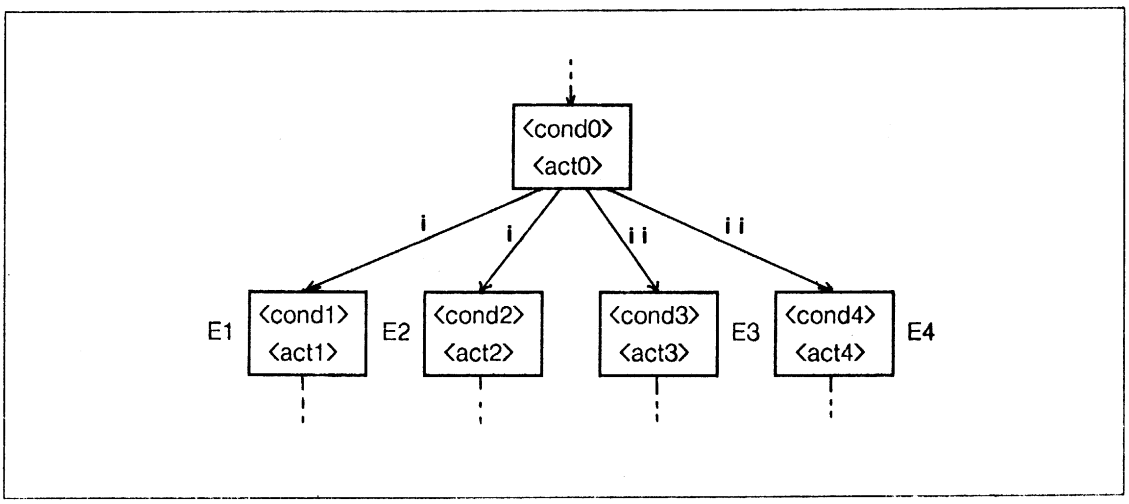
A procedure is executed by beginning at the start event and following the sequence links to the succeeding events. Whenever an event is entered, the action it specifies is performed. The feedback pattern that results from this action is then matched against the conditions of each of the succeeders of the current event. If the links are labeled, then the pattern is matched against the conditions of the higher priority events first. Control passes to the first succeeder event whose condition matches the pattern. If no succeeder event has a condition that matches, the procedure fails and halts. If a stop event is reached, the procedure halts successfully.

For example, during the execution of the procedure in figure 2-1, when the robot (or whatever is executing the procedure) enters event **E1**, it performs the action <act1>. This will cause the world to return a new feedback pattern. Because the highest priority link out of **E1** points to **E2**. the new pattern will first be tested against the condition <cond2>. If it matches, the robot will enter event **E2**, and perform the action <act2>. If the pattern does not match <cond2>, then it will be tested against <cond4>—the condition of the next highest priority event. If it matches, the robot will enter **E4**. If the pattern matches neither <cond2> nor <cond4>, the procedure will fail and halt. If the robot enters **E4**, it will perform the action <act4>, which will result in a new pattern form the world. As long as this pattern matches <cond5>, control will pass to E5, the robot will perform action <act5>, and then loop back to **E1**. The procedure will halt successfully only when the pattern following an <act1> matches <cond2> and control passes to **E2** and then **E3** where the procedure halts.

Conditional branching is therefore represented by multiple links out of one event with the conditions on the succeeding events being the *branch conditions.* The event with multiple links is referred to as the *fork event,* (**E1** in figure 2-1). A *merge event* is an event with multiple links pointing into it. In figure 2-1, E1 is also a merge event. It is not necessary for a fork event to be a merge event also, but is commonly so in simple loops such as that of the procedure in figure 2-1. A *branch* of a procedure is a linear sequence of events, beginning at the start event, a merge event, or an event immediately following a fork event and continuing up to a stop event, a fork event, or an event immediately preceding a merge event. The branches of a fork event are the branches immediately

following it. The branch conditions of a fork are just the conditions of the first events in its branches.

Iteration is represented by a loop in the flow diagram. There must be at least one fork event in a loop, and one of the branch conditions of the fork event (the *exit condition*) must be external to the loop if the procedure is to be able to halt successfully, since control would otherwise circle the loop indefinitely. Note that this is a "DO UNTIL" type of iteration, not a Fortran style "DO LOOP"—control will iterate around the loop until the exit condition is satisfied and control exits.



— — Figure 2-2 Deterministic Branching — —

A fork event with four succeeders. Events E1 and E2 are marked to be at a higher priority than events E3 and E4.

The procedures that we will consider are constrained to be **deterministic** in that they must specify exactly one possible action at each step. One consequence of this constraint is that at a fork event, only one branch can be taken at any time. Therefore, there must be no possible pattern that could match more than one of the branch conditions at any single priority. For example, in the procedure fragment of figure 2-2 this constraint would mean that there must be no pattern that would satisfy both <cond1> and <cond2>, nor both <cond3> and <cond4>. If some pattern satisfied both <cond1> and <cond3>, then that pattern would always cause control to flow to E1 since the link to E1 has a higher priority. This would be useful if <cond1> were to be considered a special case of a more general condition <cond3>.

This determinacy constraint also constrains the space of allowed actions, but this will be discussed in section 2.4.

Traces are represented in exactly the same manner as procedures but are further constrained. A trace event may have no more than one succeeder and one preceeder, so that there can be no forks, merges, or loops in a trace. Also, the events must be maximally specific; *i.e.*, the condition of a trace event must be a pattern, defined in the next section, and the action of a trace event must be a primitive action, defined in section 2.4.

2.3 Representation of Patterns and Conditions

The feedback pattern from the world will depend entirely on the particular domain chosen. In general, the pattern may have a number of distinct components. In the robot domain described in section 2.5, these components correspond to the different sensors—the position of the robot, its orientation, and the signals from its force and grasp sensors. In other domains, they would correspond to whatever aspects of the world that the executor of the procedure can access and base its decisions on. NODDY currently requires a domain in which each component of a pattern can be specified by a list of parameter values, whether numerical or symbolic. The pattern will then be represented by a conjunction of the lists of values specifying each component of the pattern. Not every component need be present in every pattern, for example, the force feedback component of the pattern in the robot domain will be present only when the robot is in contact with something.

The condition of an event is a description of a class of patterns for which control should enter the event. It is therefore a generalization of patterns. A condition is represented as an implicit conjunction of components corresponding to the components of the pattern. A condition will match a pattern only if each component of the condition matches the corresponding component of the pattern. Each condition component, or *node*, is represented by the name of a *template*, a list of *parameter values* and a set of *past values*.

The template contains, among other things, the specification of a predicate and its parameters. A particular node represents an instantiation of its template with particular values for the parameters. For example, the node `[vertical-strip (3 4)]` would be an instance of the `VERTICAL-STRIP` template which has two parameters, `P1` and `P2` and a predicate `(LAMBDA (x,y) (AND (≤ x P1) (≤ P2 x)))`. (For convenience, we will always print a node with a shortened form of the predicate from the template, so that the above node will be given as `[vert-strip:`$3 \leq x \leq 4$`]`.) A condition will be satisfied

by a pattern iff the instantiated predicates of each node in the condition are satisfied by the corresponding component of the pattern. (A null node is only satisfied by a null component of the pattern). The past values of the node are a set of the actual pattern values of which the condition is a generalization. They are ignored during the execution of a procedure, but NODDY uses them while generalizing the procedure to accommodate a new trace. The disjunction of the past-values is the most specific generalization possible of data, and is used somewhat as the maximally specific bound of Mitchell's version space algorithm. This will be discussed further in chapter 6.

Note that the space of conditions that NODDY can represent is limited by the set of templates. Even with only one template, there may still be infinitely many possible conditions since there may be infinitely many possible values for the parameters of the template, but NODDY cannot represent a predicate for which it does not have a template.

2.4 Representation of Actions

The set of legal primitive actions, like the patterns, is determined by the domain. An action may have a list of parameters associated with it. For example, one of the primitive actions in the robot domain is the `MOVE` action which has two parameters specifying the distance and direction that the robot is to move. NODDY represents actions by the type of the action and a list of the values of the parameters. For example, `MOVE` {1, 60°} would be a legal action in the robot domain.

Each action in a trace must be one of the primitive actions specified by the domain, but the action descriptions in the events of a procedure may be generalizations of primitive actions. The generalized actions are represented by the name of an action *type* and either a list of parameter values (like the primitive actions) or a function expression and a list of variables. Like the conditions, a generalized action also has a list of *past values*—the parameters of the primitive actions from which it was generalized.

The generalized actions are somewhat different from the generalized patterns (*i.e.*, conditions). The conditions are predicates describing a set of patterns. A condition is satisfied by any of the patterns it covers. An action, however, must be executed, not satisfied. A generalized action, although it may describe a large set of primitive actions, must specify just one of those primitive actions in any particular situation. For example, the `MOVE-TO` {(1,0)} action covers any primitive `MOVE` action that will take the robot to the position (1,0). However, at any particular time, the robot will be in just one position,

and the MOVE-TO {(1,0)} action will specify just one primitive MOVE—the one that will get from the current position to (1,0).

The *type* of an action refers to a template in a hierarchy of generalized action templates. Each template contains a parameterized description of a generalized action. For example, the MOVE-TO template specifies that any MOVE-TO action has a list of just one parameter which must be a position, and that the action specifies the primitive MOVE that gets from the current position to the position specified by the parameter. Each template also describes its relation to other templates in the hierarchy.

The action hierarchy actually consists of several disjoint hierarchies, each corresponding to one of the primitive actions. For example, the **MOVE** hierarchy contains templates for generalizations of the primitive MOVE action. The *class* of a generalized action is the particular hierarchy that it is in (named by the primitive action on which the hierarchy is built). The action hierarchies will be described in detail in chapter 5. The action hierarchies are clearly domain dependent and must currently be provided by the user, but chapter 8 outlines how NODDY might construct some of its own action templates in the future.

The simplest generalized actions are those with constant parameters, such as MOVE-TO {(1,0)}, but the parameters may be replaced by a function that will evaluate to a list of parameter values. The function is represented as a lambda expression and a list of variables each referring to some pattern component in the procedure. We will refer to such actions as *functional actions.* For example, the MOVE *<from (1,0) to remembered position>* action of the RETRIEVE-2 procedure (event 5 of figure 1-5) is actually represented by the action

```
MOVE {(lambda (a1) (vect->list (-pos (first a1) (1,0))))}
     {(V1: position component, event E1)}
```

This specifies a MOVE action whose parameters are the result of applying the lambda expression to the first element of the value of the variable V_1 which is the list of parameter values of the position component of the pattern in event 1; *i.e.*, the MOVE action that would take the robot from the position (1,0) to the position it was at during event 1.

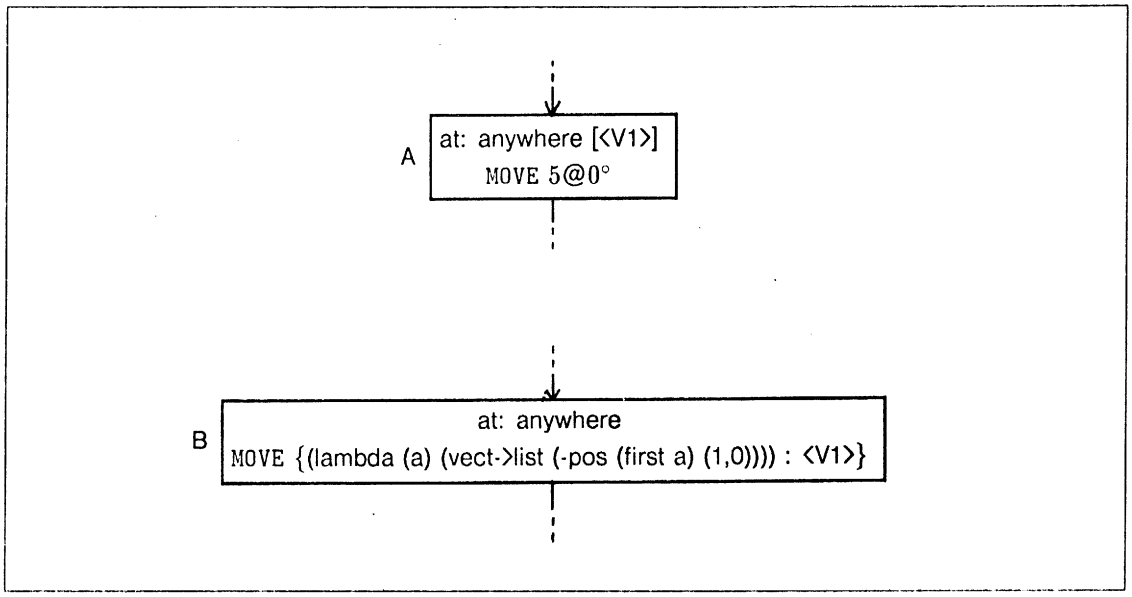
If a variable refers to a component of the current pattern, it is a *local variable,* and represented by just the component name. If all the variables are local variables, then the action is called a *local function.* For example, the action MOVE *<back to entry point of row>*

in event 4 of the RETRIEVE procedure (see figure 1-2) is the local function:

`MOVE-TO{(lambda (a) (ncons (constr-pos 1.0 (y-part (first a)))))}`
`{<current-position>}`

This specifies a move from the current position to the position whose x component is 1.0 and whose y component is the same as that of the current position.

Note: For clarity we will generally drop the braces around the parameters of an action, and add some notation to distinguish the parameters when necessary, so that we will write `MOVE` {1, 60°} as `MOVE` 1@60°, and `MOVE-TO` {(1,0)} as `MOVE-TO` (1,0). We will also abbreviate functional parameters by writing only the body of the lambda expressions, and substituting the variables in place of the lambda arguments, so that the `MOVE` action above will be written as `MOVE (vect->list (-pos (first <`V_1`>) (1,0)))`. Occasionally, we will also omit the `ncons`'s and `first`'s in the function since these are just a result of representing the parameters of patterns and actions as lists.



— — Figure 2-3 A Functional Action — —

The action of event B refers to the position component of the condition of event A, but this position component is the generalized position [`anywhere`]. The action is actually a function of whatever position matches the [`anywhere`] node.

Note also that a functional action actually depends on a *pattern*, not the *condition* referred to by the variables. For example, the variable in the `MOVE` *<from (1,0) to remembered position>*

action above refers to the position component of the condition in event E1. However, this position component (which is marked with the name of the variable) is actually the node [anywhere], which has no parameters. The MOVE-TO action is not a function of [anywhere], but of the value of the position component of the pattern that matches the [anywhere]. That is, If the robot is executing the procedure, it will assign the parameters of the position that matched the [anywhere] node to the variable $\langle V_1 \rangle$; then, when it reaches the MOVE-TO action, it will apply the lambda expression to the value of $\langle V_1 \rangle$ and move to the resulting position.

The following table summarizes most of the terms defined in this chapter. It may be useful for future reference.

Pattern	The feedback from the world after every action.
Primitive Action	One of the legal actions that the executor of the procedure may perform.
Event	The basic element of a procedure. Contains a condition and an action.
Condition	A predicate on patterns specifying when control may enter an event. The predicate is the conjunction of the predicates specified by each *node* in the condition.
Component	A "slot" of a pattern or condition corresponding to one part of the feedback. In the robot domain, there is one component for each sensor. Each component of a pattern will contain an actual value returned by the sensor. Each component of a condition will contain a *node.*
Node	The contents of one component of a condition. Represented by the name of a *condition template* and a list of parameter values.
Condition Template	A description of a family of nodes, defined by a list of parameters and their types and a parameterized predicate on patterns (actually on the relevant component of patterns).
Generalized Action	A specification from which the executor can determine a primitive action at the time of execution. Represented by a template name

and a list of parameter values or by a template name, a lambda expression and a list of variables.

Action Template A description of a family of generalized actions. Contains a parameterized specification for computing a primitive action from the action parameter values and the current pattern.

2.5 2-D Robot domain

The current implementation of NODDY acquires procedures for a simple robot domain. The domain was designed to be sufficiently interesting to explore the issues of acquiring and generalizing procedures, not necessarily to be a usefully realistic domain. It is also intended to be easily extendable so that more realistic components could be introduced later as appropriate.

The robot domain is a two dimensional world containing a point robot, *i.e.*, the robot occupies no area. In addition to the robot, there are two dimensional objects or obstacles in the world which may take any shape and may be either fixed or movable. The robot can move around the plane, rotate about its axis, bump into objects, grasp them, "carry" the objects around, and let them go. Moving is always done in straight line segments, and cannot be done concurrently with rotation, (no pair of actions may be done concurrently). The robot cannot tell the difference between a fixed and a movable object except by grasping it and trying to move it. The robot is not allowed to push objects without grasping them—even movable objects remain fixed until grasped. Grasping is done by "sticking"—since the robot has zero size it cannot clamp onto an object. Rather, when it is in contact with an obstacle it can simply stick to it, forming a rigid joint until it `UNGRASP`'s the object.

2.5.1 Actions

The primitive actions of this robot domain are `MOVE`, `ROTATE`, `GRASP` and `UNGRASP`. The `GRASP` and `UNGRASP` actions do not take any parameters and consequently do not have any generalizations.

The `MOVE` action is the most interesting action. It takes two parameters—the direction to move in and the distance to move. The direction parameter is an essential parameter of the `MOVE` action, in that it must be specified for the action to be begun. The distance parameter, on the other hand, is a guard parameter. *i.e.*, a stopping condition which which

determines when the movement in the specified direction should be stopped.* There are other valid guard conditions such as hitting an object, so that the robot could perform the action MOVE @90° until: [contact] which would mean stopping when the force sensors indicate contact with an object. In executing a procedure, the robot will accept any guard condition expressed in terms of the continuously monitorable features of the world. For the 2-D robot world, these include the distance traveled and the state of the force sensors. The position is not a continuously monitorable feature.

When the teacher is entering a MOVE action while creating a new trace, the robot cannot tell what stopping condition the teacher used and therefore expresses the move as a MOVE *<dist>@<dir>*. All the moves in traces are therefore MOVE *<dist>@<dir>*, and NODDY must infer other stopping conditions the teacher intended, if any.

The ROTATE action is similar to the MOVE action, but less interesting because it is constrained to one dimension. It takes two parameters, the direction to rotate, which is positive or negative, and an angle specifying the amount to rotate. The former is the essential parameter, and the latter is the guard parameter.

2.5.2 Patterns and Conditions

The sensory feedback or pattern that the robot world returns in response to an action has up to four components: the **position**, the **orientation**, the **contact-angle**, and the **grasp-angle**. The position is the position of the robot in the plane, given in x-y coordinates. The orientation is the direction the robot is "facing" in, given as an angle from the x-axis. Both of these components are always present in the feedback pattern. The contact-angle is the force feedback. It gives the direction of contact between the robot and an obstacle the robot is touching. Since the robot is assumed to be of zero size, this direction is normal to the edge of the obstacle at the point of contact. The direction is relative to the x-axis and is not affected by the orientation of the robot, (this was an arbitrary choice of no consequence). If the robot is not in contact with any obstacle there is no contact-angle component of the pattern returned. If the robot is in contact with an obstacle, it may grasp the obstacle. The grasp-angle of the following patterns is then the direction between the robot and the object it is grasping and is identical to what the force angle would be if the robot were not grasping the object. There is no grasp-angle component of the

* The distinction between essential and guard parameters will be discussed more fully in chapter 5

pattern if the robot is not grasping anything. A GRASP action when the robot is not in contact with anything also produces a null grasp angle component.

Chapter 3
Generalizing Procedures

This chapter gives an overview of NODDY's algorithm for acquiring procedures. It describes the interrelation of the three different types of generalization in NODDY that are each described in detail in the following chapters.

3.1 Incremental, Syntactic Acquisition

Section 1.2 placed NODDY within the spectrum of learning tasks. Essentially, NODDY acquires procedures incrementally from a sequence of teacher-provided traces, where each trace is a positive instance of one path through a procedure. NODDY is incremental in that the current description of the procedure being acquired is incrementally improved with each new example. The initial version of the procedure is just the first example that the teacher provides. This version is very specific—it applies to only one situation and specifies exactly the same sequence of actions every time it is executed. NODDY uses each succeeding example to generalize the procedure until the teacher is satisfied with NODDY's procedure. The incremental approach allows the teacher to tailor each new example to the deficiencies of the current procedure description, and to stop as soon as a correct procedure is obtained, which is in contrast to the non-incremental approach, in which the teacher must know beforehand exactly what examples should be given. This approach also allows for easier extension to a system that generated its own examples by experimentation. One consequence of the incremental approach is that NODDY should not have to remember any previous traces—its current version of the procedure should contain all the necessary information for constructing the new version with the new trace. Several of the key principles underlying NODDY's algorithm are based on the assumption of incremental acquisition,

Another characteristic of NODDY is that it is syntactic—it only uses knowledge of the syntax of valid procedure descriptions and does not use any knowledge of the goal or purpose of the procedure, nor of the meaning or effect of any of the actions. A "complete" procedure learner would certainly use the semantics of the procedure being learned, perhaps in a similar way to Mitchell's LEX2 [1983] analyze a single example, determine the purpose of the individual steps and then generalize the example on the basis of the analysis. NODDY is an exploration of what can be achieved without using

this sort of semantic information. NODDY's algorithm is also as domain independent as possible. All the domain specific knowledge is confined to two places—the action and condition generalization hierarchies and a small data base of operators and type information. NODDY accesses this information, but its algorithm is independent of the content of the hierarchies and and operator data base specific to the domain. Future work will address the problem of constructing the hierarchies themselves using syntactic generalization.

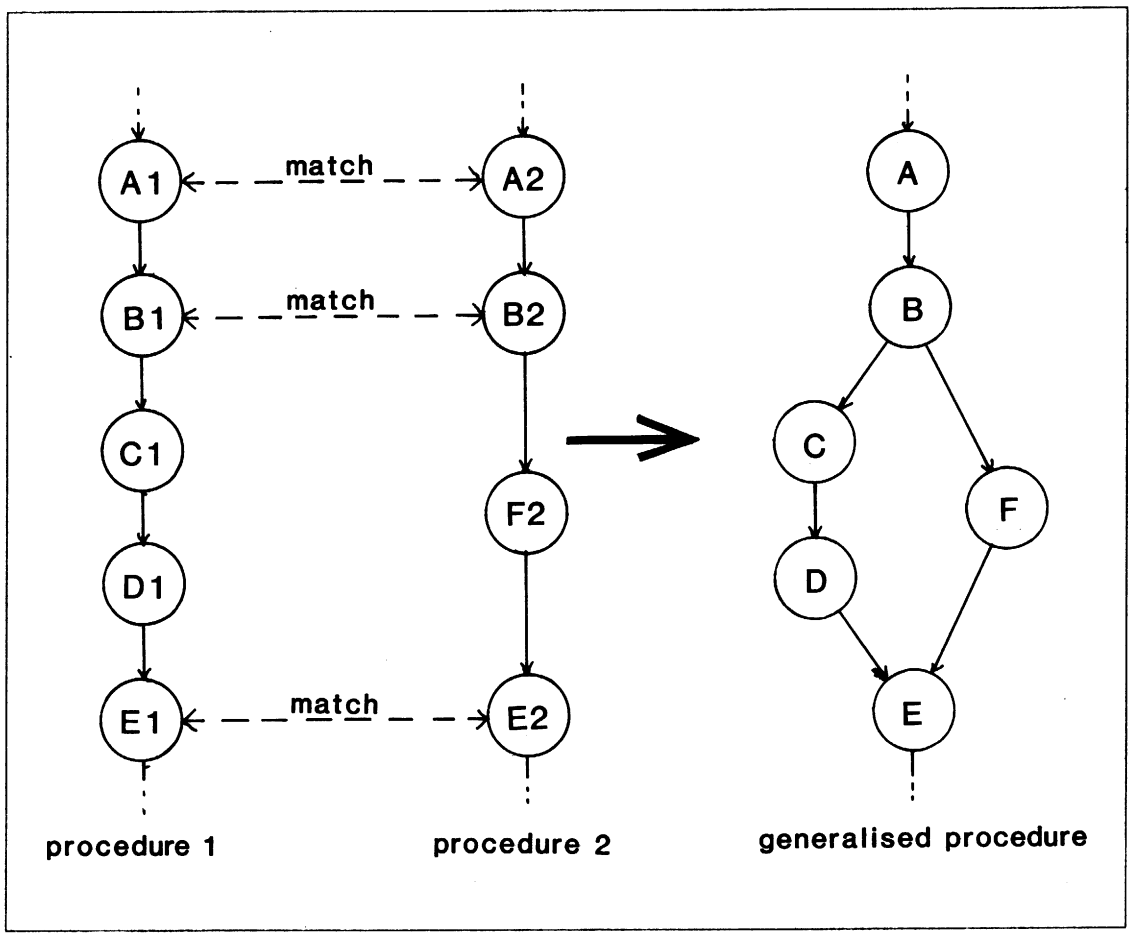
3.2 Generalizing Structured Descriptions

A procedure is a structured description containing a set of nodes (the events) and relations between the nodes (the sequence relations and the variable reference relations). Finding a generalization of two structured descriptions involves matching the two descriptions, identifying the differences, and generalizing the appropriate parts of the descriptions to resolve the differences.

Matching two structured descriptions requires two different matchers: the *structure matcher*, which will find the appropriate correspondence between the nodes of the two descriptions and construct a new structured description by merging the nodes and relations appropriately; and the *node matcher* which will determine whether a set of nodes could be merged, and will create a new node that is a generalization of the nodes if necessary.

The structure matcher can generalize the descriptions in several different ways. For example, in matching two procedures, the structure matcher will find a correspondence between the events of the procedures. If there are events in each procedure that are not paired with any nodes in the other procedure, the structure matcher could create a new procedure with two branches instead of one (see figure 3-1). If the structure matcher pairs an event in one procedure with several events in the other procedure, it could form one of two other kinds of structural generalization—either forming a loop (which is a special case of forming a *group* of nodes), or merging several branches into a single, more general branch (see figures 3-2 and 3-3).

The node matcher will determine how similar two nodes are, and unless they do not match at all, it will return a generalization of the two nodes. How the node matcher matches and generalizes the nodes will depend on the nature of the nodes—if the nodes are themselves structured descriptions, it will be another structure matcher; otherwise it will use whatever generalization methods are appropriate to the representation language



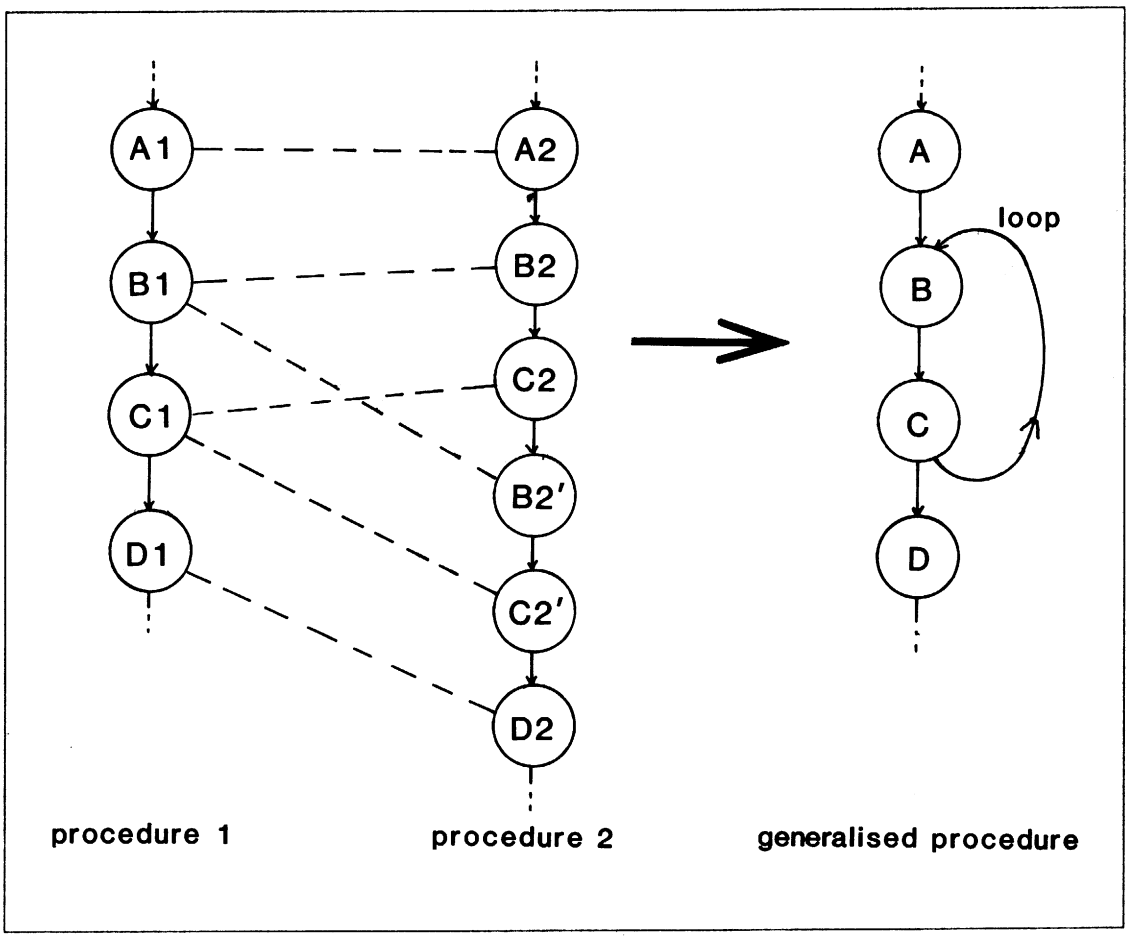
— — Figure 3-1 Adding a Branch — —

The first two and the last events of procedure 1 and procedure 2 are paired, but the other events do not match. The structure matcher merges the matched events, and forms two alternative branches from the events that do not match.

for the node descriptions.

NODDY's node matcher contains two separate matchers—the condition matcher and the action matcher, which use different generalization methods, although they both do a type of constructive generalization appropriate for descriptions in terms of parameters.

Given two structured descriptions, there will be many ways that the structure matcher can put the nodes in correspondence, depending in part upon how radical a node generalization it will accept from the node matcher. If it accepts very radical node generalizations (eg, the generalized node "**anything**"), it will produce a new description with very few, very general nodes. If, on the other hand, it insists on a perfect node match (no node generalization at all), it will tend to produce a new description with many, very specific,



— — Figure 3-2 Forming a Loop — —

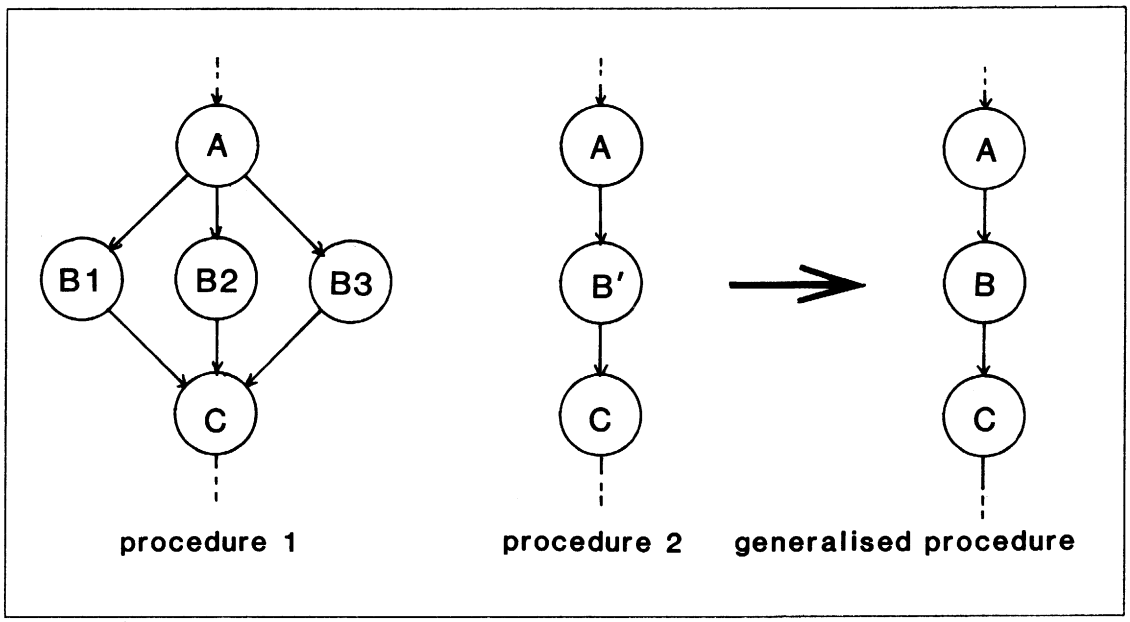
Events B and C of procedure 1 are each paired with two events from procedure 2. By merging all the paired events, the structure matcher forms a new procedure with a loop.

nodes. Once it has fixed a pairing of the nodes, the structure matcher may still choose to merge the nodes and their relations in different ways.

The *bias* [Mitchell 1983] of the matcher determines what choices the matcher will make. NODDY's overall bias is to introduce a new branch into the procedure only when forced to, and to make the branch as short as possible if one must be introduced. (The details of the bias will be given in the following chapters). The effect of this bias is to prefer procedures with few branches, each of which is as general as possible.

3.2.1 Backtracking or Determinism

In constructing a matcher for structured descriptions, the major issue is how to search the space of all possible pairings of nodes, subject to the various constraints on descriptions.



— — Figure 3-3 Forming a More General Branch — —

Events B, B′, and B″ of procedure 1 are all paired with event B of procedure 2. By merging all the paired events, (which will require the node matcher to find a generalization of all the B events), the structure matcher forms a more general branch in the new procedure.

There are two extreme approaches to controlling this search.

One extreme is to perform an exhaustive search of all possible ways of pairing nodes from the two descriptions. The structure matcher would then construct a new description from each set of pairings, and discard those that do not satisfy the constraints on valid descriptions. The matcher would then choose the best description that was left according to some measure.

If matching two procedures, this approach would generate the $2^{n_1 \times n_2}$ candidate procedures generated by all possible pairings of procedure and trace events (where n_1 and n_2 are the number of events in the procedure and the trace respectively). Most of these candidate procedures would probably be eliminated because they did not satisfy the constraints on procedure descriptions—candidate procedures that did not have a single start event, had no stop event, or were not completely deterministic. The remaining candidates would be compared according to a measure perhaps based on the goodness of the event matches and on properties of the structure, such as the fewest branches.

This approach does a large amount of backtracking, constructing many pairings and candidate procedures which it later abandons.

The approach at the other extreme avoids backtracking completely by analyzing the descriptions being matched to determine which nodes ought to be paired, and creating the new description on the first pass. The structure matcher would always propose pairings for which the node matcher could form valid generalizations, and would only construct valid descriptions. If it paired the nodes incrementally, it could use the partially constructed description to determine the next nodes to pair.

While this approach would be very efficient, it will seldom be possible. At best, a matcher could approach this end of the spectrum by exploiting the constraints on the descriptions to guide the choice for the pairing and reduce the search. NODDY does exactly this and is committed to minimal backtracking.

3.2.2 Justification Based Matching

The basic principles underlying the matching and generalization in NODDY are:

- Adopt a generalization only when there is sufficient evidence for the choice to be unambiguous and untailorable.
- Abandon the search for a generalization when you no longer have sufficient evidence to justify a generalization if you were to find one.

Applied to the structure matcher, this means that no two events should be handed to the event matcher unless the structure matcher has justification for believing that they might match, and no paired events will be merged in the new procedure unless the generalization of the events produced by the event matcher justifies merging them. If the justification was well founded, there will be no need to retract the pairing.

A prime source of justification for the structure matcher is the *context* of two events. If the preceeders of a trace event have been merged with the preceeders of a procedure event, this provides some justification for the event matcher to pair the two events. If their succeeders have also been merged, this is even stronger justification for pairing them, and the event matcher can consider more powerful generalizations.

Applied to the event matcher, these principles mean that the matcher should only look for generalizations of a degree that is supported by the justification for trying to match them. The space of possible generalizations of events is structured in several "levels". Searching for a generalization in a higher level requires more justification than searching for a generalization in a lower level. When the event matcher is asked to match two

events, it only considers generalizations in the levels that are consistent with the structure matcher's justification for pairing the events.

Chapter 7 will discuss the various sources of justification in more detail.

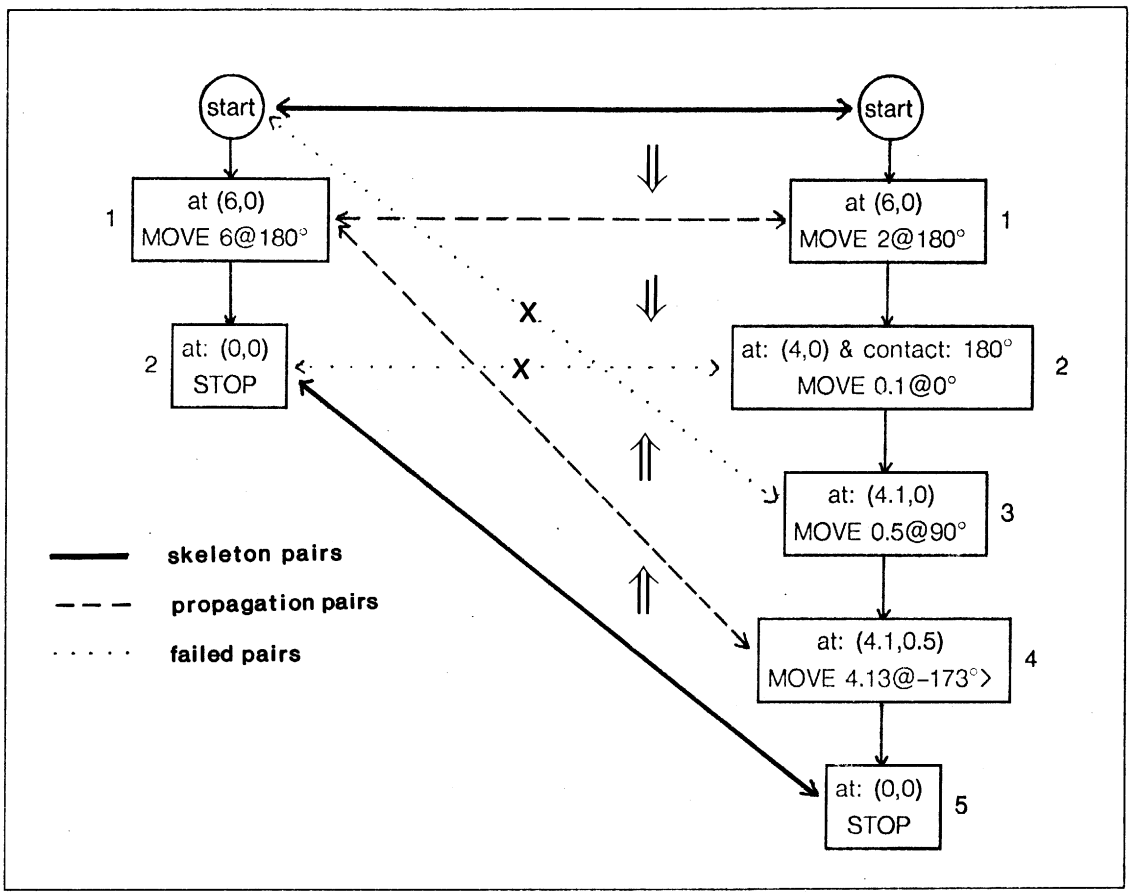
3.3 Overview of the Matching Stages in NODDY

Following the principles of the previous section, NODDY's algorithm is built around a structure matcher consisting of several stages. Each stage finds some candidate pairings of events in the procedure and the trace, and invokes the event matcher to match the events. If the event matcher is able to match the events within the justification level specified by the structure matcher, the structure matcher will use the generalized events to construct more of the generalized procedure. Each stage of the structure matcher uses the result of the previous stage to guide its search for new pairings.

Before trying to find any generalization of the procedure, NODDY first checks to see whether the procedure already **includes** the trace. Because of the constrained structure of procedures, NODDY need not do any search to check this, but simply "executes" the procedure using the patterns of the trace as the sensory feedback, and checking that the actions specified by the procedure are identical to the actions in the trace. If the procedure does include the trace, there is no need to find a generalization, so NODDY proceeds to the following stages only when there is at least one difference between the procedure and the trace.

In the first stage—the **skeleton stage**—of matching a procedure and a trace, NODDY constructs a *skeleton* of the new procedure based around the *key events* of the procedure and trace. The key events are those events that have some property that identifies them as unique within the procedure or trace. For example, start events are always key events because there is only one start event in any procedure or trace. Because of their uniqueness, the key events of the procedure can be paired easily and reliably with key events in the trace having the same property. This shared property is strong evidence (but not a guarantee) for assuming that the events should be paired and merged, so the structure matcher will only merge a pair of key events if the event matcher can match them without any generalization, *i.e.*, a perfect match. The pairs that do match perfectly are merged and form the skeleton of the new procedure.

The **propagation stage** uses the context of the skeleton as the basis for choosing new pairings. NODDY propagates forward and backward along the sequence links from the



— — Figure 3-4 Skeleton and Propagation Stages — —

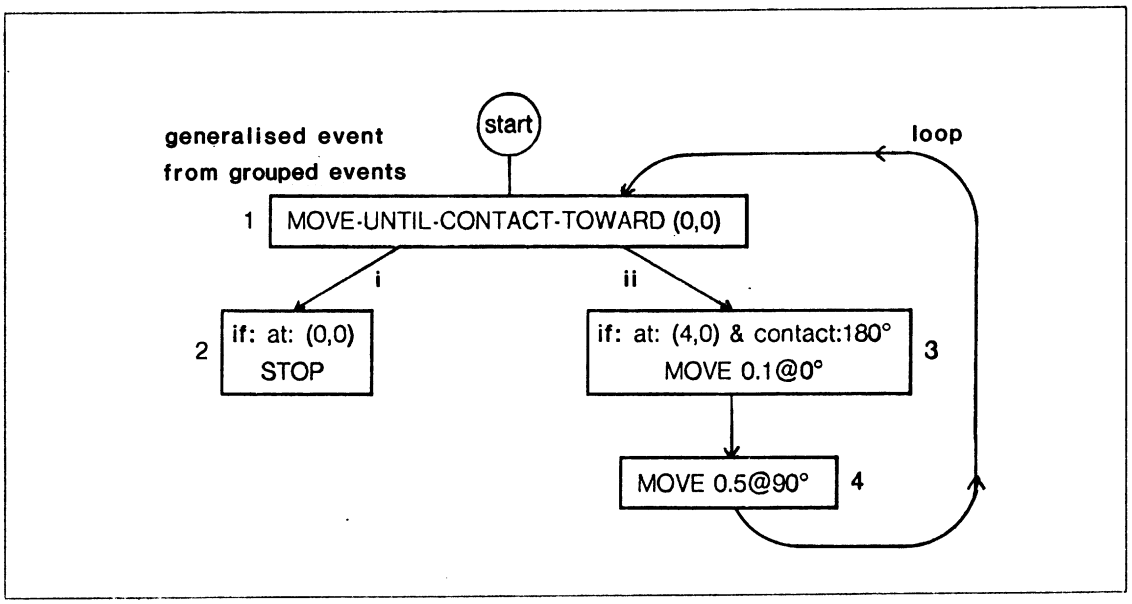
Matching the first two traces for the TURTLE procedure. The skeleton stage pairs the start events and stop events (solid lines). The propagation stage propagates forward and backward from these pairs to find the two pairs involving event 2 of the first trace (dashed lines). The propagation halted at the dotted lines because the event matcher could find no generalization of the events.

key event pairs, pairing the trace and procedure events it reaches. The event matcher matches each pair of events as it is found. If they match, they are merged, and the propagation continues; if they do not match, then the pair is rejected and propagation along that path halts. Figure 3-4 shows the skeleton and propagation stages applied to the first two traces of the TURTLE procedure from the scenario (section 1.1.1).

Each pair of events found in the propagation stage shares some context justification—either the preceeders or the succeeders of the events have been merged. Therefore, the structure matcher has stronger justification for believing that the events should be paired and asks the event matcher to consider partial matches of the events corresponding to first level event generalizations. This introduces the first (event) generalization into the new

procedure.

The **grouping stage** introduces structural generalization into the new procedure. If the propagation stage paired one trace event with several procedure events, or one procedure event with several trace events, the grouping stage attempts to merge all the events together into a single generalized event, again using first level event generalization.



— — Figure 3-5 Introducing a Loop — —

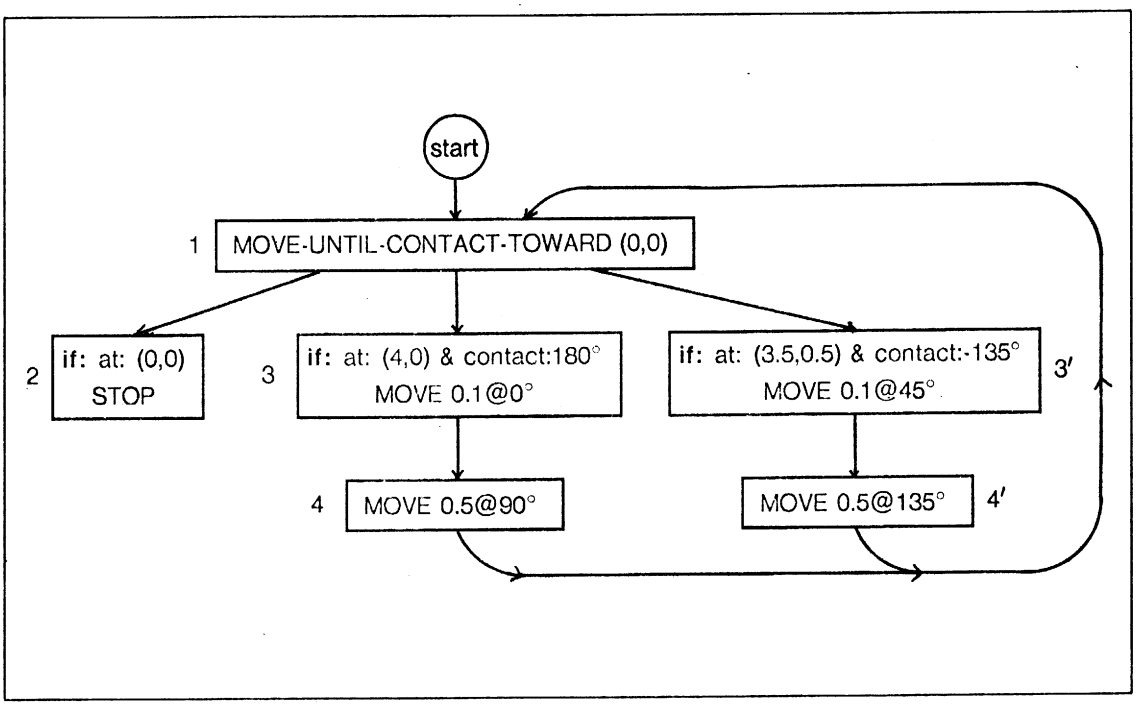
The new procedure with a loop produced by the grouping stage from the pairs in figure 3-4. Event 1 is a generalization of event 1 of the first trace and events 1 and 4 of the second trace which were grouped and generalized by the grouping stage. The loop resulted from the combining of events 1 and 4 in the second trace.

One effect of the grouping stage is to introduce loops into the new procedure. For example, when the grouping stage is applied to the pairs of events shown in figure 3-4, it will group event 1 of the first trace with events 1 and 4 of the second trace. The effect of this will be to introduce the loop shown in figure 3-5.

The final two stages analyze the procedure resulting from the grouping stage, and look for two particular situations—*parallel branches* and *nondeterministic forks*—which provide justification for event generalization at higher levels.

A set of branches of a procedure are parallel branches if they all start and end at the same fork and merge events and the corresponding actions are all potentially generalizable (*i.e.*of the same *type*). If there is a set of parallel branches in the new procedure after

the grouping stage, NODDY attempts to merge the branches into a single branch, using a more powerful action generalization which will look for functional dependencies between the actions in the parallel branches and a pattern occurring earlier in the procedure.

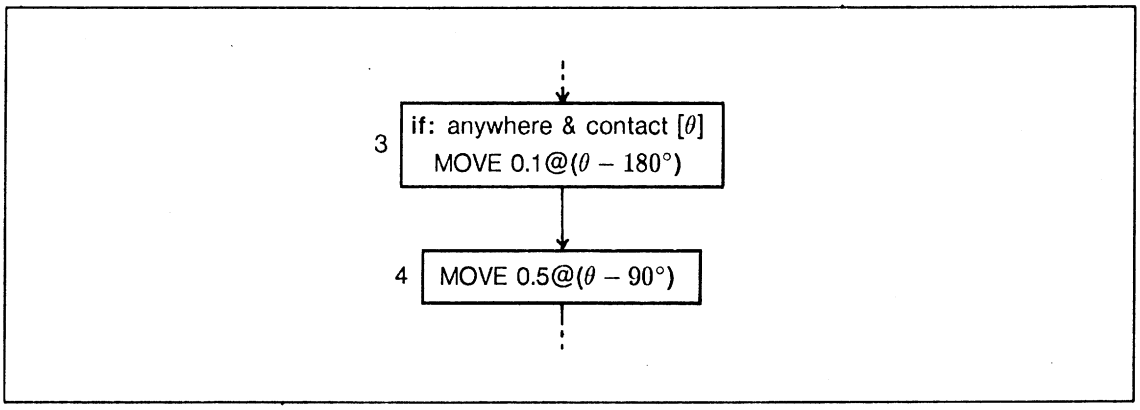


— — Figure 3-6 Procedure with Parallel Branches. — —

The new version of the TURTLE procedure after the third trace, but before the parallel branches stage. The propagation stage could not match events 3 and 3′, nor events 4 and 4′. The two branches are parallel because they start at the same event, and end at the same event, both have 2 events, and the corresponding actions are of the same type

For example, the skeleton, propagation and grouping stages produced the procedure shown in figure 3-6, which contains a pair of parallel branches. The parallel branches stage attempted to merge the branches by generalizing the actions with the the more powerful action generalization. The event matcher found that both pairs of actions could be generalized to actions in which the direction to move was a function of the pattern in event 3/3′. The new, merged branch is shown in figure 3-7. To find such generalized actions, NODDY searches backwards through the procedure from the actions being generalized, looking for the first pattern with a component for which it can find a function relating the pattern component to the action.

A nondeterministic fork is a fork whose branching conditions do not satisfy the



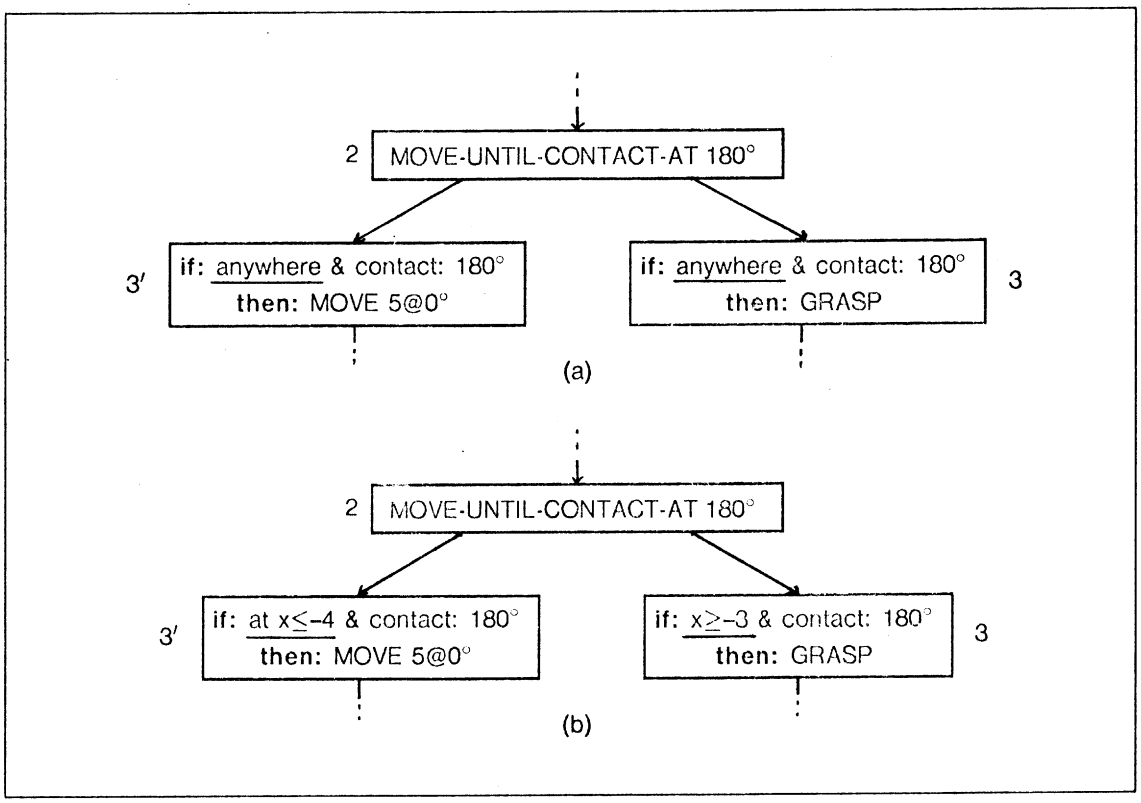
— — Figure 3-7 The Generalized Branch with Functional Actions — —

The actions in the new branch both depend (in slightly different ways) upon the contact direction component of the pattern of the first event in the branch.

determinacy constraint. That is, two of the branch conditions are either identical or they intersect one another—some patterns would satisfy both conditions, but neither condition is a strict generalization of the other. If NODDY finds a nondeterministic fork in the procedure after the parallel branches have been resolved, this is ironclad justification for doing something because nondeterministic forks are invalid in the procedures NODDY acquires and cannot be left in the new procedure. For example, figure 3-8-a shows the nondeterministic fork NODDY found when processing the fifth trace for the RETRIEVE procedure.

NODDY first tries to resolve the conflict by merging the events containing the conflicting conditions. If the event matcher cannot find a generalization of the actions in the two events, as in the case of figure 3-8, NODDY assumes that the conditions have been overgeneralized, and invokes the event matcher again to find more more powerful generalizations of the conditions that do not conflict. In the example from the scenario, the event matcher resolved the conflict by finding the new branch conditions shown in figure 3-8-b. If the event matcher cannot find any conditions that do not conflict, NODDY abandons the trace as impossible to match and waits for another trace that is not as difficult.

NODDY's final procedure is the one that issues from the final stage of nondeterministic fork resolution. This procedure is guaranteed to be a syntactically valid procedure and to cover both the old procedure and the new trace. If the teacher is satisfied with it, he will conclude the training sequence, otherwise he will provide another trace which NODDY will process like the previous one, generating a still more general procedure.



— — Figure 3-8 A Nondeterministic Fork from the RETRIEVE Procedure — —

(a) The two branch conditions of the fork are identical. To resolve the fork, NODDY constructs new conditions, as shown in (b)

Chapter 4
Structure Matching and Generalization

The task of the structure matcher is to match NODDY's current description of the procedure against a new trace and generalize the procedure to account for the trace. There are several different aspects of this matching and generalization:

- Identifying the parts of the trace that can be accommodated by the structure of the current procedure and those parts that cannot.
- Choosing sets of events to generalize. The event matcher, described in chapters 5 and 6, actually constructs the generalized events, but the structure matcher must propose the events to generalize.
- Introducing new loops, forks, and branches into the procedure when necessary.
- Identifying candidate state variables and the actions that may depend on them.

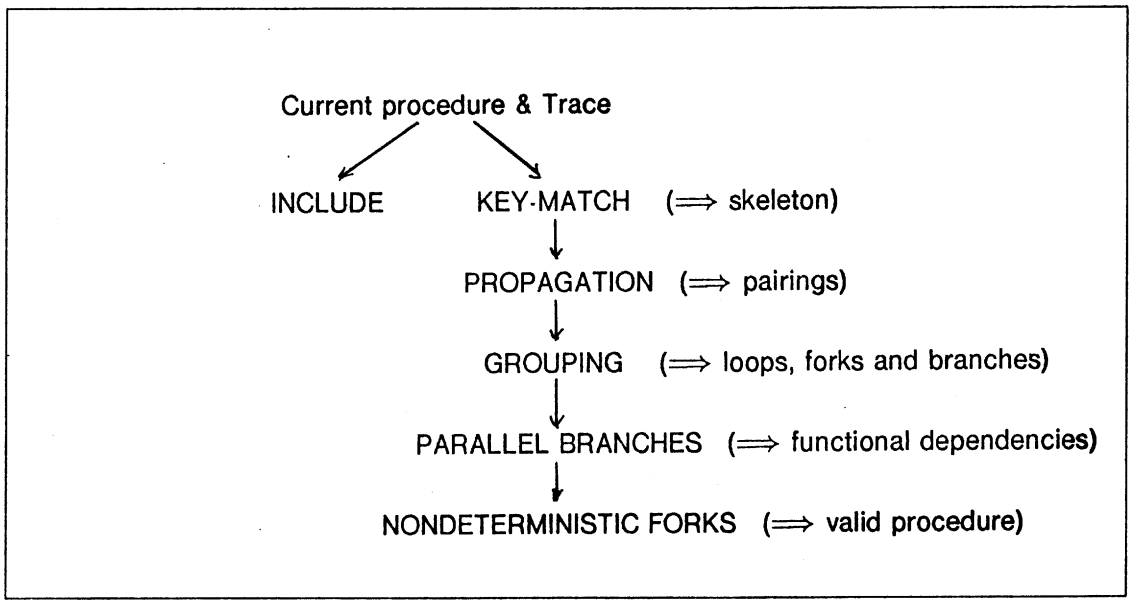
This chapter describes NODDY's algorithms for these matching and generalization tasks. As outlined in chapter 3, the algorithm is structured into several stages. Each stage generates a partial match which provides the basis for the matching and generalizing of the next stage. Figure 4-1 shows the dependencies of the several matching stages and is also a roadmap for this chapter. Each of the following sections describes one of the stages in detail and discusses the nature of the generalization that is carried out by that stage.

4.1 Stage 0: Include

The first stage determines whether the new trace actually adds any new information or whether it is already covered by the current procedure. There is no point in proceeding to the stages that generalize the procedure if it is already sufficiently general.

The include stage essentially executes the current procedure, using the patterns in the trace as the feedback from the world, and checking that the each action specified by the procedure is the same as that of the trace. If any of the actions are different or if a trace pattern does not match any of the conditions at any step, then the procedure does not cover the trace.

NODDY steps through the procedure and the trace together, beginning at the start events of each, attempting to match each trace event to the corresponding procedure event. At a fork in the procedure, NODDY chooses the the first branch whose branch

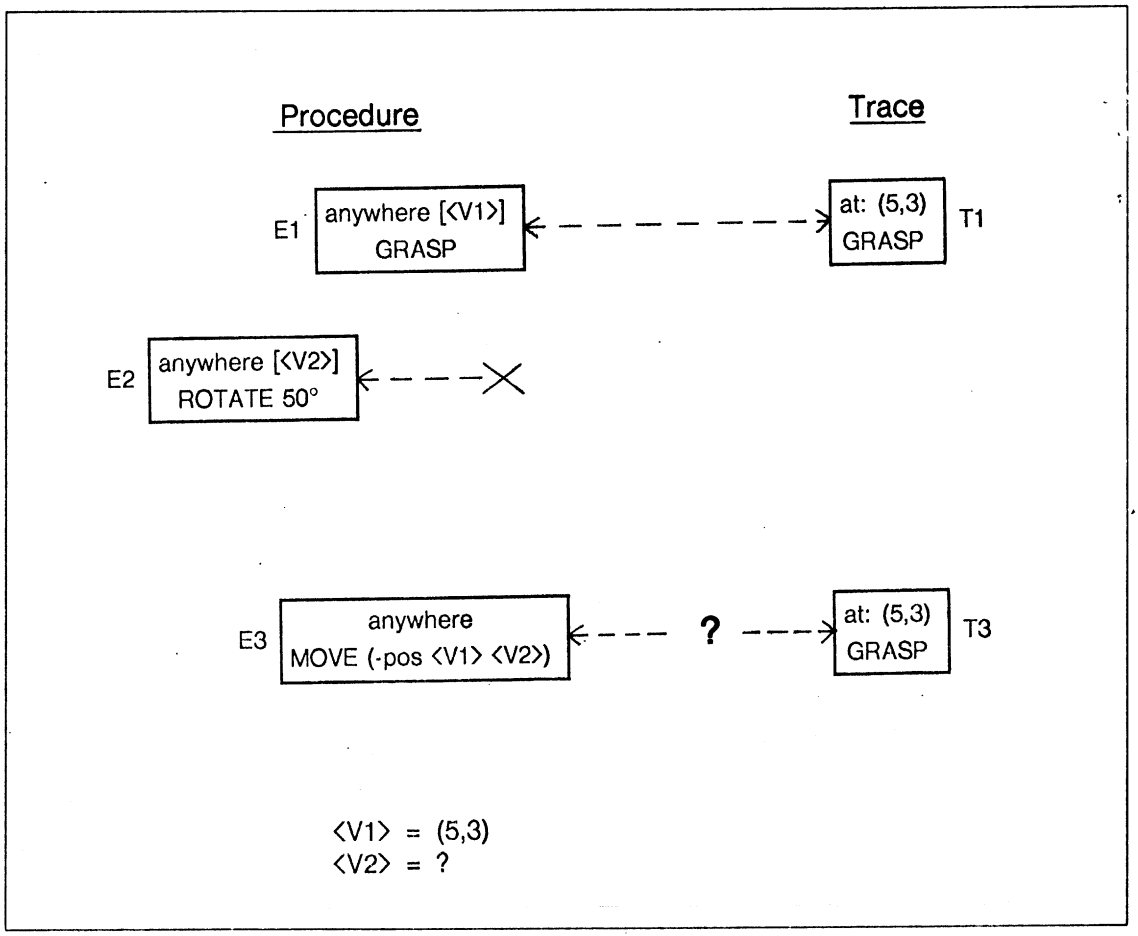


— — Figure 4-1 Overview of the Structure Matcher. — —

This figure shows the relation between the various stages of the structure matcher, and what each stage produces.

condition matches the pattern of the trace event, (the branches are searched according to their priority, if the links are labeled). If there is no branch condition that matches, or if the action of the branch event does not match the action of the trace event, then the trace is not covered by the procedure and the include stage fails. The trace is covered if NODDY reaches and matches the stop event of the trace.

The include stage does not generalize the events, so it asks the event matcher to determine only whether the procedure event is equal to or includes the trace event. The match may fail either if the procedure event is different from the trace event, or if the action of the procedure event is a functional action and it cannot be assigned a value. To find the value of a functional action, the event matcher must find the variables of which the action is a function; determine the procedure events to which the variables refer; then use the patterns of the trace events that have been matched to those procedure events to compute the value of the function (see figure 4-2). If the procedure events to which the variables refer have not been matched to any trace event, then the function cannot be computed, and so the match fails.



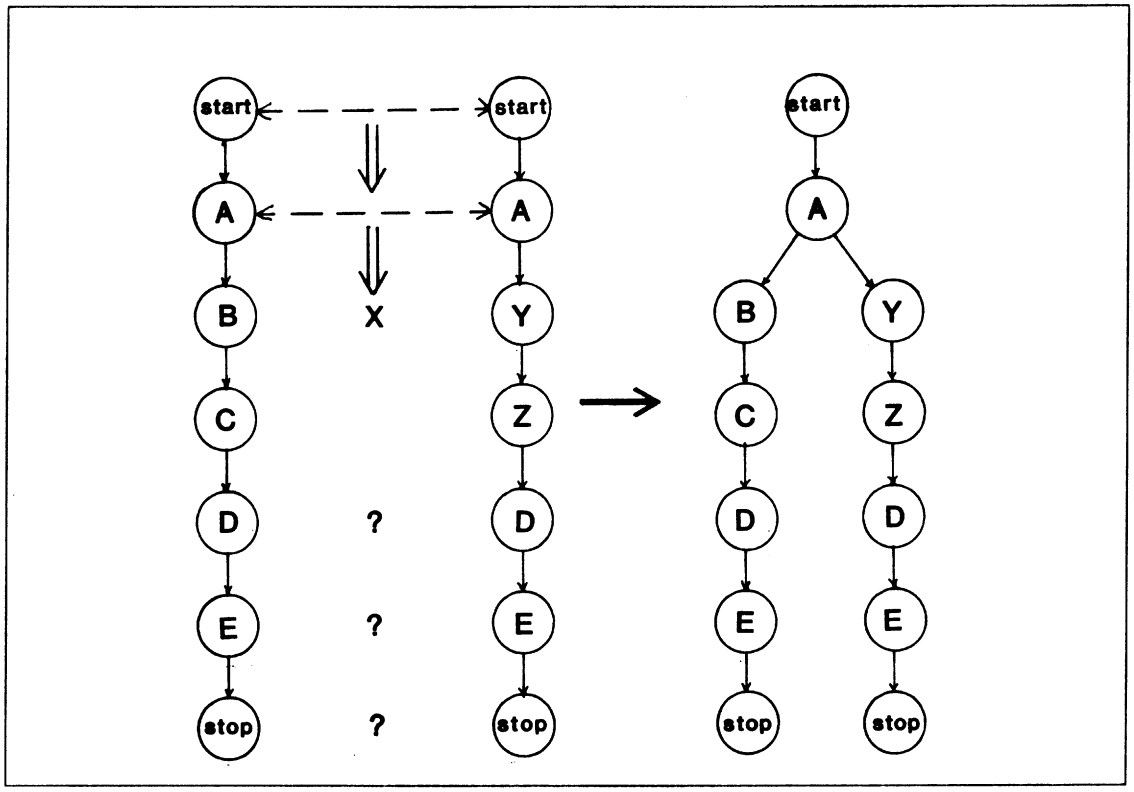
— — Figure 4-2 Matching Functional Actions — —

NODDY has stepped through the procedure and the trace, matching the events as shown, and is trying to match the procedure event E_3 against the trace event T_3. However, the action of procedure event E_3 is a function of variable V_1 that refers to the condition of event E_1 and variable V_2 that refers to the condition of event E_2. E_1 has been matched to the trace event T_1, so NODDY can obtain a value from V_1 from the pattern of T_1, but E_2 has not been matched to any trace event, so NODDY cannot assign any value to V_2. Therefore, NODDY cannot determine the action of E_3 and cannot match E_3 to T_3.

4.2 Stage 1: Skeleton

The include stage finds pairs of procedure and trace events by pairing the start events then stepping through the procedure and trace along the sequence links. The start events should always be paired, and are guaranteed to match, since all start events are identical. The pairs found by stepping through the procedure are justified by the context—two events are only paired when their preceeders have been paired and matched. The propagation stage will use exactly the same principle of stepping along the sequence links to find pairs. It is not sufficient, though, to start stepping only at the start events. For example, if there

is an unresolvable difference between the procedure and the trace near the beginning of the procedure, but the last parts are identical (see figure 4-3), stepping only from the start events will halt when the difference is reached and will never find the appropriate pairing of the rest of the events.

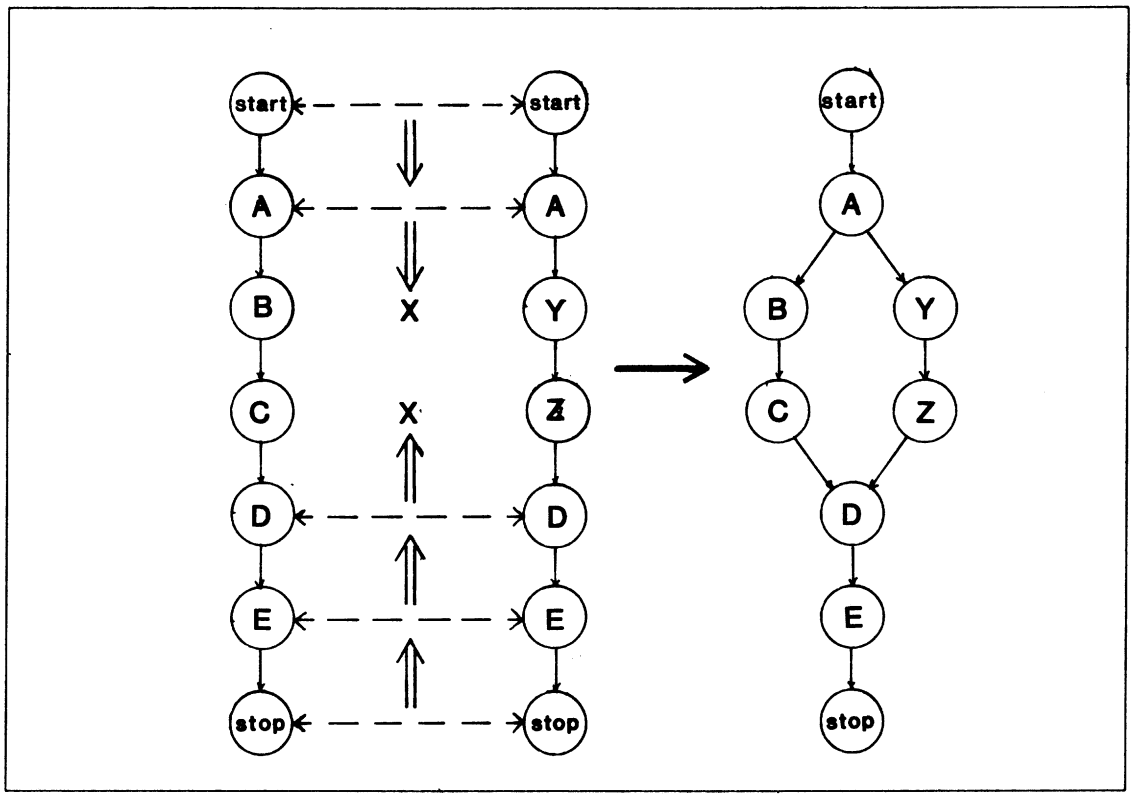


— — Figure 4-3 Blocking the Stepping — —

The trace and procedure have an unresolvable difference near the beginning which blocks the stepping. Therefore, the stepper never finds the identical portions in the last half of the procedure and trace.

For the procedure and trace of figure 4-3, it would be useful to step backwards along the links from the stop events, as in figure 4-4. Key pairs in the middle of the procedure and trace will also be necessary to cope with traces that introduce differences near the beginning *and* end that block the stepping in both directions.

To solve this problem, the skeleton stage finds a set of *key pairs*—pairs of procedure and trace events that partition the matching problem into isolated blocks so that differences in one block do not interact or interfere with differences in the other block. Without this, NODDY would not be able to deal with traces that had more than one significant



— — Figure 4-4 Stepping Backwards from Stop Events. — —

The same trace and procedure from figure 4-3 but zipping up from the bottom

difference from the current procedure. The new events constructed from the key pairs form the skeleton of the new, generalized, procedure. The propagation stage will step through the procedure and trace, starting at these key pairs.

It is important that the key pairs be correct pairings. All the later stages will build on the skeleton of key pairs, and an incorrect pair will generate many other incorrect pairings that will be difficult to undo. Therefore, NODDY requires strong justification before constructing any key pair.

If a procedure event is the only event in the procedure to have some property, and a trace event not only has the same property, but is the only event in the trace with that property, then there is good reason to believe that they occupy the same role in the procedure. When they also match perfectly—they are either the same event, or the procedure event is a generalization of the trace event—NODDY accepts this as sufficient justification for merging them into a key pair. The events that are the only event in the procedure to have some property are referred to as *unique events.*

The properties that NODDY currently considers are the action type of the event, so that if there is only one ROTATE action in the procedure and only one ROTATE action in the trace, the events containing those actions are good candidates for key pairs. The START events always form a key pair since there is always just one START event in each procedure. NODDY cannot always form key pairs out of the STOP events because a procedure may have several STOP events (though a trace may only have one).

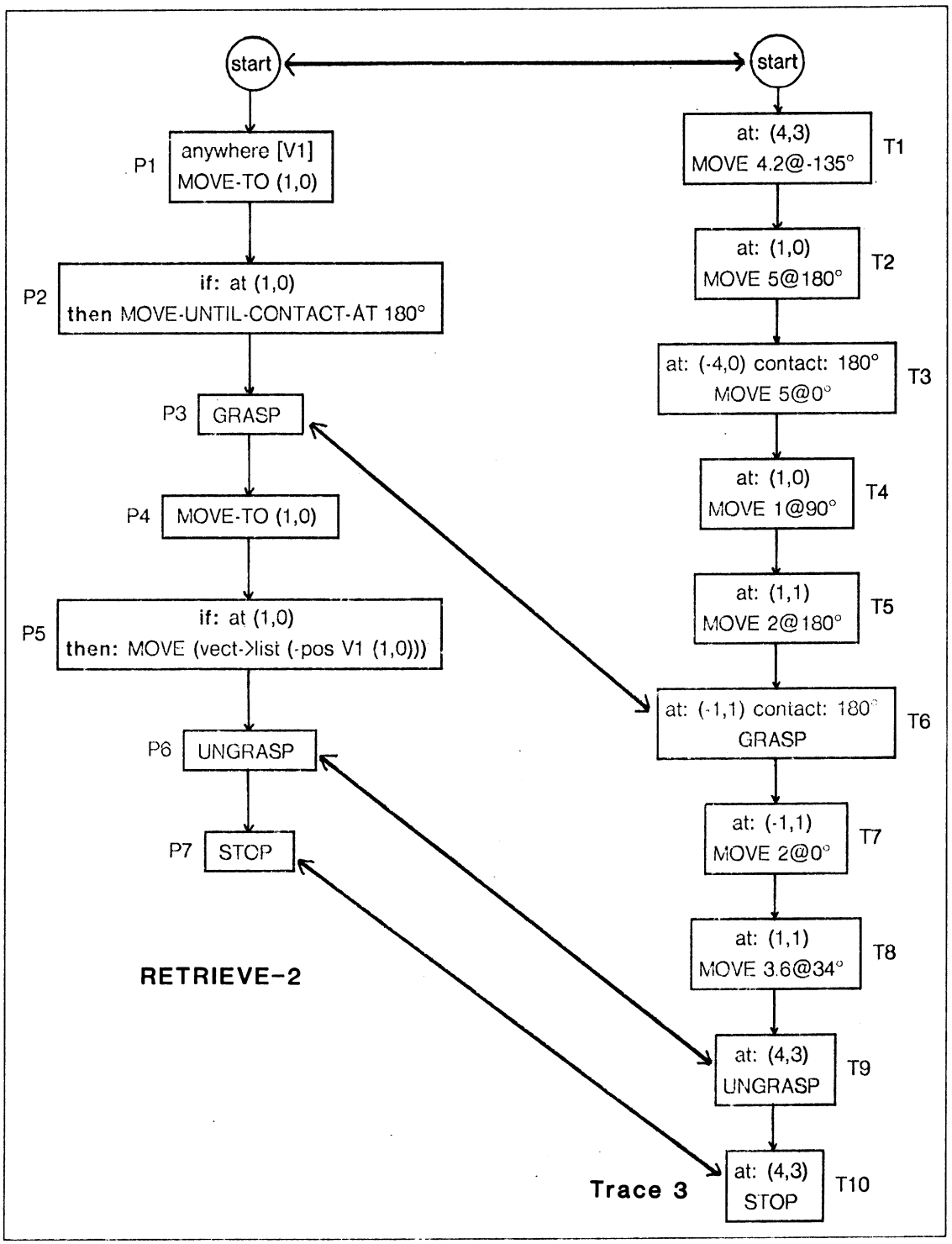
For example, in the second step of acquiring the RETRIEVE procedure of the scenario (matching RETRIEVE-2 to the third trace, see figure 4-5), NODDY found four unique events—the START, STOP, GRASP and UNGRASP events. The GRASP event was crucial in isolating the differences due to looping up to the next row from the differences due to returning from different rows.

In the first step of the skeleton stage, NODDY finds all the unique events of the procedure and trace, indexed by the action class. NODDY invokes the event matcher on the pairs of corresponding events, and rejects the pairs that do not match perfectly. It constructs the skeleton of the new procedure from the key pairs, and hands the skeleton to the propagation stage.

The only justification for key events that the current version of NODDY accepts is the uniqueness criteria to suggest a pair of events, combined with a perfect match of the events. There are several other options that could be tried. One is to extend uniqueness to complete branches of the procedure instead of single events—if a branch is unique in the sense that no other branches have the same sequence of action classes, and there is just one series of events in the trace that match that branch, then NODDY could form key pairs from all the events in the branch. Another option would be to relax the requirement that the trace events in key pairs be unique, so that NODDY could find several key pairs involving one unique procedure event. Experimentation is required to determine what sort of justification is sufficient to prevent NODDY finding spurious key pairs.

4.3 Stage 2: Propagation

The propagation stage puts flesh on the skeleton of the key pairs and introduces the first structural generalizations of the procedure. In this stage, NODDY finds candidate pairs by stepping along the sequence links, just as in the include stage but with three important differences: First, NODDY begins stepping at each of the key pairs, rather than only from the start events, and steps both forward and backwards from the key pairs. Second, it



— — Figure 4-5 Skeleton for RETRIEVE-3 — —

There are four unique events in RETRIEVE-2 and Trace 3—the START, STOP, GRASP and UNGRASP events. The GRASP event isolates one different from the other so that the propagation stage can attempt to resolve each different separately.

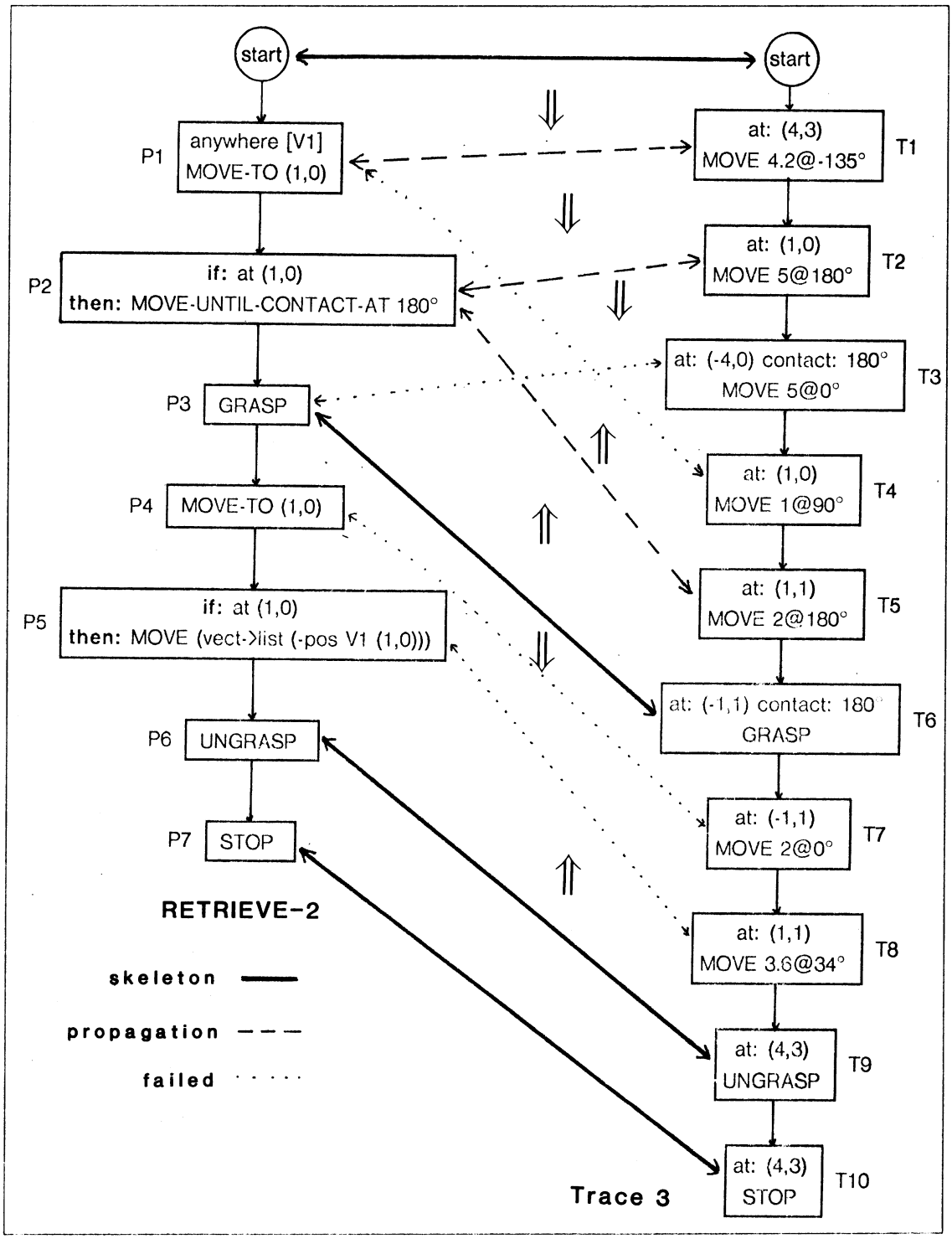
steps along all the branches at a fork (or a merge if stepping backwards) rather than just one branch. Third, it considers first level event generalization rather than insisting on a perfect match.

Finding candidate pairs by stepping through the procedure and trace ensures that there is some justification for matching the procedure and trace events found: at least part of the context of the trace event (its preceeder or succeeder) has already been matched to the context of the procedure event. This suggests that the events play the same role in the procedure and trace. On the basis of this justification, NODDY considers the first level of event generalization in matching the events. It will actually merge the events and pass the pair to the next stage only if the event matcher matches them, either as a perfect match or a first level generalization.

There are two reasons for not considering more powerful generalizations. The first is that, with only two data points, the more powerful generalization methods could find spurious generalizations of many pairs of events, even when there is no underlying commonality to the events. The partial shared context is not sufficient evidence that the events really do have the same role in the procedure to justify an assumption that one of the more powerful generalizations represents a genuine commonality. The second reason is that the more powerful generalization methods are non local in that they must also take into account events in the new procedure other than the ones being generalized. During the propagation stage, NODDY has not constructed enough of the new procedure for these non-local methods to be reliably applied.

NODDY steps through the procedure and trace, starting at each of the key pairs, and stepping in parallel down all the branches at a fork (or merge, if stepping backwards). When stepping from a pair of procedure and trace events, NODDY matches each succeeder (preceeder, if stepping backward) of the procedure event against the succeeder (preceeder) of the trace event. When the event matcher can match the two events, NODDY marks the pair with the new, generalized event that the event matcher returns, and adds the pair to the list of pairs to step from. If the event matcher cannot match the events, it does not continue stepping from them. When there are no more pairs to step from, NODDY collect all the pairs that it was able to match (along with the key pairs) and hands them to the grouping stage. It also collects all the events that it could not pair with anything and marks them to be copied directly into the new procedure.

Figure 4-6 shows NODDY stepping through the procedure and trace of figure 4-5.



— — Figure 4-6 Propagation — —

Propagating from the skeleton of figure 4-5. NODDY tries to match seven pairs, but the event matcher finds generalizations for only three of them.

NODDY found only three more pairs than in the skeleton, two of them involving event P2. Events P4, P5, T3, T4, T7 and T8 were not paired, so they will be copied directly into the new procedure.

The propagation stage must address the same problem with functional actions as the include stage. To reduce the number of events with functional actions that are matched before the events to which their variables refer, NODDY steps from the earlier key pairs before stepping from the later ones. If any events fail to match for this reason, they are reconsidered after all the other pairs have been found.

4.3.1 Generalizations in the Propagation Stage

The set of new events produced by the propagation stage implicitly contains four types of generalization to the current procedure.

- The unpaired trace events represent new branches to be added to the procedure. For example, events T7 and T8 in figure 4-6.
- The pairs for which the event matcher had to create a generalization represent a generalization of the branches already present in the procedure. For example, RETRIEVE-2 (the procedure in figure 4-6) has just one branch which was generalized from the first two traces.
- Multiple pairings involving a single procedure event represent a loop in the procedure, which must be added if a loop was not already present. For example, in figure 4-6, P2 has been paired with both T2 and T5. This will introduce a loop into the new procedure.
- Multiple pairings involving a single trace event may either represent a loop, as in the previous item, or represent consolidation of several existing branches into a single, more general branch.

These generalizations are only implicit in the pairs and unpaired events from the propagation stage. It is the task of the grouping stage to make them explicit.

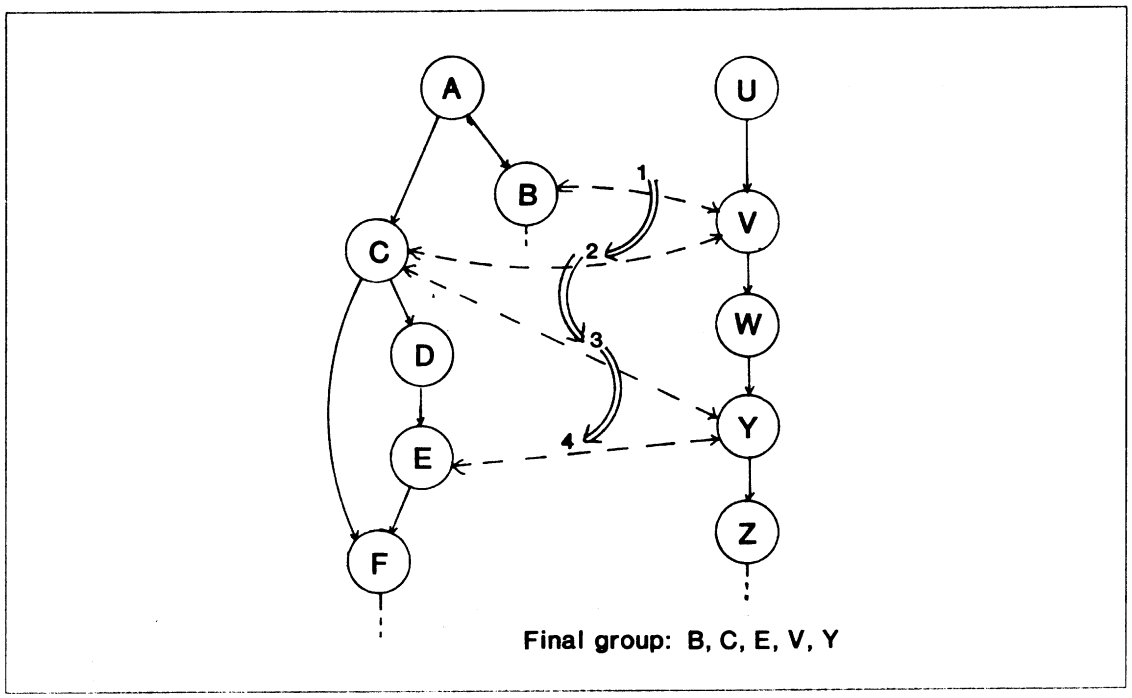
4.4 Stage 3: Grouping

The set of new events generated by the propagation stage may not be a coherent procedure because the propagation stage generates a new event for each two procedure and trace events that it pairs, regardless of whether either of the events has been paired with

other events also. The grouping stage will remove this incoherence by grouping all the overlapping pairs involving the same procedure or trace event, and either replacing them by a single generalization, or eliminating some of the pairs to remove the overlap.

The grouping stage analyzes the set of pairings produced by the propagation stage to find groups of pairs involving the same procedure and/or trace events. It will attempt to merge all the events involved in the group into a single new event. The new event will replace all the pairs out of which it was created.

NODDY finds groups of pairs by picking one pair, finding the pairs that share a procedure or trace event with the first pair, then the pairs that share an event with one of these pairs, and so on until there are no more pairs. Figure 4-7 shows a group involving 5 events.



— — Figure 4-7 Forming A Group from Pairs — —

Some procedure and trace events that have been paired by the propagation stage. The grouping stage starts with one of the pairs (*B—V*) and follows the pairings through the events, as shown by the numbering on the links, to find all the events involved in the group.

Having found a group, NODDY then attempts to merge all the procedure and trace events of the group into a single new event. Currently, it does the same event matching as

in the propagation stage. That is, it matches just two events at a time and only considers first level event generalization. However, there are several differences from the propagation stage which could provide justification for more powerful event generalization, as will be discussed below. Currently, NODDY does not exploit this justification.

The event matcher matches only two events at a time. The grouping stage creates a new event by passing the events in the group, one at a time, to the event matcher which incrementally constructs a new, generalized event. This is a standard generalization from examples task, and, as usual, the order in which the examples (the events of the group) are presented is important. If presented in the wrong order the event matcher may work itself into a corner by constructing the wrong generalization from the first events. The order that NODDY currently chooses is to match all the procedure events before the trace events, and to start with the procedure event that has the most general action. The rationale for this is that it is more difficult to merge the most general action once the other events have been matched and generalized because the condition is likely to be generalized and information needed for action generalization lost. Chapter 5 will discuss this in more detail.

If the event matcher is able to create a generalization of all the events in the group, NODDY replaces the events corresponding to the pairs from which the group was constructed by the generalized event.

If the event matcher cannot find a generalization of all the events in the group, then NODDY must split the group into smaller groups by retracting some of the pairs found by the propagation stage. The event matcher will construct a generalization of as many of the events in the group as it can. NODDY retracts any pair that involves one event that did match the generalized event, and one event that did not. It removes from the group (but does not retract) any pair of which neither event matched the generalized event and looks again for new groups involving these pairs.

None of the examples given to NODDY so far required this backtracking—all the groups that were found could be merged with no conflicts. This suggests that the justification required for the propagation stage was sufficiently strong to prevent spurious pairs.

The grouping stage produces a coherent new procedure by collecting all the new events corresponding to the groups, the pairs which were not in groups and the procedure and trace events that were not paired at all. It connects all the new events by succeeder

links by merging the succeeder links of the procedure and trace. That is, a new event E_2 will be a succeeder of E_1 if any of the procedure or trace events from which E_2 was constructed is a succeeder of any of the procedure or trace events from which E_1 was constructed.

The new procedure may contain new loops, branches and forks. Section 4.4.2 discusses loop introduction, which is a byproduct, rather than an explicit operation, of the grouping stage. The new procedure may have unnecessarily many branches, and the **parallel branches stage** will look for branches satisfying certain criteria that can be merged with more powerful action matching. Furthermore, some of the forks may not be syntactically valid forks because the propagation and grouping stages are strictly local in forming new events and do not check the interaction with other events that causes non-deterministic forks. The **nondeterministic forks stage** will check and correct any invalid forks.

Figure 4-8 shows the new procedure after the grouping stage has been applied to the results of the propagation stage shown in figure 4-6.

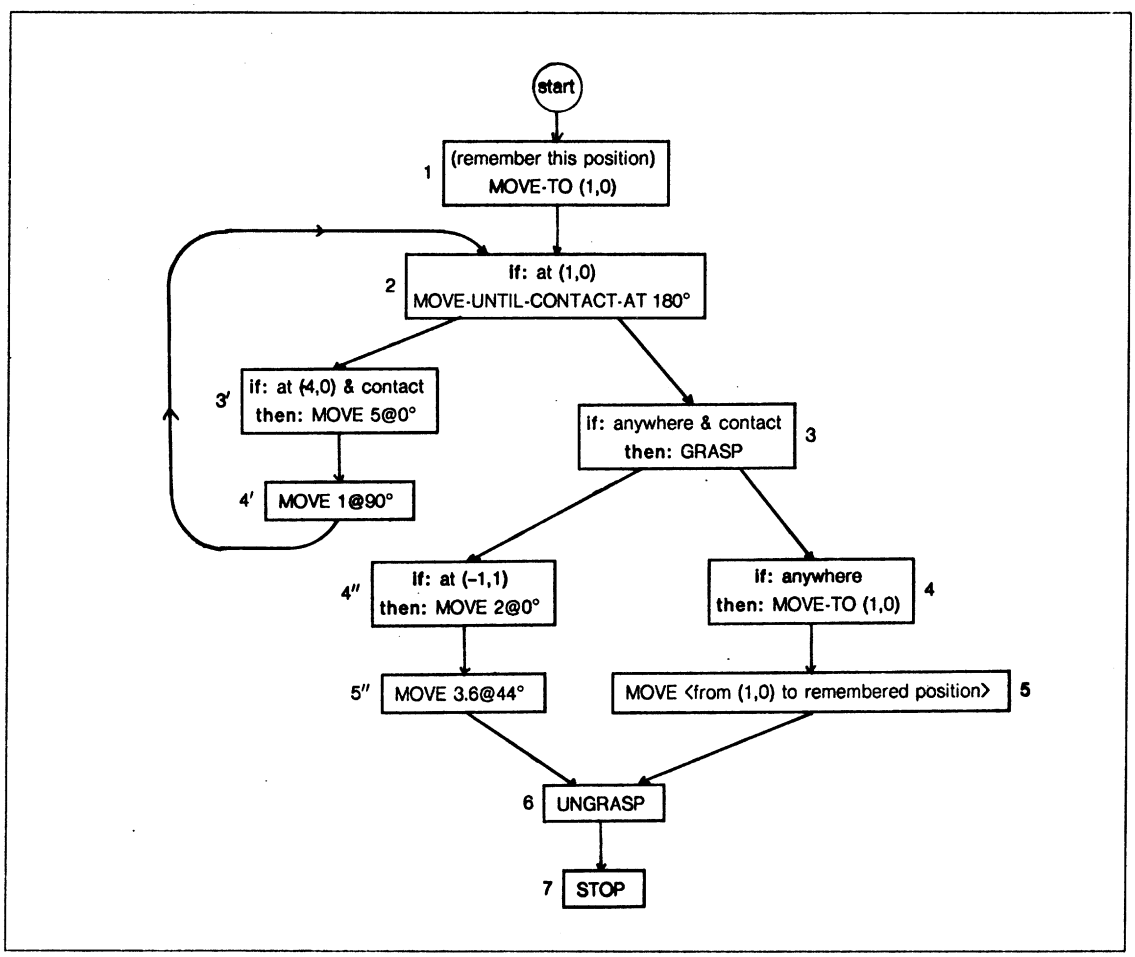
4.4.1 Event Generalization in the Grouping Stage

Currently, NODDY does exactly the same level of event generalization in the grouping stage as in the propagation stage. However, there are two differences which could allow more powerful event generalization.

The first difference is that the grouping stage matches more than two events into a single event. For the action generalization, this means that there are more "data points" which might provide a basis for finding local functions when the actions could not be matched using the action hierarchy. Because the structure of the rest of the procedure is not fixed until the grouping stage is complete, there would not be sufficient justification to consider non-local functions.

The second difference is the nature of the conditions of the events to be matched. The propagation stage always matches a procedure event against a trace event and the trace event always contains patterns, not generalized conditions. The grouping stage, on the other hand, may match several procedure events, as well as trace events, which involves matching generalized conditions. It is not as obvious what sort of condition generalization NODDY should do in this case.

There are three classes of conditions in procedure events: primitive patterns, which represent no generalization at all; the always-satisfied condition, which means NODDY has



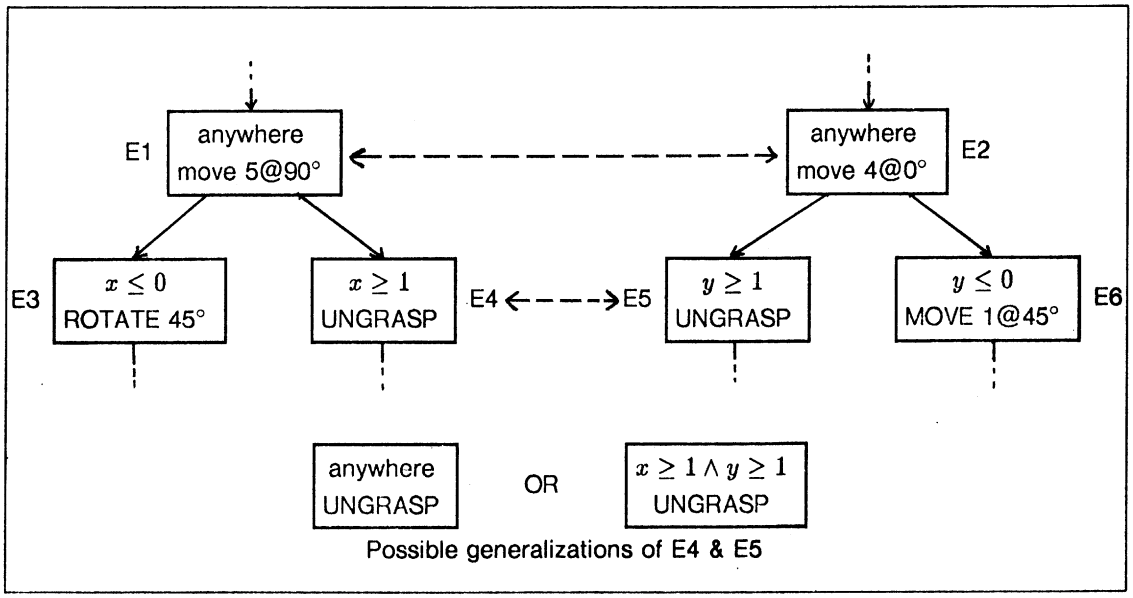
— — Figure 4-8 Results of Grouping — —

Procedure RETRIEVE-3 is the result of the grouping stage applied to the pairings of events in RETRIEVE-2 and trace 3 found by the propagation stage (figure 4-6). The grouping has generated the loop by combining events P2, T2 and T5 into the new event 2.

only seen positive examples of the condition; and any other condition—more complex (and also more specific) than the always-satisfied condition. The more complex conditions only occur as the branch condition of some fork, where NODDY had to find a more complex condition to keep the branch conditions disjoint. As will be described in section 4.6 and chapter 6, NODDY constructs these conditions by using the other branch conditions as negative examples. To do this, NODDY must consider all the branch conditions at once, in contrast with the propagation stage which considers only the events being merged into the new event.

The problem arises in matching a specialized branch condition to some other condition,

whether a pattern from the trace or a condition from the procedure. The grouping stage could either find a local generalization of the two conditions, ignoring the other branch conditions, or a less local generalization by taking into account the other branch conditions and finding a generalization of the two conditions that is disjoint from the other branch conditions.



— — Figure 4-9 Generalizing Conditions — —

Events $E4$ and $E5$ are part of a group for which the grouping stage is trying to find a generalization. If the generalization is strictly local and ignores events $E3$ and $E6$, it will find the generalization on the left, which will cause NODDY to do more work later on. If the generalization takes $E3$ and $E6$ into account, it might find the generalization on the right, which will save work later on.

For example, in the situation of figure 4-9, the grouping stage is trying to construct a generalized event for a group that includes event $E4$ and $E5$. The events are branch events of different forks. If NODDY does the local generalization, ignoring the other branch events $E3$ and $E6$, it will generalize the conditions of $E4$ and $E5$ to [anywhere]. The problem with this is that it throws away all the work that went into finding conditions that did not conflict with events $E3$ and $E6$, and NODDY will have to do the work over again in a later stage to find a new condition for the new that does not conflict with events $E3$ and $E6$. If NODDY could take $E3$ and $E6$ into account while generalizing $E4$ and $E5$, it might find the generalization $[x \geq 1 \wedge y \geq 1]$.* so that the later stage would not have

* Whether this is the correct generalization would depend on the past values of the conditions.

to do the work all over again.

A more complex grouping algorithm, would take the other branch conditions into account, and generalize the two conditions accordingly. In particular, if a group includes one branch event of a given fork, it would determine whether the group includes all the other branch events as well. If so, then the appropriate generalization will be the always-satisfied condition, because all the branches will be merged and the fork will disappear so there will be no conditions that have to be disjoint. If, on the other hand, the group does not include all the branch events of the fork, then NODDY should find a generalization of the conditions that is disjoint from the remaining branch conditions, as in figure 4-9.

Currently, NODDY is not this sophisticated, and throws away most of the information and recomputes it in a later stage. The details of the condition generalization are in chapter 6.

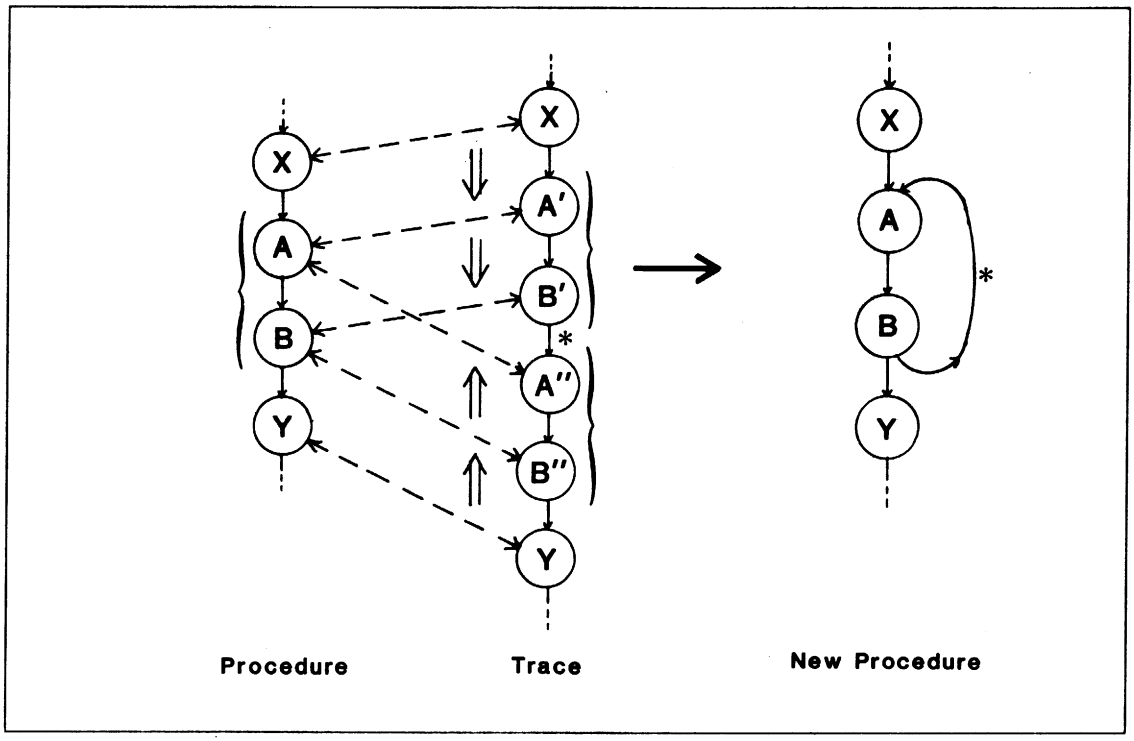
4.4.2 Introduction of Loops

The introduction of loops is a byproduct of the propagation and grouping stages—loops "fall out" at the end of the grouping stage without NODDY explicitly representing or looking for them. Matching two trace events to a single procedure event implies that the two trace events are really the same event so that the goal procedure must have a loop in order to repeat the event.

NODDY can infer a loop when the current version of the procedure and the new trace each contain a different number of iterations of the loop, and the key event stage and/or the propagation stage is able to pair the procedure and trace events at the entry to the loop and the events at the exit. Figure 4-10 shows such a case when the procedure has one iteration of the loop and the trace has two iterations. The propagation stage will then propagate forward from the entrance of the loop (the X events) and backwards from the exit of the loop (the Y events), pairing the events of the procedure iteration with with the events of both the first and second trace iterations. The grouping stage will then merge all the triples of events which will result in a loop.

Note that the loop is a "DO UNTIL" loop—execution will iterate around the loop as long as the condition of event A is satisfied after the action of event B. When the condition of event Y is satisfied instead, execution will exit the loop into event Y. NODDY cannot

Chapter 6 will discuss this in more detail



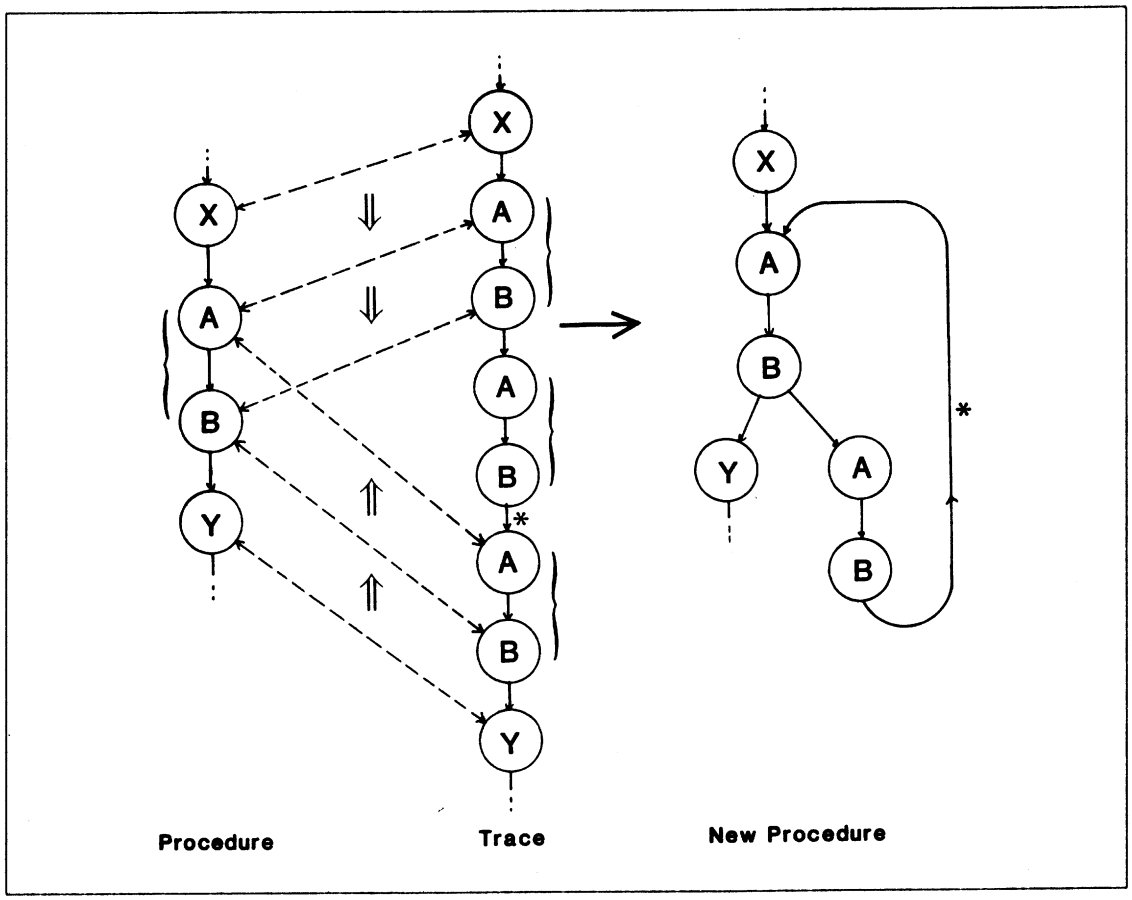
— — Figure 4-10 Forming Loops — —

The procedure (on the left) has one iteration of the loop (A — B ···)and the trace has two iterations. The propagation stage propagates forward from the matching X's and backward from the matching Y's, to find the pairs A - - - A', B - - - B', and A - - - A'', B - - - B''. The grouping stage merges the A events into a new A, merges the B events into a new B, and merges all the succeeder links. The succeeder link between B' and A'' ($*$) is the link that forms the loop back from the new B to the new A.

currently infer any other kind of loop (such as a counted loop). The reasons for this are given below.

If the procedure that the teacher is teaching NODDY has a loop of k events, and the current version of the procedure contains n iterations of the loop and the trace contains m iterations, then this algorithm will infer a loop of $|m-n|\,k$ events. For instance, if the trace in figure 4-10 had had 3 iterations of the 2 event loop A - B, the new procedure would have had a loop with $|3-1| \times 2 = 4$ events, as in figure 4-11.

This means that NODDY will not necessarily find the shortest loop, but rather, the most conservative loop—it will never assume that the loop can be broken into smaller pieces than the evidence demands. For example, in the situation of figure 4-11 the sequence A - B occurs once in the procedure and three times in the trace. The resulting loop generates any odd number of repetitions of A - B. It is currently important that



— — Figure 4-11 An Odd Loop — —

Similar to figure 4-10, but the trace has three iterations instead of two. The final loop is not the shortest loop, but is a more conservative generalization than the two event loop

NODDY generalize conservatively in this way rather than guess that the loop could be further consolidated because it currently has no way of undoing an overgeneralization of this kind.

This is not entirely satisfactory, and future work should address the issue of inferring loops in a more general way, rather than letting them "drop out" of the matching. This would involve explicitly representing loops and applying generalization methods that exploit the constraints on iterative constructs.

At first glance, it looks as though NODDY inferred the loops in the TURTLE and RETRIEVE procedures (figures 1-13 and 4-8) from 0 iterations and 1 iteration, which would be a very unconservative generalization. In fact, the loop is a little odd in that the loop is entered and exited by the same event (the MOVE-UNTIL-CONTACT-TOWARD event).

The loop contains 3 events (the entrance/exit and the two MOVE's) but the final iteration of the loop only passes through the MOVE-UNTIL-CONTACT-TOWARD event. The first trace therefore contains half an iteration of the loop, and the second trace contains one and a half iterations, so that inferring the loop is not as unjustified as might appear.

4.5 Stage 4: Parallel Branches

The parallel branches stage invokes a more powerful action matching method than used in the previous stages to merge some events that should be merged but could not be matched using only the action hierarchy. Part of this action matching may also introduce state variables into the new procedure. Because its generalization space is larger, the second level action generalization is more likely to find spurious generalizations. The structure matcher must therefore obtain stronger justification for invoking the second level action matcher than it did in the propagation and grouping stages.

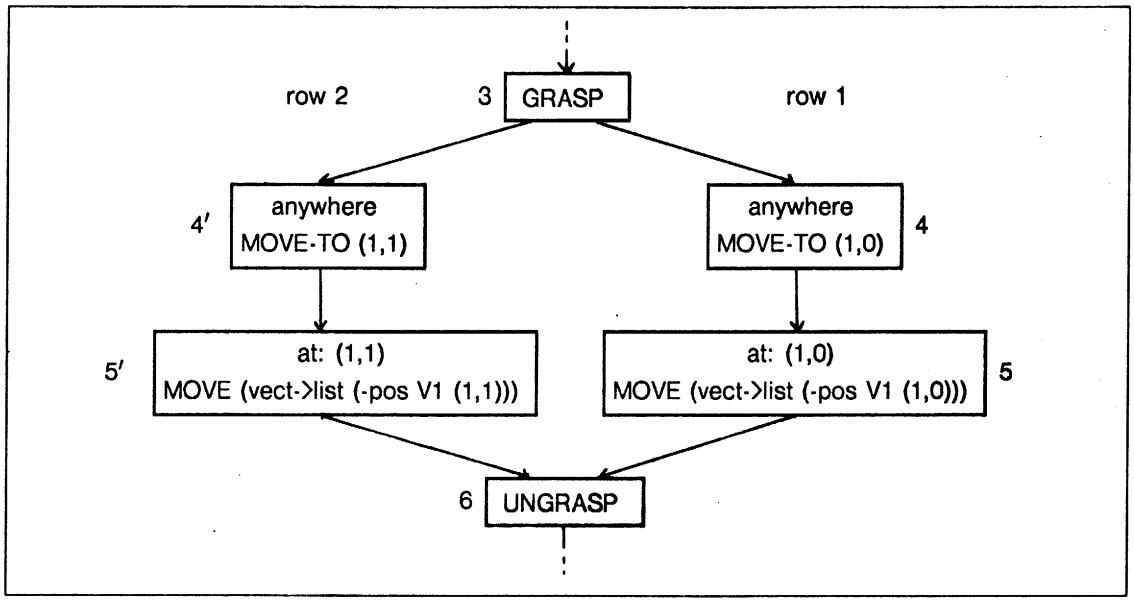
The prime component of this justification is contextual. NODDY searches for a particular construct—parallel branches—in the procedure produced by the grouping stage. Two branches are parallel if:

- They both start at the same fork;
- They both end either at the same merge or at a STOP event;
- They have the same number of events;
- The conditions of the corresponding events in the two branches match;
- The actions of the corresponding events in the two branches match except for constant values in the parameters.

Figure 4-12 shows the pair of parallel branches generated at one stage during the match of RETRIEVE-3 and the fourth trace, shown in more detail in the scenario (section 1.1). The branches satisfy each of the five conditions above.

These properties are strong (syntactic) evidence that the two branches have the same role in the procedure—that they perform the same function. NODDY uses this as justification for trying to merge the branches using a more powerful action generalization method than it used in the earlier stages.

NODDY attempts to merge the branches by merging all the corresponding events. If the actions cannot be matched using the action hierarchy, NODDY appeals to the function inducer to find a generalization that expresses the action as a function of some

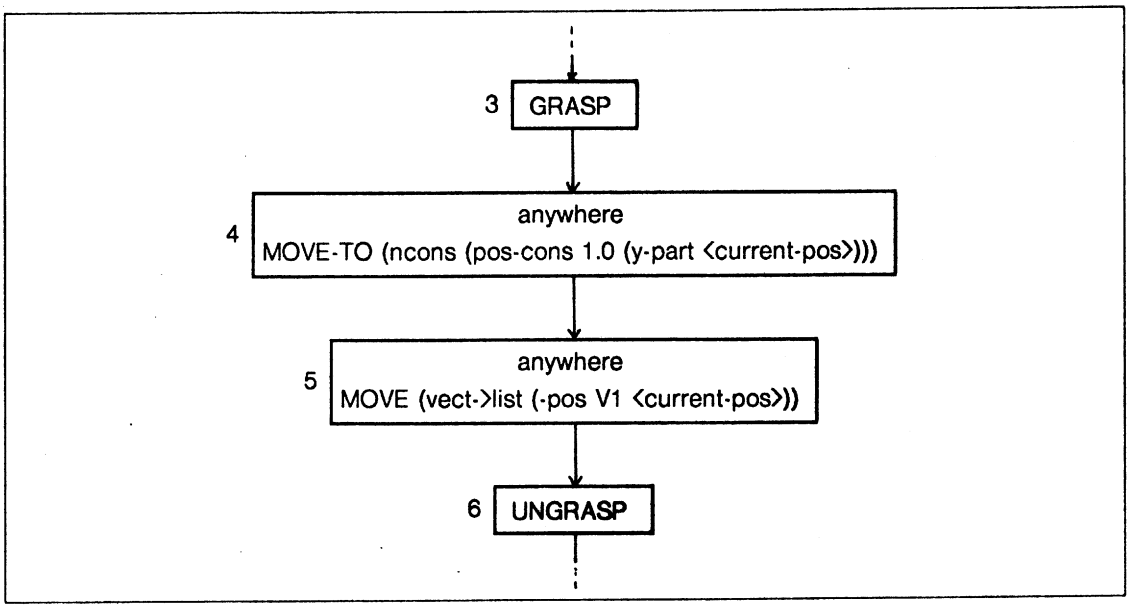


— — Figure 4-12 Parallel Branches — —

A fragment of the new procedure generated from RETRIEVE-3 and trace 4, part way through the parallel branches stage. The two branches are parallel because they both start at the same fork (event 2), end at the same merge (event) and the corresponding events match. Events 4 and 4′ differ only in the value of the parameter of the `MOVE-TO` action. The conditions of events 5 and 5′ can be generalized to `[anywhere]`, and the actions differ only in the constant in the function expression.

previous pattern. As long as NODDY can find generalizations for *all* the corresponding events of the new procedure, it constructs a new branch out of the generalized events and installs it into the new procedure in place of the parallel branches. If NODDY had to appeal to the function inducer to match any of the actions, it may also have to create new state variables for the procedure. If any of the actions cannot be matched, using either the action hierarchy or the function inducer, then the merging of the parallel branches is abandoned.

For example, NODDY was able to merge the parallel branches of figure 4-12 to form the single branch shown in figure 4-13. To do this, NODDY had to find a function for each pair of actions. The action of event 4 is a function of the current position, and the action of event 5 is a (different) function of both the current position and the initial position (<*V1*>). The actions of the events out of which event 5 was formed were already expressed in terms of a function of the initial position. To merge them, NODDY had to determine that the functions were almost identical, and that the difference could be expressed as a function of the current position. The details of the function induction are



— — Figure 4-13 Merged Branches — —

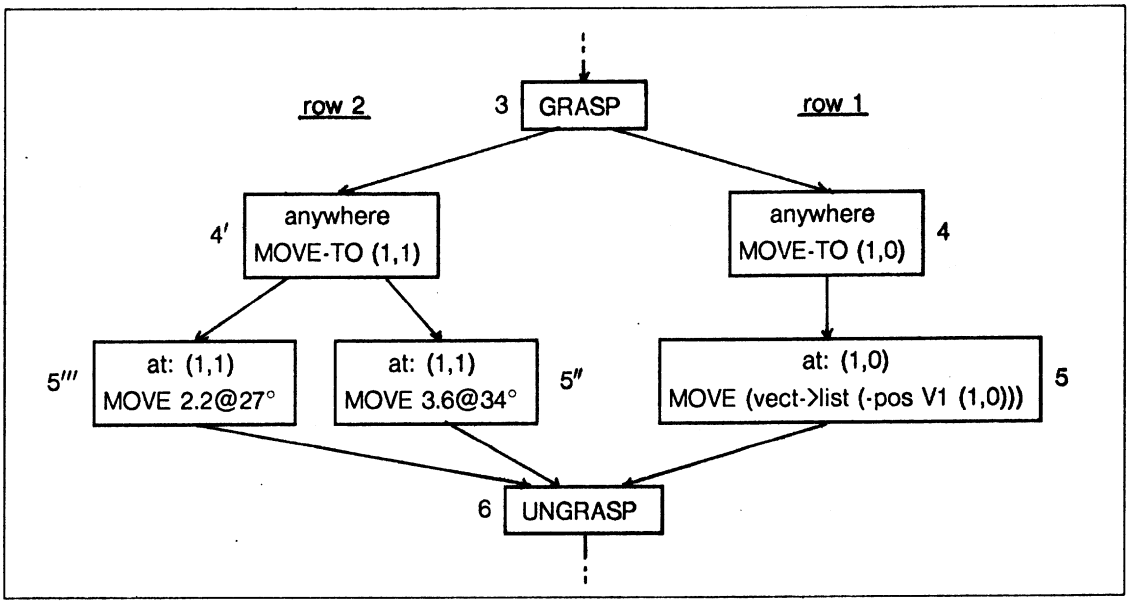
The branches of figure 4-12 after they have been merged.

in section 5.2.

There may be several sets of parallel branches in a new procedure after the grouping stage, and it can be the case that merging one set of parallel branches creates a new set of parallel branches. To allow for this situation, NODDY finds one set of parallel branches, attempts to merge them, then finds another set, and repeats the process until no new sets of branches can be found.

This was actually the case in matching RETRIEVE-3 and the fourth trace. Figure 4-14 shows the emerging procedure one step before figure 4-12 above. The left hand branch contains a fork followed by two parallel branches, each with one event. When NODDY merged these two branches by finding a generalization of the two actions, it created the parallel branches of figure 4-12.

The key new step of the parallel branches stage is the second level action matching which is invoked when there is a set of corresponding actions in the parallel branches that cannot be matched using the action hierarchy. The actions differ in one constant, either the list of parameters, or one constant in a function expression. If y_i is the set of the different constants, NODDY must find some function f and one component of the condition of an event (the *source component*) such that $y_i = f(x_i)$ where the x_i are

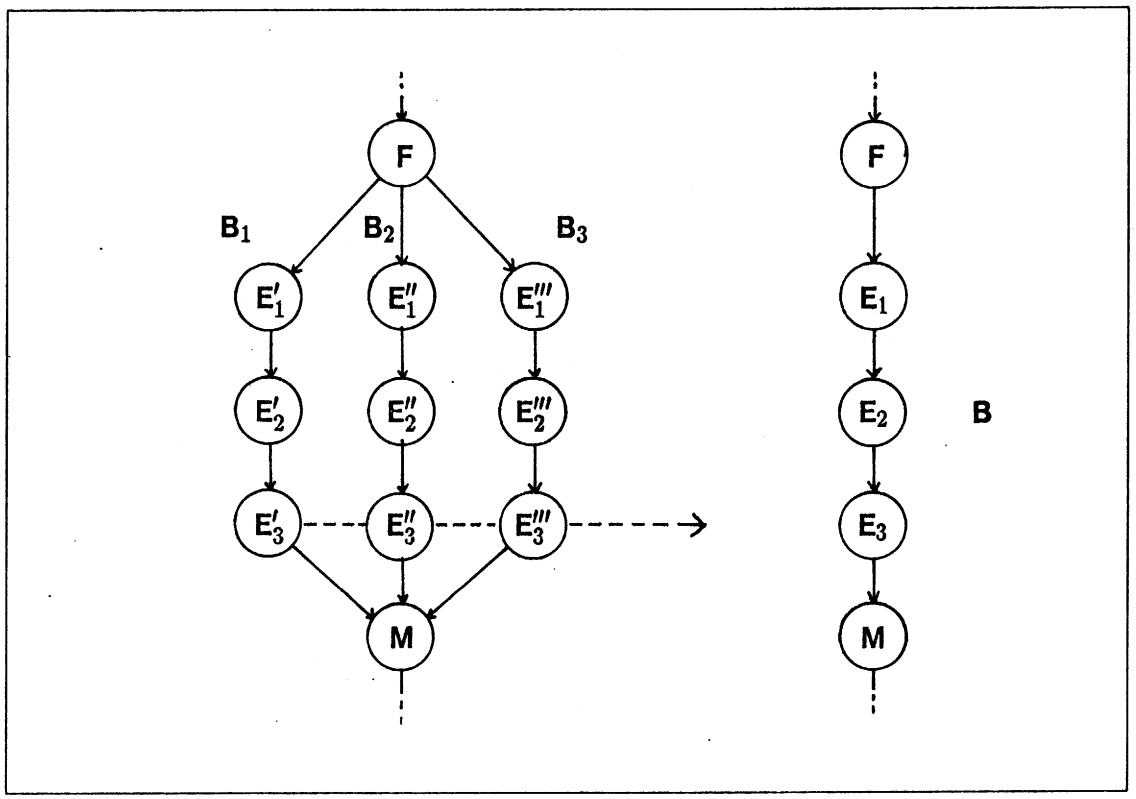


— — Figure 4-14 Nested Parallel Branches — —

This is the state of the emerging procedure just before the state shown in figure 4-12. Events 5'' and 5''' constitute two parallel branches. When NODDY merges them, the result is the two parallel branches 4'—5' and 4—5 of figure 4-12. NODDY found a generalization of events 5'' and 5''' by discovering that the actions were a function of the initial position (<*VI*>).

the past values of the component—the pattern values from which the source component was constructed, (the actions and the past values of conditions pattern values are marked with the traces from which they came, so that NODDY can establish the appropriate correspondence between the x_i and the y_i). Finding the function f is a standard function induction problem, except that NODDY has only the y_i and must find both a function f and a set of x_i. Chapter 5 describes the function induction algorithm for finding f; here we only describe the search for the source component, since that depends upon the structure of the procedure.

Figure 4-15 shows three parallel branches being merged into a new branch, and three events E_3', E_3'' and E_3''' being merged into a new event E_3. Assume that the action matcher cannot match the actions of E_3', E_3'' and E_3''' using the action hierarchy. It will therefore search for a functional action. The source component for the function must occur earlier in the procedure than the event E_3 (when executing the action, NODDY must have a value for the source pattern in order to compute the parameters of the new action). There are three distinct places the source event could be:



— — Figure 4-15 Inferring Functions in Parallel Branches — —

B_1 B_2 B_3 are parallel branches which NODDY is attempting to merge into the new branch B on the right (which will be just part of a larger branch). NODDY is trying to merge E'_3, E''_3 and E'''_3 by finding a functional dependence between the actions of the three events and an earlier condition. It will first consider the condition in E_3, which is the generalization of the conditions of E'_3, E''_3 and E'''_3. It will then consider the conditions of E_2 and E_1, then of earlier events—the fork event F and earlier.

- The condition of event E_3 itself (which would make the function a local function).
- One of E_1, or E_2, *i.e.*, an event of the new branch being constructed.
- An event earlier in the procedure the parallel branches (F or earlier)

NODDY searches these three places in order: first considering E_3; next, stepping back along preceeder links to the events in the new branch, then starting at F, stepping back along preceeder links in a breadth first fashion through the procedure (considering all the preceeders of an event before considering the preceeders of any of them). If the parallel branches are part of a loop, it will be possible to step back to the parallel branches themselves or to events that occur after E in the loop. Currently, NODDY stops stepping if it gets back to the parallel branches B_1, B_2 or B_3, but does not do more careful pruning of other events in the loop. At each event that it reaches, NODDY hands the candidate

source event, along with the actions to be merged, to the action matcher. The search ends at the first source event for which the action matcher finds a function to merge the branch events.

A consequence of this breadth first search is that NODDY is biased toward finding the most recent source event for the action to depend on, rather than the one which gives the simplest function. For example, if the action in event E_3 can be expressed either as a complex function of the position in event F, or as a simple function of the position in the earlier event G, NODDY will always choose the later event (F), even if the function is more complex. There are two reasons for this bias. One is that the resulting procedure is more robust in that later changes to the procedure are less likely to come between the source event and the action that depends on it. The second reason is that the search is more efficient—NODDY has to consider fewer candidates than it would if it had to find the best possible candidate.

A second consequence of the current search algorithm is that NODDY does not handle functions for actions in loops very well. In particular, it can handle actions that depend upon a variable set earlier in the current iteration, but not actions that depend on a value set in the previous iteration. This would require explicit reasoning about the loop and initializing the variable when first entering the loop. More powerful loop generalization is an area for future work.

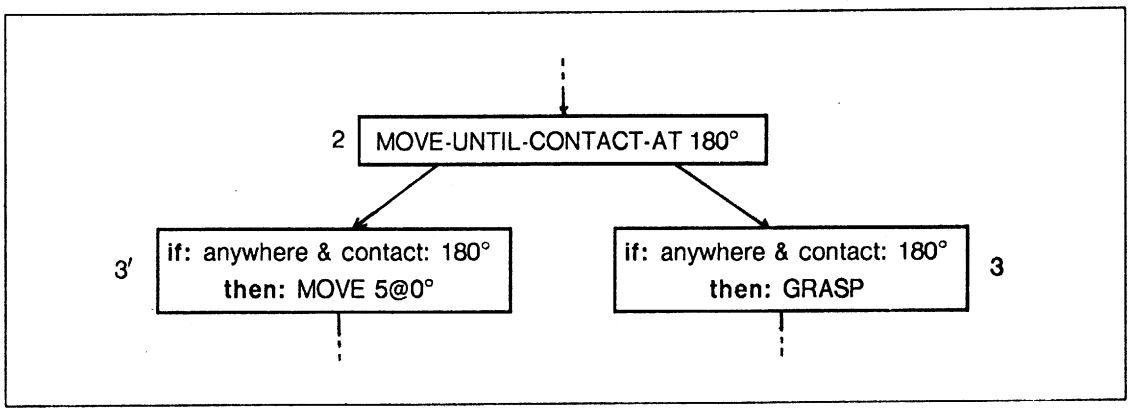
When NODDY is able to find generalizations of all the events of the parallel branches, it replaces the parallel branches with the new branch consisting of all the generalized events. It must also mark the source components of the functions it discovered as state variables.

4.6 Stage 5: Nondeterministic Forks

The final stage of the structure matcher ensures that the new procedure is a valid procedure. In particular, it checks whether all the forks are deterministic, and attempts to fix any that are not. A fork is deterministic when, for any given pattern, the choice of which branch to take is unambiguous. That is, each pair of branch conditions must either be disjoint, or one condition may be a special case of the other and must be marked as having a higher priority.

The new procedure before this final stage has no priority ordering on any of the forks, and no guarantee that the branch conditions will be disjoint. This is a result of the local nature of all the previous matching which only took into account the events being

matched, not the surrounding events. NODDY checks each fork in turn by matching each pair of branch conditions, to determine whether they are disjoint, they conflict, or one condition is a generalization of the other.



— — Figure 4-16 Nondeterministic Fork — —

The nondeterministic fork from the final trace for the RETRIEVE procedure. The left branch condition should specify the condition for being at the wall and moving up to the next row. The right branch should specify the condition for grasping the block and returning with it. The two conditions are identical, therefore the choice is nondeterministic.

Figure 4-16 shows the nondeterministic fork that NODDY generated while processing the final trace for the RETRIEVE procedure. Both branch conditions are [at: `anywhere` contact: 180°], so there is a conflict between them.

There are three possible reasons why a fork might be nondeterministic:

(a) The branch events with the conflicting conditions should be merged, but the previous stages did not do so;

(b) The branch conditions were overgeneralized; (the case in figure 4-16)

(c) The fork in the goal procedure actually occurred earlier in the procedure, and NODDY mistakenly merged two events that should have remained distinct.

When NODDY finds a fork with a pair of branch conditions that are not disjoint, it first assumes that (a) above may be the cause, and attempts to merge the events. Currently, it only uses the first level event generalization of the propagation and grouping stages. However, there would seem to be sufficient justification to also search for a functional generalization of the actions, if necessary, since the events appear to play an identical role in the procedure to the point that NODDY cannot distinguish between the conditions for performing the actions.

When NODDY is able to merge the conflicting branch events, it rechecks the fork to ensure that this removed the nondeterminacy, and then proceeds to another fork.

If one branch condition is a generalization of the other, but the events cannot be merged, NODDY marks the fork as ordered, and gives a higher priority to the more specific (special case) condition.

If the branch conditions conflict, ordering the branches is of no help, so NODDY assumes that the earlier stages overgeneralized the branch conditions during a previous stage if the events cannot be merged (as in figure 4-16). This is very likely, since the simplest generalized condition is the always-satisfied condition, and the event matching of the earlier stages usually finds the simplest condition generalization, as section 4.4.1 discussed. However, in contrast to the earlier stages, NODDY now has justification for considering more complex, and therefore more specialized, conditions.

To resolve a conflicting branch condition, NODDY must backtrack in the space of generalized conditions to find another condition that satisfies all the patterns of the original condition, but does not conflict with any of the other branch conditions. Each condition contains, along with the description of the condition, a list of the patterns from which it was constructed.* The patterns in a branch condition are therefore positive examples of the correct branch condition, and the patterns in all the other branch conditions negative examples. The negative examples are essential for justifying the choice of a more complex condition, so NODDY invokes the event matcher with both the positive and negative examples.

If the event matcher is able to find new conditions that satisfy both the positive and negative data, NODDY installs them into the branch events in place of the overgeneralized conditions, and rechecks the fork.

If the event matcher cannot find new conditions, then NODDY assumes that the new trace is too complex to be assimilated yet, gives up the attempt to find a valid generalization of its current version of the procedure and the trace, and waits for the teacher to give it a simpler trace. In the future, it should consider the third possible cause above—that the real fork ought to occur earlier in the procedure. This situation can arise very simply

* It is not strictly necessary to keep all the patterns around, though NODDY may make mistakes if it discards some of the patterns. None of the examples which NODDY has acquired so far has been sufficiently complex that the tradeoff between the mistakes and the extra computational cost needed addressing.

if the past values of the branch conditions intersect—there is one pattern for which both branches have been taken in the past. Such a conflict implies that the choice between the two actions actually depends upon an earlier pattern, but NODDY found no need to form a fork at that pattern because all the events matched.

NODDY should be able to resolve this situation by "unzipping" the branch leading to the fork, effectively moving the fork to an earlier point in the procedure. This involves undoing some of the merges of the previous stages. Some of the mechanism for doing this backtracking is already in place, but the actual backtracking is not yet implemented.

When all the forks have been checked and either passed, ordered, or resolved, the new procedure is complete, and NODDY waits for a new trace.

Chapter 5
Action Matching and Generalization

The task of the event matcher is to match a set of events and find a generalization of them if they match sufficiently well. The structure matcher hands the event matcher a set of events and a specification of what sort of match is required—for example, whether the structure matcher will accept a generalization if necessary or just wants to know if the events are exactly the same. The event matcher will return a measure of how well the actions matched, if at all, and a new event that is a generalization of the set of events if they did match.

The event matching and generalization consists of two, almost completely separable, parts—the action matching/generalization, and the condition matching/generalization. This chapter addresses the action matching/generalization and the following one will address the condition matching/generalization.

There are two different generalization tasks that the action matcher must do. The first is illustrated by the "move toward goal" action in the TURTLE procedure (see figure 1-11) in which the action matcher determines that the three primitive actions `MOVE 6@180°`, `MOVE 2@180°` and `MOVE 4.1@-173°` of the first two traces have the generalization `MOVE-UNTIL-CONTACT-TOWARD` (0,0).

The second generalization task is illustrated by the "move to side" action. Here, the structure matcher suggested that the `MOVE 0.5@0°` and `MOVE 0.5@45°` actions of the second and third traces might be dependent on one of the components of the pattern of the preceding event. The action matcher could find no relation between the actions and the position component, but found that the direction of the `MOVE` was 90° to the right of the contact direction. The generalized action, therefore, was `MOVE` 0.5@(<*contact direction of* E3>−90°).

This chapter describes the two algorithms by which NODDY does these two types of generalization. The first algorithm is based on a hierarchy of templates of generalized actions and matches exactly two actions at a time. The second algorithm may be invoked on two or more actions at once, and involves searching a space of lambda expressions.

5.1 First Level Action Generalization

There are three ways that a pair of actions can match: they may be equal, one may

include the other, or there may exist a generalization of them both. The first level action matcher checks each of these possibilities in turn, using the action template hierarchy to check for inclusion or to construct a generalization. Section 5.1.1 describes the equality testing; section 5.1.2 discusses the structure of the action template hierarchy, and sections 5.1.3 and 5.1.4 describe how the action matcher uses the action template hierarchy to test inclusion and construct generalizations.

5.1.1 Equality testing

The first step of the first level action matcher is to determine whether the two actions are the same action, and if not, whether a generalization is possible. If the actions are equal, it will return a copy of the actions and a score of EQUAL. If no generalization is possible, it will return a score of FAIL. Otherwise, the actions will be passed to the next step of the action matcher.

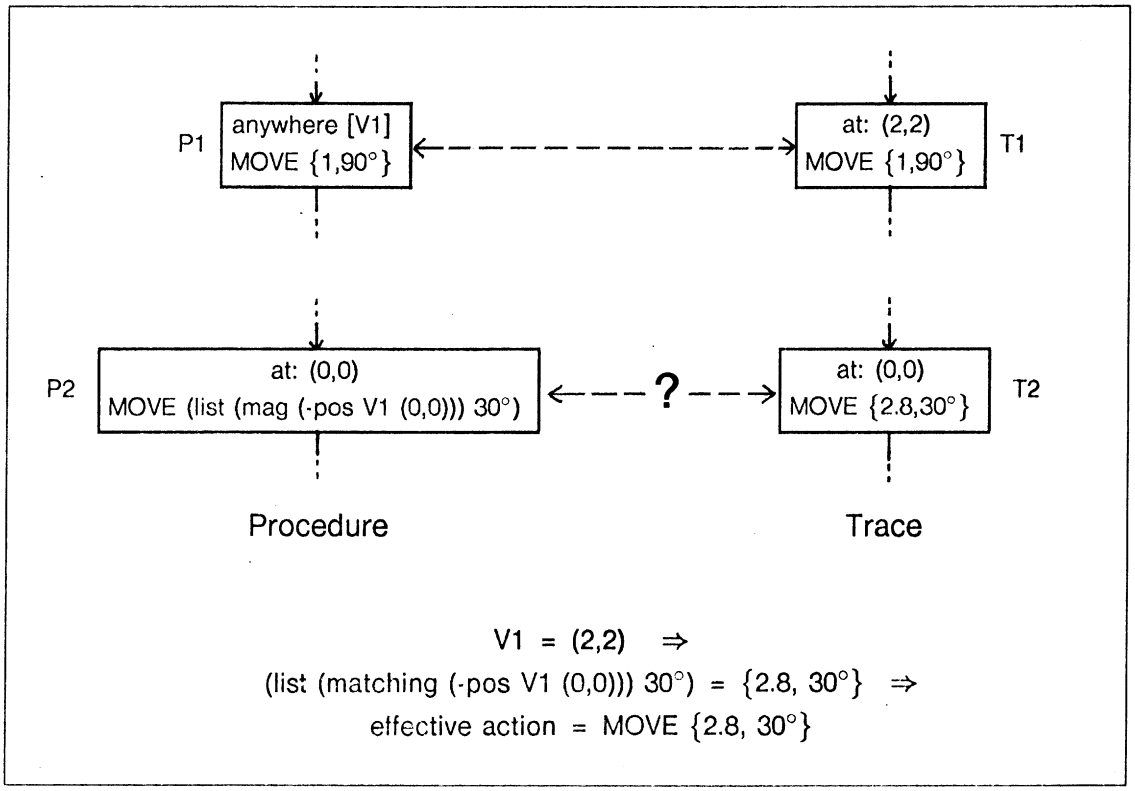
If the actions are of different classes, (*e.g.*, a `MOVE-TO` $(0,0)$ and a `ROTATE` 60°), there can be no possible generalization* and the action matcher will return a score of FAIL. If the actions have the same type, the action matcher must check their parameters. There are three cases:

- The parameters of the two actions are constants: The actions are equal if the parameter lists are identical.
- Both actions have functional parameters: The actions are equal if both actions are functions of the same variables (or their variables refer to the same pattern component) and the two expressions are the same with the appropriate substitution of lambda arguments.
- Only the first action has functional parameters: The actions are equal if the parameters of the "effective action" specified by the first action in the context of the current trace are identical to the parameters of the second action.

To determine the effective action, NODDY must find values for the variables of the action and apply the function expression to these values. The variables refer to components of conditions in the procedure. The values of the variables are the values of

* It would be possible to to have an action that would move or rotate depending upon the situation, but this would require a disjunctive description of the action. Rather than allow explicit disjunctions in actions (which would introduce the full disjunction problem into the action generalization), the disjunction is left in the procedure structure: a fork will be placed in the new procedure, and the two events will form the branches.

the corresponding components of the trace patterns that matched the procedure conditions (see figure 5-1). If NODDY can find values for the variables, it computes the parameters of the effective action by applying the expression to the variable values. It can then check whether these parameters of the effective action are equal to the constant parameters of the second action.



— — Figure 5-1 Matching Functions to Constants — —

NODDY is matching the action on the left (event $P2$) which has functional parameters to the trace action on the right with constant parameters. The variable V1 of the action on the left refers to the position component of the condition of event $P1$. Event $P1$ was earlier matched to the trace event $T1$. The position component of $T1$'s pattern is the position (2,2), which is therefore the value of V_1. Evaluating the expression `(list (mag (-pos V1 (0,0)))30°)` produces the effective action MOVE 2.8@30° which is identical to the action of event $T2$ on the right. The actions are therefore equal.

5.1.2 Matching and Generalizing with the Action Hierarchy

If the two actions are not equal, then the first level action matcher checks whether one action is a generalization of—includes—the other. Otherwise, it looks for a generalization of the actions.

5.1.2.1 Generalizing Actions

Both of these steps—testing for inclusion and finding a generalization—involve reasoning about the space of generalized actions. If NODDY could represent this space explicitly as a generalization hierarchy, NODDY could use the standard generalization method of generalizing by climbing a generalization hierarchy: the include step would check whether one action is a superior of the other, and the generalization step would look for the lowest common superior of both actions.

However, the space of all possible actions is too large to represent or search explicitly, even in a simple domain such as the 2-D robot domain, since the parameters of the actions may range over infinite sets so that even the set of actions of a single type (*e.g.*, the `MOVE-TO` actions) is too large to represent explicitly. Therefore, NODDY cannot use the "climb hierarchy" generalization method. Nor can it use the method of breaking actions into subcomponents and generalizing each component separately (*e.g.*, by climbing a hierarchy for each component), since the parameters interact with one another.

An alternative to searching an explicit generalization hierarchy is to dynamically construct just the relevant part of the generalization hierarchy using data-driven generalization heuristics. Given two actions to generalize, the heuristics would construct the relevant generalizations of the actions until it reached a common generalization.

NODDY expresses these heuristics implicitly in a hierarchy of action templates. Each template has its own form of the general heuristic specialized to the action type represented by the template. The reason for expressing the heuristics this way is that the general heuristics are quite expensive, and it is much more efficient to separate them into two distinct parts—the generalization associated with the action type, and the generalization associated with the parameters of a particular action—and to cache the results of the type generalization in a template with specialized heuristics for generalizing the parameters of actions of that type.

Any generalized action must satisfy the determinacy constraint—an action must specify just one primitive action to perform in any given situation. For example, the action `MOVE {7}` is a generalized action in the sense that it covers all `MOVE`'s that move a distance of 7 in some direction. However, it does not satisfy the determinacy constraint since it does not specify which `MOVE` to do in any particular situation. This constraint greatly reduces the space of generalized actions.

There are two ways of constructing generalized actions that satisfy the determinacy constraint—*functional generalization* and *guard generalization.*

Functional Generalization

Functional generalization replaces constant parameters of an action (whether guard parameters or essential parameters) by a function of one or more components of previous patterns. It requires that the actions have parameters that can be expressed as a function of a previous state. If the generalized action is expressed as a function of the current state (or pattern), the function is referred to as a *local function.* Local functions are particularly important because the action matcher can construct a local functional generalization of the events of two actions without referring to any other events. The structure matcher invokes the first level action matcher on pairs of events that must be matched and generalized independently of the the rest of the procedure and trace because the rest of the procedure and trace may not be matched yet. Therefore, the first level action matcher can only do local functional generalization, and the non-local functional generalization is left to the second level action matcher.

The action `MOVE (vect->list (-pos (1,2) <current-position>))` action is an example of a local functional generalization. The action specifies whatever primitive `MOVE` will take the robot from its current position to the position $(1,2)$. The `MOVE-TO` template expresses a class of action generalizations of this form so that `MOVE-TO` $(1,2)$ is an equivalent (though simpler) description of the same action. The details of what is in a template, such as the `MOVE-TO` template, are given in the next section.

The ordering on functional generalizations is determined by the number of variables in the expression, so that one functional action is a generalization of another if the function expressions are identical except that some constants in the expression of the more specific action are replaced by variables in the expression of the more general action.

Guard Generalization

Guard generalization depends upon the distinction between two types of parameters—*essential parameters* and *guard parameters.* The essential parameters of an action are those which determine how to start performing the action. The guard parameters, on the other hand, specify a stopping condition, or *guard*—specifying *when* an action should stop rather than *how* it should start. For example, the direction parameter of a `MOVE` action must be given before the move can start, else the robot cannot determine which direction to move

in, and is therefore an essential parameter. The distance parameter, on the other hand, is needed only to determine when the action should stop, and could conceivably be specified after the action had already started. To be more precise, the MOVE {r,θ} action should be represented as MOVE {θ `until` [$distance = r$]} to make explicit that the r parameter is actually the parameter of a stopping condition.

Guard generalization replaces a guard condition by a more general guard condition. For example, one guard generalization of the action

$$\texttt{MOVE } \{\theta \texttt{ until } [distance = r]\}$$

is the action

$$\texttt{MOVE } \{\theta \texttt{ until } [distance = r \vee contact]\}$$

The MOVE-UNTIL-CONTACT-UP-TO template expresses this generalization, so that MOVE-UNTIL-CONTACT-UP-TO {2, 50°} means MOVE {50° `until` [$distance = 2 \vee contact$]}.

Any element of the sensory feedback that is continuously monitorable during the execution of an action could be the basis for a guard condition. Clearly, what these elements are depends on the domain. Note that whereas the local functional generalization involves an explicit dependency on the pattern immediately preceding the execution of the action, the guard generalization is related to the pattern immediately *following* the action. That is, the guard condition will generally be present, whether explicitly or implicitly, as part of the following pattern. The action is not, however, a **function** of the following pattern since it is not possible to compute a function of a pattern that has not yet occurred.

The ordering on the guard generalizations is strictly by generality of the guard conditions, so that actions with the most specific conditions are always considered before actions with more general conditions.

In the 2-D robot domain, the only elements of the feedback that are available for guard generalization are the distance traveled and whether the robot is in contact with an obstacle or not. There are therefore only a few possible guard generalizations, which are discussed below. NODDY does not embody any strong theory of guard generalization. Chapter 8 will mention some directions for future work in the area of guard generalization and procedures with interesting guarded moves.

5.1.2.2 The Action Template Hierarchy

The current version of NODDY cannot construct its own action template hierarchy, so it must be provided by the user. The form of the templates will be the same in any domain,

but the content is domain dependent. There is no domain dependent information in the action matcher outside the action hierarchy. Chapter 8 will discuss the ways in which NODDY could construct its own hierarchy in the future.

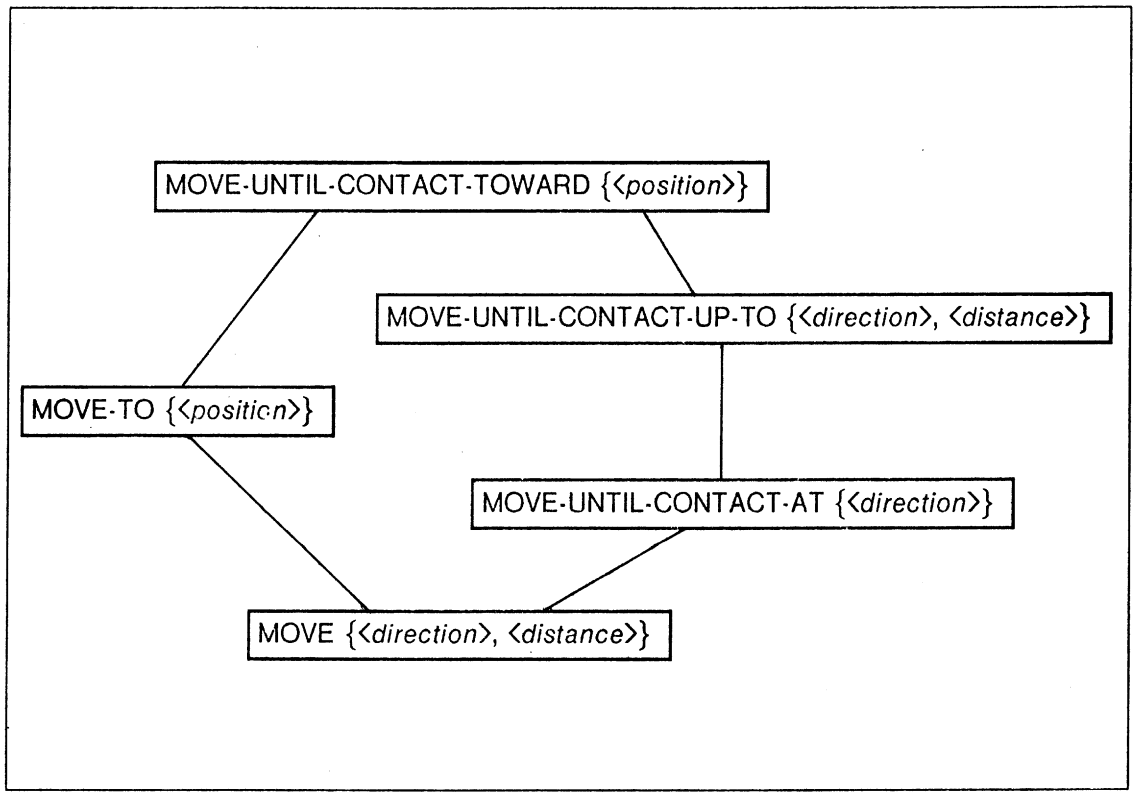
The hierarchy consists of a set of templates and relations between them. Each template contains a specification of the parameters of an instance of the template, the specialized generalization heuristics for instances of itself, and a set of pointers to other templates in the hierarchy. Since the actions of one type are strictly incomparable with the actions of another type, the hierarchy is actually a set of hierarchies, one for each action type. Action types corresponding to primitive actions with no parameters have only one template, which is the trivial one describing the primitive action.

Action templates cannot strictly be generalizations of one another since they specify whole classes of actions rather than single actions. However, we will speak of a template U being a generalization of a template L if any instance of L has a generalization that is an instance of U. Each template has pointers to its immediate generalizations and specializations. U is an immediate generalization of L if an instance of L can be generalized directly to an instance of U without requiring an intermediate generalization to an instance of a template between L and U. For example, two primitive `MOVE`'s cannot be generalized to a `MOVE-UNTIL-CONTACT-TOWARD` without an intermediate generalization to a `MOVE-TO` or `MOVE-UNTIL-CONTACT-UP-TO`.

The specialized generalization heuristics of each template are expressed as a set of functions. Each function constructs an instance of the template that is a generalization of two instances of lower templates. These functions embody the local functional generalization and the guard generalization described above. A template also contains a set of predicates which determine whether an instance of the template is a generalization of an instance of a lower template. The functions and predicates take a pair of actions as arguments. They also refer to a component of either the current pattern (functional generalizations) or a component of the pattern in the following event (guard generalizations), or both. The predicates and functions are indexed according to the more specific templates to which they correspond.

Figure 5-2 shows the structure of the `MOVE` segment of the action template hierarchy for the 2-D robot domain.

The hierarchy does not contain all the templates that could be put in it. In particular, it does not contain any of the many other local functional generalizations that exist. This



— — Figure 5-2 The MOVE Template Hierarchy — —

The root of the MOVE hierarchy is the `MOVE` action which has one essential parameter (the direction to move) and one guard parameter (the distance to move). The `MOVE-TO` template represents a functional generalization of `MOVE`; the `MOVE-UNTIL-CONTACT-AT` and `MOVE-UNTIL-CONTACT-UP-TO` templates represent guard generalizations, respectively moving in the specified direction until contact with an obstacle, and moving in the specified direction up to the specified distance, or until contact whichever comes first. The `MOVE-UNTIL-CONTACT-TOWARD` template combines both types of generalizations.

is not a significant limitation, for two reasons. The first is that, ultimately, NODDY will be able to construct its own hierarchy, so that the hierarchy need not be complete because NODDY will add to it when necessary. The second reason is that NODDY can find local functional generalizations with the second level action generalization, though not as easily, and under more constrained circumstances, so that the lack of templates will not prevent NODDY from acquiring a procedure, just slow the rate of acquisition and increase the number of examples needed. The effect of having a small template hierarchy is to bias NODDY toward certain generalizations, and increase the ease of finding generalizations in the propagation stage. It is, in fact, desirable to keep the hierarchy small for this reason. Therefore, when NODDY becomes able to augment the hierarchy, it should only add "common" templates that are likely to be useful for many procedures, otherwise the

increased range of the first level action generalization will be offset by the need for more examples to determine which generalization is appropriate.

The guard generalizations expressed in the hierarchy are somewhat strange. Although MOVE-UNTIL-CONTACT-AT is treated as a generalization of the primitive MOVE, this is not strictly so, since the guard condition `until [contact]` is not a generalization of `until [distance = r]`. Also the distance traveled never appears in the pattern of the following event. Strictly, the primitive MOVE ought to have the most specific guard condition possible. In the case of a primitive move ending in a contact, this would be the conjunction of the distance traveled and the contact. NODDY's representation for actions and patterns was fixed before the issues involved in guard generalization were sufficiently understood, and as a result, the generalization functions in the MOVE-UNTIL-CONTACT-AT and MOVE-UNTIL-CONTACT-UP-TO templates are more complex and less consistent than they should be. Future versions of NODDY will need to have a cleaner representation of guard conditions, particularly if NODDY is to generate its own templates based on guard generalization.

5.1.3 Checking Inclusion

NODDY first tests for inclusion, checking whether the first action includes the second, and then *vice versa.* There are several different cases that NODDY must test in different ways depending on whether the actions are instances of templates in the hierarchy, have functional parameters, or both. If neither action has functional parameters, then NODDY finds whether the template of the first action is a superior of the template of the second action. NODDY retrieves the predicate attached to the template of the first action corresponding to the template of the second action and applies it to the two actions. If there is either no predicate or the predicate is not satisfied, then the first action does not include the second, and the action matcher proceeds to the next step of trying to construct a new generalization. If the predicate is satisfied, then the first action includes the second, and the matcher returns a copy of the more general action.

For example, to test whether the action MOVE-UNTIL-CONTACT-AT 180° includes an action MOVE 4@180°, NODDY would retrieve the appropriate predicate from the MOVE-UNTIL-CONTACT-AT template and apply it to the two actions. This predicate would check that the directions of the two actions are the same and that the pattern following the MOVE action involves a contact.

When the first action has functional parameters and the second action does not, NODDY computes the "effective action" (as in section 5.1.1) and determines whether the effective action is a generalization of the second action. If both actions have functional parameters, the first is a generalization of the second if its variables are a superset of the variables of the the second action and the second action is identical to the second action with the "excess" variables replaced by their effective values. (This last test is not actually implemented, though it would not be hard to do so).

5.1.4 Finding Generalizations

If neither action is a generalization of the other, the action matcher will attempt to find a new action that is a generalization of both actions. Currently, it will try first level action generalization only when neither action has functional parameters. In other words, once an action has been generalized using the second level action generalization, the first level no longer applies. One area of future work is to improve the representation of generalized actions so that the two types of generalization can be better integrated.

When both actions are simple instances of templates in the template hierarchy (possibly different instances of the same template), NODDY climbs the hierarchy to find templates that are common superiors of the templates of each action. For each superior template, NODDY retrieves the specialized generalization heuristics for that template, and applies the appropriate heuristic to the two actions. The action matcher returns the generalized action from the first superior template whose generalization heuristic returns a generalization. If there are no generalizations from any of the superior templates, the action matcher fails.

If the superior template represents a functional generalization, then the generalization heuristic of the template must examine the relevant components of the conditions associated with the actions.

For example, if the actions are `MOVE` 6@0° and `MOVE` 4@90°, NODDY considers each of the action templates `MOVE-TO`, `MOVE-UNTIL-CONTACT-AT` and `MOVE-UNTIL-CONTACT-UP-TO` since these are superiors of the `MOVE` template. The generalization heuristic of the `MOVE-TO` template must consider the position components of the conditions associated with each of the actions to determine whether both `MOVE`'s end at the same point. If the position components were (-6,0) and (0,-4), then the actions would both end at (0,0), so the generalization heuristic will construct the generalized action `MOVE-TO` (0,0).

However, a problem may arise if the associated conditions have been generalized. For

example, if the position components of the condition associated with the MOVE 4@90° action had been generalized to [anywhere], NODDY would not be able to determine where the MOVE started, and therefore could not tell whether it ended at the same place as the MOVE 6@0° action.

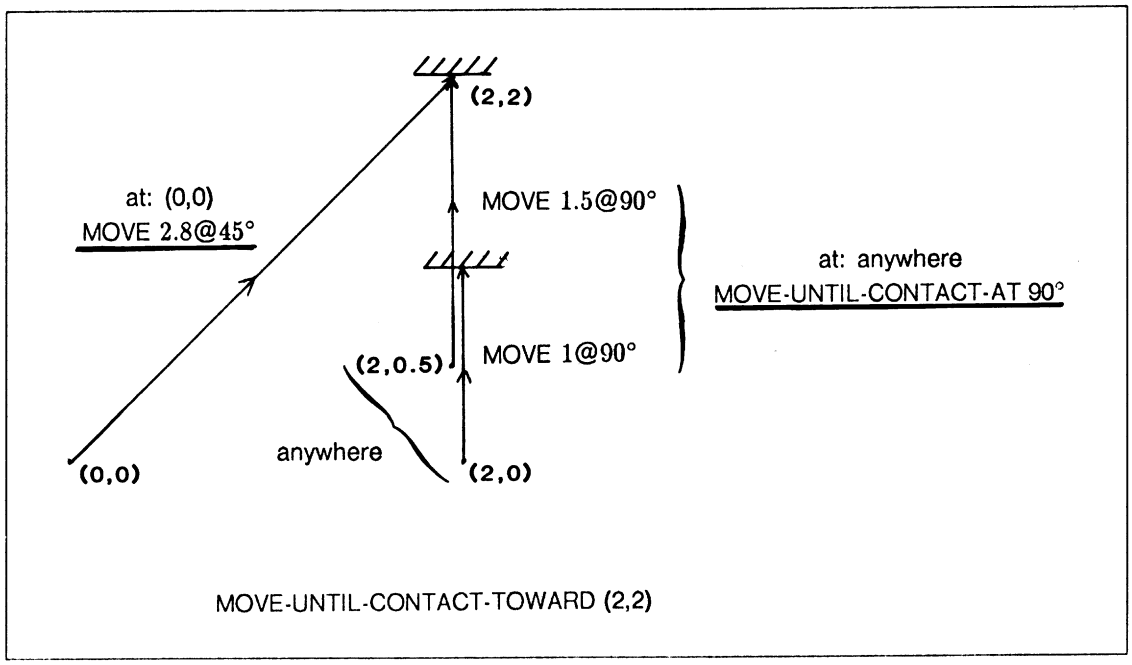
It would seem, therefore, that NODDY can only generalize actions if the associated conditions have not been generalized. This is not, in fact, quite true, nor would it be a significant restriction even if it were true. NODDY could only have generated the [anywhere] condition by merging two events that had different positions, but the same MOVE 4@90° actions. But if this were so, the MOVE's must have started at different positions and could not end at the same position. Therefore there is no MOVE-TO generalization of the actions anyway. In general, an ungeneralized action could not be a function of any generalized condition component since this would imply a *many-1* relation between the pattern parameters and the action parameters.

If the only types of generalized actions in the action hierarchy were functional generalizations, there would be no problem at all—the constructor functions would not apply unless the relevant part of the condition component was ungeneralized. However, the guard generalizations complicate the issue.

For example, figure 5-3 shows a MOVE 2.8@45° action and a MOVE-UNTIL-CONTACT-AT 90° action which can be generalized to a MOVE-UNTIL-CONTACT-TOWARD (2, 2). The MOVE-UNTIL-CONTACT-AT was generalized from two primitive MOVE's that started from different positions, and hence its associated condition is [anywhere]. To find the appropriate generalization of the two actions, the action matcher must determine that all the primitive MOVEs were headed toward the same point.

It is quite possible for a MOVE-UNTIL-CONTACT-AT 90° with a condition of [anywhere] to be a generalization of MOVEs that head toward quite different points. For example, if the MOVEs started at different points on the x-axis, they would all be parallel, but would never intersect. Therefore, the action matcher cannot tell whether the MOVE 2.8@45° and MOVE-UNTIL-CONTACT-AT 90° actions can be generalized to MOVE-UNTIL-CONTACT-TOWARD (2,2) just by looking at the parameters of the actions and the associated conditions.

Currently, NODDY solves this problem by using the past values of the condition node—the actual positions from which the [anywhere] was constructed. It determines whether the point (2,2) lies on each line at 90° through one of these positions. If so, it



— — Figure 5-3 Complications With Guard Generalizations. — —

NODDY must find a generalization of the action MOVE 2.8@45° and MOVE-UNTIL-CONTACT-AT 90°. The MOVE-UNTIL-CONTACT-AT 90° action is a generalization of the two primitive actions MOVE 1@90° and MOVE 1.5@90°, shown in the center of the figure. Clearly, the appropriate generalization is the action MOVE-UNTIL-CONTACT-TOWARD (2,2) since all the primitive actions are moving toward the point (2,2) and either reach that point or or hit an obstacle. However, the two MOVEs in the center started at different positions, so that when the actions and the associated conditions were generalized, part of the information about where the MOVEs started and ended was lost. Therefore, determining that MOVE-UNTIL-CONTACT-TOWARD (2,2) is a valid generalization is not straightforward.

guesses that all the primitive MOVEs of the MOVE-UNTIL-CONTACT-AT must have been heading toward (2,2).*

Without using the past values of the conditions in this way, NODDY would become much more dependent on the order of presentation of the examples—to teach an action that involved both functional and guard generalization, the teacher would have to present traces that would introduce the functional generalization first, before the associated conditions are generalized, and then traces to introduce the guard generalization.

For example, to teach the MOVE-UNTIL-CONTACT-TOWARD action of figure 5-3, the teacher would have to first present traces with the two MOVEs that reach (2,2), which would

* This is an informed guess—an inductive inference—, because there is no way of telling whether any of the MOVEs might have overshot (2,2) without keeping around a complete record of all the MOVEs from which the MOVE-UNTIL-CONTACT-AT was generalized.

be generalized to MOVE-TO (2,2), and then the third MOVE which hits an obstacle halfway to (2,2) which would then enable NODDY to generalize the actions to MOVE-UNTIL-CONTACT-TOWARD (2,2).

This would also mean that NODDY could only acquire the RETRIEVE procedure if the traces were presented in the order shown in the scenario. However, because NODDY can generalize actions, even when the associated conditions have been generalized, it can acquire RETRIEVE with the first four traces presented in any order.

5.2 Second Level Action Generalization

The second level action matcher is invoked when the first level action matcher could not find any generalization of the actions in the action hierarchy **and** there is sufficient justification to search for a new generalization. The justification is a combination of the contextual justification, described in section 4.5, and the internal justification that the actions are of the same type and differ only the parameters. Unlike the first level action matcher, the second level action matcher will match any number of actions at once. In fact, the more actions there are, the better, for reasons discussed below. The second level action matcher tries to find a functional dependency between the actions and one component of a previous pattern. Its first step is to obtain a list of "input/output pairs" of the function it has to find—the pattern values and the corresponding action values. If there is a consistent set of input/output values, the second step is to search for a function that explains those values.

5.2.1 Finding the Input/Output Pairs

The structure matcher hands the second level action matcher a set of actions, a partial match of the actions, and the component of the condition of some event. The partial match of the actions was generated by the first level action matcher and specifies the parts of the actions that are identical. If the actions have constant parameters, the partial match will just be the action type and the output values will be just the parameters of each action. If the actions have functional parameters, then the the function expressions must be identical except for some constant, and the partial match will be the action type and the function expression, marked at the place where the actions have differing values for a constant in the expression. The output values are just the different values of the constant, which NODDY obtains from the actions using the specialization in the partial match.

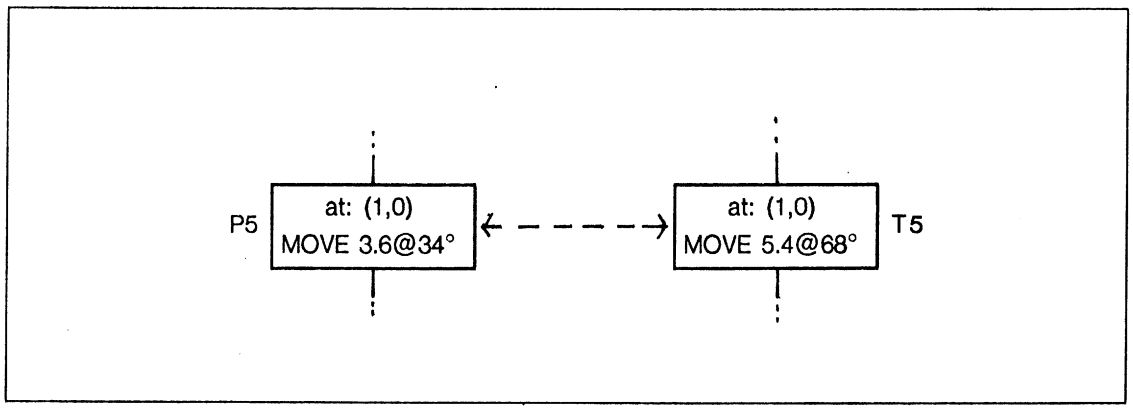
To construct the input/output pairs, NODDY must determine the input values from the condition component, and the correspondence between the input and output values. The condition component has two parts: the actual condition, which is an instance of some template in the condition hierarchy, and the past values, which are the actual pattern values from which the condition bound was generalized. The function that NODDY must find is a function of the pattern value that matches the condition component, not of the condition component itself, so the input values are just the past values of the condition component. Inferring functions is one of the two reasons that the past values must be remembered—they are also used in constructing more complex conditions, as described in chapter 6.

To determine the correct correspondence between the input values and the output values, NODDY needs to know which actions occurred after which pattern values. If NODDY remembered all the traces and the relation of the events in the current procedure to the traces, it could determine the correspondence by referring back to the traces. It is not necessary, however, to remember everything about the traces: it is sufficient to mark the actions and the past values of the conditions with the name of the traces from which they came. NODDY can then pair the input and output values that are marked with the same trace name and remove any duplicate pairs.

Once it has obtained the set of input/output pairs, NODDY checks the pairs for consistency. If the same input value is paired with two different output values, the output values could not be a function of the input values, and the matching is abandoned immediately. For example, when NODDY was trying to generalize the "move back to starting point" actions of the first two traces of the RETRIEVE procedure, it first looked for a function relating the the `MOVE` parameters to the current position (see figure 5-4). However, the position was the same—(1, 0)—for each action, so this position component was abandoned, and NODDY tried the condition of the previous event.

5.2.2 Searching for a Function

When the input/output pairs are consistent, and there are at least two pairs, NODDY searches for a function that satisfies the pairs. The space of possible functions is implicitly defined by the set of operators from which NODDY may construct expressions. The functions are represented by the set of lambda expressions that can be built out of the operators, lambda variables, and constants. NODDY does a breadth first search of the



— — Figure 5-4 Abandoning a Condition because of Inconsistent Pairs — —

NODDY is looking for a second level generalization of the "move back to starting point" actions of the first two traces of the RETRIEVE procedure. NODDY first considers the position component of the current condition as the source of the functional dependency. However, since the positions are both (1,0), the different actions cannot possibly be a function of this condition component. Therefore NODDY prunes this condition from the search and steps back in the procedure to consider and earlier condition

relevant part of the space of expressions ordered by depth of nesting of operators. The set of operators is domain dependent and must be given to NODDY (figure 5-5 shows the set of operators for the robot domain), but the algorithm for searching the space is domain independent.

An essential part of the search algorithm is the use of *type* information. As shown in figure 5-5, each operator must have associated with it type information on its arguments and result, and a specification of its inverse (or inverses, if there is more than one). NODDY incrementally builds expressions inwards from the input values and the output values simultaneously, using the type of the values to constrain the possible operators. The precise algorithm is given in section 5.2.3.

As long as the set of operators is small, as it is for the robot domain, the search space of expressions without constants is quite manageable. When constants are introduced, however, the space of expressions immediately becomes very wide and unmanageable because it is not possible consider all possible values of the constants. (This does not apply to domains in which the constants can only take on values from small sets such as constants ranging over the primary colours or even letters of the alphabet, but would apply to constants ranging over large finite sets such as the set of all words.) The algorithm below avoids this problem by tightly restricting the consideration of expressions with

OPERATOR	DOMAIN	RANGE	INVERSE1	INVERSE2
pos-cons	(num,num)	pos	x-part	y-part
x-part	(pos)	num	pos-cons	
y-part	(pos)	num	pos-cons	
vect-cons	(num,dir)	vect	mag	direction
mag	(vect)	num	vect-cons	
direction	(vect)	dir	vect-cons	
list	(any,any)	2list	first	second
ncons	(any)	1list	first	
first	(1list)	any	ncons	
cons	(any,any)	list	car	cdr
car	(list)	any	cons	
cdr	list	list	cons	
second	(2list)	any	list	
vect->list	(vect)	list	list->vect	
list->vect	(list)	vect	vect->list	
pos+vect	(pos,vect)	pos	pos-vect	-pos
-pos	(pos,pos)	vect	pos+vect	pos-vect
pos-vect	(pos,vect)	pos	pos+vect	-pos
+vect	(vect,vect)	vect	-vect	-vect
-vect	(vect,vect)	vect	+vect	-vect
minus-vect	(vect)	vect	minus-vect	
minus	(num)	num	minus	
+	(num,num)	num	-	-
-	(num,num)	num	+	-
+ang	(ang,ang)	ang	-ang	-ang
-ang	(ang,ang)	ang	+ang	-ang
-dir	(dir,dir)	ang	dir+ang	dir-ang
dir+ang	(dir,ang)	dir	dir-ang	-dir
dir-ang	(ang,ang)	ang	dir+ang	-dir

— — Figure 5-5 Table of operators for the Robot Domain. — —

Each operator must specify the types of its arguments and range. These are used in the expression building algorithm to prune the candidate operators. It must also specify its inverses which are used to build the expression from both ends at once.

constants. It will consider expressions with one "new" constant, because one constant can be found without searching all possible values for the constant (see step 3 of the algorithm), but it will never introduce more than one new constant into an expression. It will consider expressions with "old" constants—constants that are known from the context

of the actions to be possibly relevant—since these also do not require searching the space of constant values. These restrictions do limit the expressions that can be considered, but the alternative is to make the search prohibitively expensive. By introducing very domain specific knowledge into the algorithm, the space of expressions could be increased somewhat (compare, for example, Langley's production system rules for finding linear relationships which involve two new constants[Langley *et. al.*, 1983]), but this runs counter to the domain independence of NODDY.

Since expressions may have arbitrarily deeply nested operators, there must be some stopping criteria for the search. NODDY uses two separate stopping criteria. The first is the number of constants in the expression, and the second is the depth of nesting. The number of constants (both "old" and "new") at which cutoff occurs depends upon the number of input/output pairs: the more constants in the expression, the more pairs are required to justify it. For example, one can find a linear relation $y = ax + b$ to satisfy two pairs of (x_i, y_i). However, since one can find such a relation for *any* two pairs, there is no reason to believe that the fact that this particular linear relation captures any underlying regularity or property of the values. One would only have confidence that the relation was a genuine generalization if one had three, or preferably four pairs that all satisfied the linear relation. Therefore, the number of constants allowed in an expression should be at most one less than the number of input/output pairs. The current version of NODDY has not addressed this issue very well, and uses a very primitive cutoff criterion. Future work should address this question.

It is less clear how the second stopping criteria—depth of expression—should be implemented. The depth of expression seems to have little relation to the number of data points. The current version cuts off the search at a fixed depth of expression, regardless of the amount of data. Effectively, this biases NODDY toward finding functions of the condition closest to the action, no matter what the complexity (up to the fixed limit). A more reasonable bias would be to trade off depth of expression against distance of the condition from the action so that a very simple function of a more distant condition might be preferred over a complex function of a closer condition. This would require a more complicated search strategy that dovetailed the search through the events with the search through expression space, and has not been implemented.

5.2.3 Expression Building Algorithm

The expression building algorithm is invoked with a single set of input/output pairs $\{in_i, out_i\}$, an initial expression, a list of known constants (constants that are known to be possibly relevant to the function) that can be used in the expression, and the stopping criteria—the maximum depth and maximum number of constants to be considered. The initial expression contains a "gap" which the algorithm must fill. For the simplest case—the actions being matched have constant parameters—the initial expression will be an empty lambda expression: `(λ (a1) (<gap> a1))`. If the actions had functional parameters, the initial expression will be the expression in the actions with a gap where the differing constant was.

The algorithm will try to close the gap in the expression. The gap represents some function F such that $\forall_i[out_i = F(in_i)]$. The algorithm can close the gap if it can find a single operator `OP` equivalent to F, in which case it can replace the gap by `OP`: `(λ (a1) (OP a1))`. It can also close the gap if it can find a single operator `OP` such $\forall_i$[(`OP` out_i in_i) = <*constant*>], in which case it can invert the operator to create an expression equivalent to F—$\forall_i[out_i$ = (`OP`$^{-1}$ in_i <*constant*>)] so that it can replace the gap by `OP`$^{-1}$: `(λ (a1) (OP`$^{-1}$ `a1` <*constant*>)).

If there is not a single operator that closes the gap, it will construct a new set of expressions, each of which partially close the gap by adding an operator to the inside or outside of the gap. Along with each new expression, the algorithm will construct a new set of input/output pairs obtained by applying the new operator to the input values (or the inverse of the operator to the output values). After pruning duplicate expressions and expressions corresponding to inconsistent input/output pairs, the algorithm repeats on the remaining expressions.

Algorithm

1 IF: there are no expressions left on the queue,
THEN: exit with no expression,

2 ELSE: Consider the next expression on the queue:

2.1 IF: input/output values are inconsistent
THEN: Abandon this expression and return to step 1.

2.2 IF: all the output values are equal,
THEN: EXIT with the the gap and subexpression inside the gap replaced by that value.

e.g., if $out_i = \{(90°), (90°), (90°)\}$ (<*gap*> a1)* $\Longrightarrow$ (90°)

2.3 IF: each output value equals its associated input value,
THEN: EXIT with the gap removed (implicit identity operator).
e.g., if $out_i = \{3@90°, 2@45°, \}$ and $in_i = \{3@90°, 2@45°, \}$,
(vect->list (<*gap*> (-pos a1 (0,0)))) $\Longrightarrow$ (vect->list (-pos a1 (0,0)))

2.4 IF: This expression has reached limit on either the depth or the number of constants,
THEN: Abandon this expression and return to step 1

3 IF: There is a binary operator OP such that applying OP to each of the input/output pairs returns the same value (*i.e.*, $\forall_i[(\text{OP } in_i\ out_i) = \langle const \rangle]$),
THEN: Replace the gap by the inverse of OP applied to <*const*> and the subexpression inside the gap, and EXIT with the new expression.
e.g., if in_i are positions and out_i are vectors, and (pos+vect in_i out_i) = (1,3),
then (<*gap*> a1) $\Longrightarrow$ (-pos (1,3) a1)

4 ELSE: Construct new expressions from this expression and add them to the queue:

4.1 IF: The out_i are a compound data type for which there is a constructor operator (*e.g.*, pos-cons for *positions*), and all but one of the corresponding components are equal (*e.g.*, $out_i = \{(2.5, 2), (2.5, 0), (2.5, 5)\}$)
THEN: Make a new expression by replacing the gap with the constructor operator applied to the constant components and the gap, replace the out_i with the non-constant components, and add the constant components to the list of known constants.
e.g., (<*gap*> a1) $\Longrightarrow$ (pos-cons 2.5 (<*gap*> a1)) and new $out_i = \{2, 0, 5\}$.

AND

4.2 IF: The in_i are a compound data type for which there are component operators (*e.g.*, mag of a *vector*)
THEN: For each component of the in_i that is not constant across the in_i, construct a new expression with the operator applied to the subexpression inside the gap, and replace the in_i by their appropriate components. Add the value of each component that is constant across the in_i to the list of known constants.
e.g., (<*gap*> a1) $\Longrightarrow$ (<*gap*> (mag a1)) and
replace $in_i = \{1@90°, 2.8@45°, 2.2@63°\}$ by $in_i = \{90°, 45°, 63°\}$.

AND

4.3 FOR EVERY unary operator OP whose range matches the type of the out_i, construct a new expression with OP applied to the gap, and replace each out_i by $(\text{OP}^{-1}\ out_i)$, where OP^{-1} is the inverse of OP.
e.g., (<*gap*> a1) $\Longrightarrow$ (ncons <*gap*> a1) and replace $out_i = \{(90°), (45°), (63°)\}$ by $out_i = \{90°, 45°, 63°\}$.

AND

4.4 FOR EVERY unary operator OP whose domain matches the type of the in_i, construct a new expression with the subexpression inside the gap replaced by OP applied to the subexpression, and replace each in_i by (OP in_i).

---

* The actual expression is (λ (a1) (<*gap*> a1)) but for clarity we will leave out the (λ (a1)

e.g., (<*gap*> a1) $\Longrightarrow$ (<*gap*> (minus-vect a1))
and replace $in_i = \{1@90°, 2.8@45°, 2.2@63°\}$ by $in_i = \{1@-90°, 2.8@-45°, 2.2@-63°\}$

AND

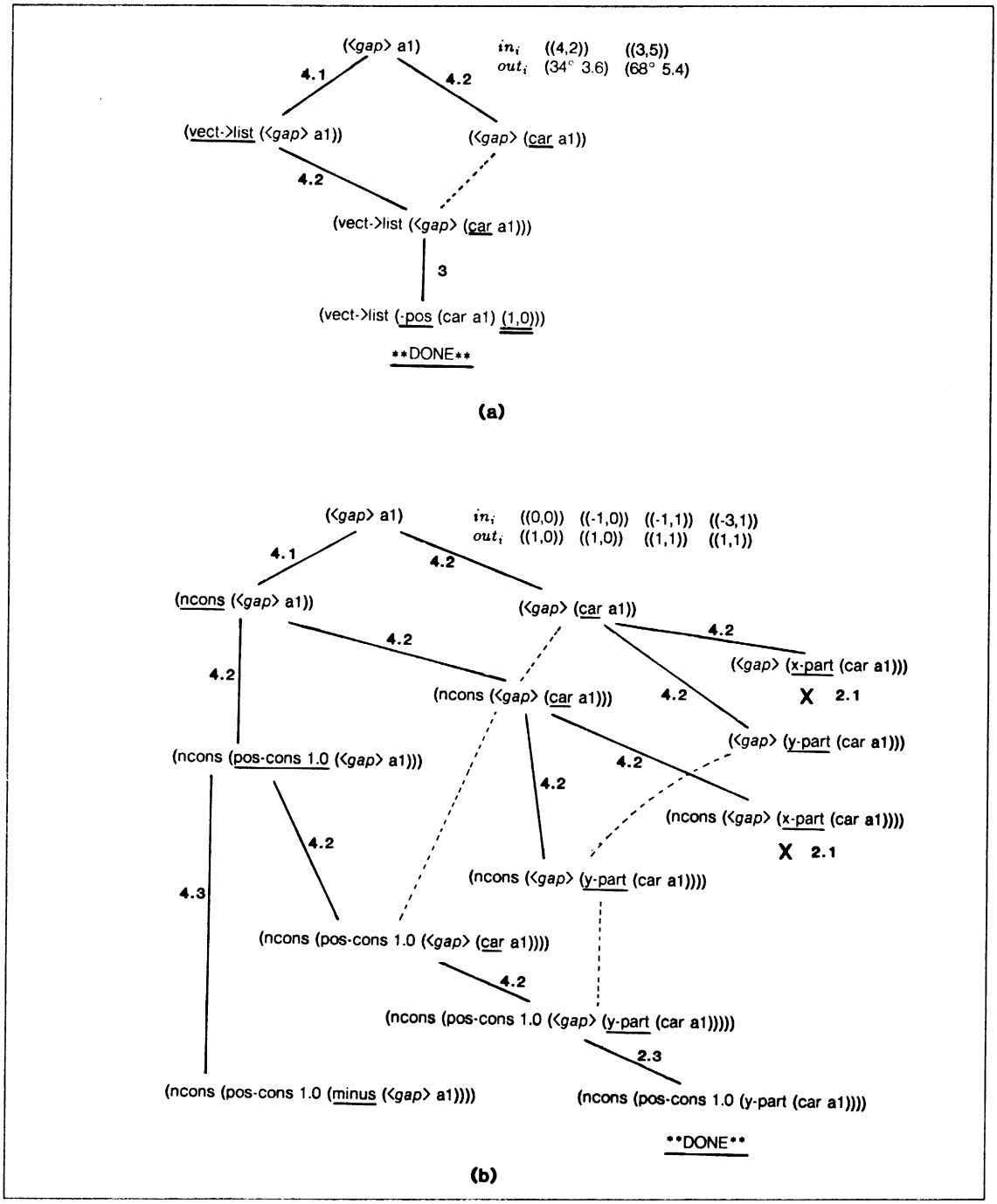
4.5 FOR EVERY non-unary operator OP whose range matches the type of the out_i, and for which there are known constants of the correct type for all but one of the arguments, construct new expressions with OP applied to the gap and the relevant constants, and replace the out_i by the appropriate inverse of OP applied to the out_i and the constants.
e.g., if $(0,1)$ is a known constant, then (<*gap*> a1) $\Longrightarrow$ (pos+vect $(0,1)$ <*gap*> a1) and replace each out_i by (-pos out_i $(0,1)$),
i.e., replace $out_i = \{(1,1),(3,2),(2,2)\}$ by $out_i = \{1@90°, 2.8@45°, 2.2@63°\}$

AND

4.5 FOR EVERY non-unary operator OP such that the type of the in_i matches one of the arguments of OP and there are known constants of the correct type for all of the other arguments, construct new expressions with OP applied to the subexpression in the gap and the relevant constants, and replace the in_i by OP applied to the in_i and the constants.
e.g., if 2.5 is a known constant, then (<*gap*> a1) $\Longrightarrow$ (<*gap*> (+ a1 2.5)) *i.e.*, replace $in_i = \{1,3,2\}$ by $in_i = \{3.5, 5.5, 4.5\}$

5 Add the new expressions (with their associated input/output pairs) to the end of the queue of expressions, remove all duplicate expressions, and return to step 1. (NODDY also removes a few expressions which are semantic, though not syntactic, duplicates of expressions already seen, *e.g.*, (lambda (a1) (first (ncons (<*gap*> a1)))) is identical to (lambda (a1) (<*gap*> a1)). Removing all such duplicates would require more knowledge of the operators than NODDY has, so NODDY is limited to removing expressions with a unary operator adjacent to its inverse.)

Figure 5-6 shows the trees of expressions that NODDY considered while deriving the functional dependency for the "move back to entry of row" and "move back to initial position" actions of the RETRIEVE procedure. To find the first function, the algorithm derived one new constant: $(1,0)$ by noticing that the input position minus the vector output was the same for all input/output pairs—$(1,0)$—and inverting the pos-vect operator to obtain the -pos operator. There are several expressions in the second example (part b) that were pruned by the consistency constraint. For example, the input and output values corresponding to the expression (<*gap*> (x-part (first a1))) were in_i: {0, -1, -1, -3} and out_i: {((1,0)), ((1,0)) ((1,1)) ((1,1))}. The second and third input values are the same, but the corresponding output values are different. Since this is not consistent, the expression was pruned.



— — Figure 5-6 The Expression Building Algorithm at Work. — —

The figure shows the two trees of expressions that the expression building algorithm constructed to account for the input/output pairs next to the roots of the trees. The upper tree is from the "move back to initial position" action in RETRIEVE-2 (see event 5 in figure 1-5) and the lower tree for the "move to entry point of row" action in RETRIEVE-4 (see event 4 in figure 1-8). The labels on the links refer to the steps of the algorithm that generated the lower expression from the upper one. Dashed links indicate redundant derivations that were pruned. The expressions marked with × were pruned because the input/output values were not consistent.

5.3 Future Improvements to the Action Matcher

There are several limitations to the action matcher in the current version of NODDY. Future work should address these limitations.

The first problem with the action matcher is that the first level action generalization (climbing the template hierarchy) is not integrated with the second level generalization (finding functions). Some of the generalizations in the first level action generalization (which uses the hierarchy of templates) are local functional generalizations, for which the function expression is "hidden" in the template. The second level action generalization may also find local functional generalizations for which the function expression is explicit in the action. These two types of function generalization should be treated identically by the action matcher, so that the matcher can determine that `MOVE-TO` (x, y) is the same as `MOVE (-pos` (x, y) *<current-position>*`)`. Currently, the template is opaque to the matcher, and these two actions do not match.

Furthermore, the first level action matcher does not apply, currently, to actions with functional parameters. The effect is that NODDY can never generalize an action using the templates (and therefore cannot do guard generalization) once it has generalized it by finding a functional dependency.

Both of these problems could be solved quite readily if the function that a template represents were made explicit in the template. The action matcher could then be extended to match equivalent actions whether they were expressed in terms of templates, functional parameters, or both. The first level action generalization could also be extended to generalize actions with functional parameters.

A second problem with the action matcher is that the function induction algorithm is not completely implemented. One aspect that must be addressed is the stopping conditions on the search. The search should abandon an expression when it reaches some complexity limit. This limit should be a function of the number of input/output pairs—the more input/output pairs, the greater the complexity of the expression that can be considered without producing unjustified, spurious relations.

The other aspect that must be addressed is the introduction of "known constants" into the expressions. It is not possible to search the space of constants, but there may be particular constants from the procedure that are likely to be relevant to the function for which it is searching. Two places the algorithm should look for such constants are (a) the

condition of the new event whose action it is trying to construct, and (b) the condition of the following event. These conditions are a partial description of the state of the world before and after the action. If any of the components of either of these conditions have constant parameters, then it is possible that the expression describing the action may involve these values. The algorithm might also use some of Langley's techniques [Langley 1981] of finding constant properties of the set of input/output pairs. For example, it could find the coefficients of a linear relationship between input and output values, by constructing difference tables.

A more significant change that could be made to the action matching concerns the different levels of generalization. Currently, generalizing with the template hierarchy is the most constrained generalization, and is put in the first level, and generalizing by searching for functional dependencies is put in a second level. However, searching for local functional dependencies is much more constrained than searching for non-local functional dependencies, in that there are fewer generalizations and less likelihood of finding a spurious generalization. Furthermore, finding local functions does not require that other events in the procedure be matched, whereas finding non-local functions does. Therefore, NODDY should not require as much justification to consider a local functional generalization as to consider non-local functional generalization. The next version of NODDY should distinguish these two categories of generalization. It should also more carefully address the question of what stages of the structure matcher should invoke what levels of action matching.

Chapter 6
Condition Matching and Generalization

The second part of the event matcher is the condition matcher. The condition of an event is a predicate on patterns which specifies when control may pass to the action of the event. The task of the condition matcher is to construct the appropriate predicates from sets of patterns.

For example, in the RETRIEVE procedure of the scenario, NODDY must construct branch conditions by which it can decide whether it has hit a block which it should grasp and take back to the start position or hit the wall and should go to the next row. It must construct these branch conditions from a small set of feedback patterns—in the five examples from which it acquires the procedure, it hits a block 5 times in five different places, and hits the wall 4 times, in two different places. The branch conditions that NODDY constructs are: if the x component of the position is ≥ -3 then the robot should grasp and return to the start; if the the x component of the position is ≤ -4, it should move to the next row.

The choice of these conditions is not determined by the data—there are many other conditions that could have distinguished the two sets of positions. For example, if the position is on the line $x = -4$, move to the next row; if the distance of the position from the origin is less than 3.5, grasp and return, if the negative of the x component is prime then grasp and return. All of these account for the data, and could have been generated by some generalization algorithm. Since there is no "right answer" we need additional criteria for choosing an algorithm. The following three criteria are appropriate for a generalizer that acquires procedures incrementally from examples presented by a teacher:

- **Efficiency**: The space of possible conditions is very large. It is therefore essential that the algorithm search it efficiently. In particular, if a simple condition to explain the data exist, NODDY should be able find a condition faster than if there are no simple conditions.
- **Good Incremental Behaviour**: When NODDY is given new data that a current condition does not account for, it should be able to use the current condition, and the work that went into constructing it, to guide the search for a new condition that accounts for the additional data, rather than throwing away the current data and starting again from scratch.

- **Teachability:** In order to teach NODDY a procedure, a teacher will try to generate traces that will be useful for acquiring the procedure. The choice of useful traces will be easier if NODDY's generalizations are reasonably consistent with the teacher's expectations of what NODDY might infer from a trace. In particular, NODDY should not find generalizations that the teacher considers to be unjustified by the data.*

The algorithm described in the rest of the chapter was designed to meet these criteria. One important difference between the algorithm for generalizing conditions and the algorithm for generalizing actions described in the previous chapter is that the condition matcher must have negative examples in order to find any interesting generalizations. This was not necessary for the action matcher because the space of generalized actions was so constrained by the determinacy constraint that positive examples were sufficient.

In addition to constructing new conditions, NODDY must also perform two matching tasks: it must determine when one condition is a special case of another condition, and it must determine when two conditions intersect. These are necessary in order to determine when the condition matcher needs to construct a new condition.

The first section of the chapter describes the ordering on the space of conditions that is the basis of the matching and generalization algorithms. This ordering is based on an account of what constitutes a justified generalization given in chapter 7. The remaining sections describe the matching and generalization algorithms.

6.1 Ordering the Search Space

6.1.1 Patterns and Conditions

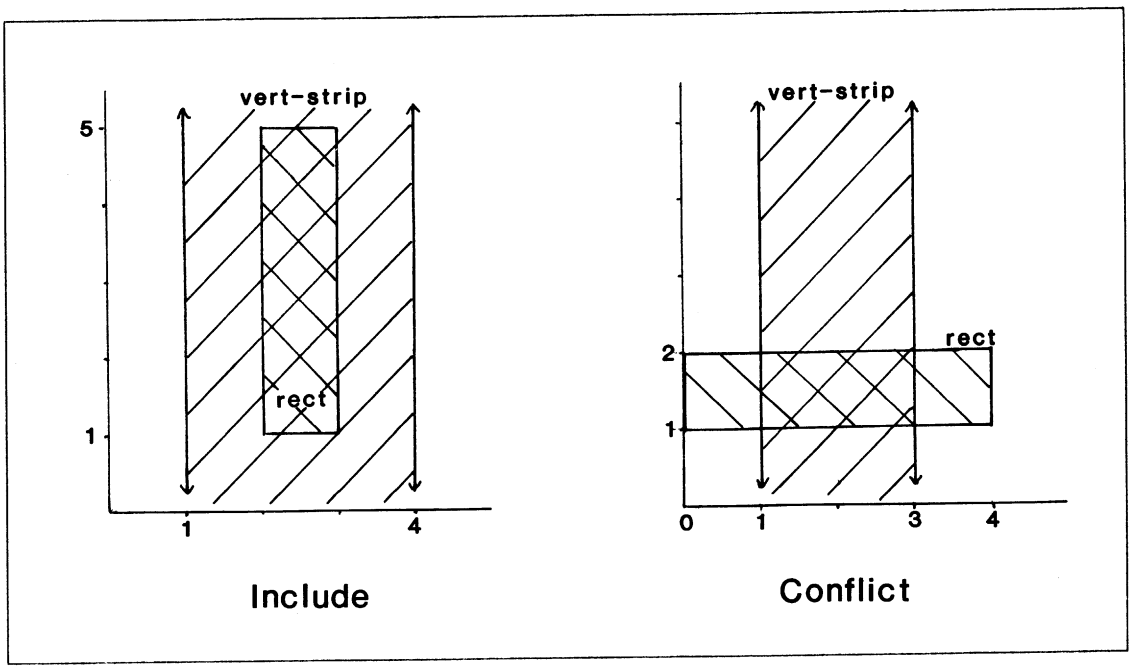
Chapter 2 described the representation of patterns and conditions.* NODDY must represent a condition as an implicit conjunction of *nodes*—instances of templates. A condition does not have explicit conjunctions or disjunctions, nor does it have explicit descriptions of predicates. The predicates are described in the templates, and the nodes have only the name of the template and the values for the parameters of the template.

---

* This does not mean that NODDY should be a theory of human generalization (and there is no claim that NODDY embodies such a theory), only that a human should not consider the product of the generalization algorithm to be strange or unjustified.

* The reader may find it helpful to refer back to the table of definitions at the end of section 2.4 to refresh his or her memory of the meanings of the terms *condition*, *component*, *node*, and *template*.

The set of templates can be partitioned according to the component of the pattern to which they correspond. Because nodes of different components are strictly incomparable, for most of the chapter, we will talk of a single component of a condition (and the corresponding set of templates) as if it were the whole condition, and talk about conditions with several components only when necessary. Most of the examples will be of position components, since that is the component with the most interesting space for the robot domain, but everything will also apply to the other components.



— — Figure 6-1 Include and Conflict relations. — —

(a) [vert-strip: $1 \leq x \leq 4$] includes [rect: $2 \leq x \leq 3, 1 \leq y \leq 5$], (b) [vert-strip: $1 \leq x \leq 3$] conflicts with [rect: $1 \leq x \leq 4, 1 \leq y \leq 2$].

There are two spaces that we must consider in searching for a generalization of a set of patterns: the space of actual conditions (nodes), and the space of templates. It will be useful to define a different ordering on each of these spaces. The appropriate ordering for the space of nodes is a generality ordering. A node A is more general than (or *includes*) node B if the set of pattern values that satisfy A is a superset of the set of pattern values that satisfy B. For example, [vert-strip: $1 \leq x \leq 4$] is more general than [rect: $2 \leq x \leq 3, 1 \leq y \leq 3$] (see figure 6-1). The space is very wide under this ordering in the sense that there are many alternate generalizations of a given node. The space is also very dense in that there are many nodes of intermediate generality

between two nodes such as [vert-strip:$2 \leq x \leq 3$] and [vert-strip:$2 \leq x \leq 4$], *e.g.*, [vert-strip:$2 \leq x \leq 3.74$]. Another important relation between nodes is the **conflict** relation: a node A conflicts with node B if A does not include B, and B does not include A, and there is at least one pattern value that satisfies both nodes. *I.e.*, two nodes will conflict if they have a common specialization. For example, [vert-strip:$1 \leq x \leq 3$] conflicts with [rect:$0 \leq x \leq 4, 1 \leq y \leq 2$] because they have the common specialization [rect:$1 \leq x \leq 3, 1 \leq y \leq 2$] (see figure 6-1).

To find a generalization of a set of patterns and/or conditions, NODDY must effectively climb up the generality ordering on nodes. There are several difficulties with doing this directly. First, the space is too large to represent explicitly, therefore NODDY must construct the relevant parts of the space as it goes from some other description of the space. Second, the space is too dense to climb one step at a time, so NODDY will have to make larger steps and backtrack to less general nodes when necessary. And third, it is too wide to determine which of many alternate direction to step in, so that NODDY will sometimes have to backtrack from a previous choice and step in a different direction.

NODDY solves all three problems by stepping through the space of templates ordered by a particular complexity measure. The complexity measure is such that stepping through a sequence of templates of increasing complexity corresponds to searching the space of nodes with the types of backtracking mentioned in the previous paragraph.

6.1.2 Complexity Ordering

The complexity ordering on the templates is based on the parameters of the predicates of the templates. The templates are ordered, first, into layers by the maximum number of interacting parameters, and second, within each layer, by the total number of parameters. That is, template T_1 is simpler than template T_2 if T_1 has fewer interacting parameters than T_2 **or** if they have the same number of interacting parameters (they are in the same layer), but T_1 has fewer total parameters than T_2.

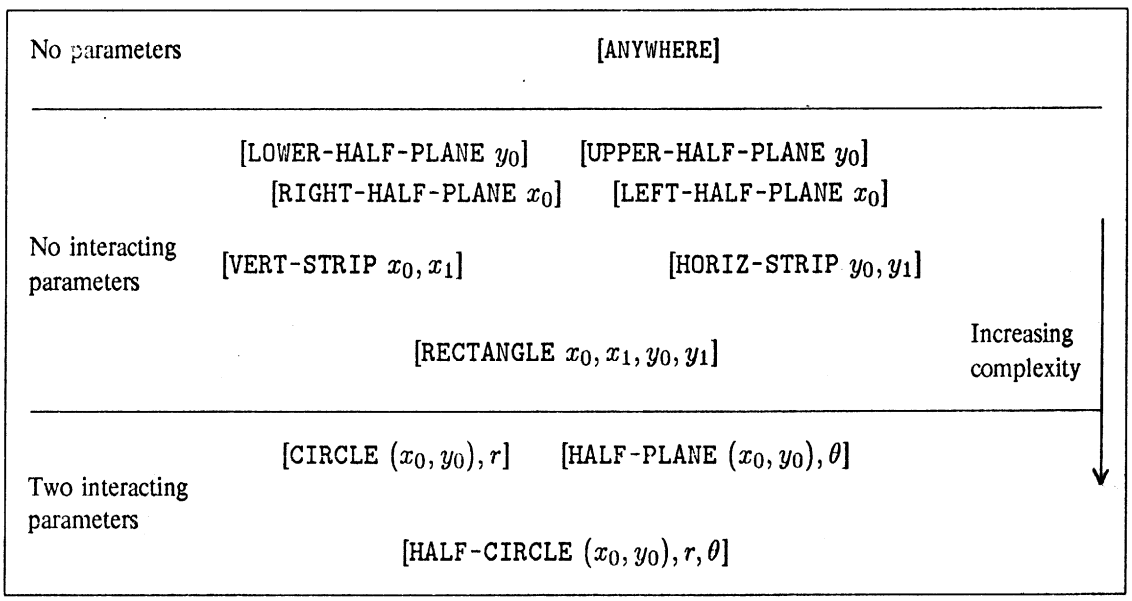
Two parameters of a template interact if, when creating an instance of the template that satisfies a set of pattern values, choosing a value for one parameter affects the choice of the other parameter. For example, the x_1 and x_2 of [vert-strip:$x_1 \leq x \leq x_2$] can be chosen quite independently—x_1 could be chosen to be any value up to the minimum x-component of the pattern values, and x_2 any value over the maximum. On the other hand, the $\langle pos \rangle$ and $\langle r \rangle$ of [circle:$|(x, y) - \langle pos \rangle| \leq \langle r \rangle$] interact because the minimum

value for the radius <*r*> depends on where the the center of the circle is placed.

The motivation for this ordering (which will be discussed more fully in chapter 7) derives from the fact that the number of data points required to determine the appropriate instantiation of a template increases with the number of parameters, especially the number of interacting parameters, in the template; and the number of possible templates of a given complexity increases with the complexity. Therefore, searching according to increasing complexity under this measure will ensure that a more complex templates will only be considered when there are enough data points to: (i) eliminate all simpler templates; (ii), distinguish between the many templates of simpler complexity, and (iii), determine the appropriate instance of the template. This complexity ordering is therefore based on the complexity of *constructing* a description from the evidence, not on the complexity of *using* the description once it has been created. The differences between these two types of complexity will also be discussed in chapter 7.

One consequence of this ordering is that conjunction of predicates is a very "cheap" operation. That is, a template whose predicate is just a conjunction of two other predicates is considered only slightly more complex than the templates of the constituent predicates since the parameters of the two conjuncts do not interact and the conjunctive template will be in the same layer as the constituent templates (or the layer of the more complex template if the constituents are in different layers). Adding a conjunct to a predicate corresponds to specializing the condition, so that stepping along the complexity ordering within one layer corresponds to the first type of backtracking in the generality ordering in the space of nodes—specializing a node after overgeneralizing. Moving to a new layer of the template space involves a larger step in complexity and corresponds to the second type of backtracking in the space of nodes—abandoning one line of generalization and trying a different direction.

Figure 6-2 shows the small part of the template space, ordered by complexity, that NODDY currently knows about for the position component of conditions in the 2-D robot domain. Note that the template `[anywhere]`, which has no parameters and therefore only one instance, namely the condition that is always satisfied, has a special position at the beginning of the ordering. There are many more templates that could be constructed using the same set of operators, but this has not yet been done. There are many procedures that could not be acquired with this limited set of templates, but it has been sufficient for acquiring some procedures, and could easily be extended as needed. Clearly, both the



— — Figure 6-2 The Template Ordering for the 2D Robot Domain. — —

templates themselves and the ordering of templates are dependent upon the language for representing the predicates.*

6.1.3 Content of Templates

The defining information in a template is a specification of the predicate it represents and a list of the parameters and their types. To use the templates to match and generalize conditions, NODDY needs other information about a template: its place in the complexity ordering, predicates to determine the relations of instances of the template, and functions for constructing instances of the template.

If NODDY could reason about the algebraic constraints in the predicate, it could derive these other predicates and functions in the template from the description of the basic predicate. However, the current implementation of NODDY cannot do this reasoning, so the template must contain most of this other information explicitly. NODDY is able

* Note that this particular template ordering may not conform very well to human intuitions of a complexity ordering on the same regions of a plane because the primitive operators include `x-part` and `y-part` which assume fixed cartesian coordinates for the plane. As a consequence, the half planes aligned with the axis are considered very simple regions. The teachability of NODDY would probably be improved by a choice of primitive operators more compatible with the human intuitions.

to reason about conjunction, so that it can derive some of the predicates and functions of templates representing conjunctive conditions from other templates. Much of the complexity of the matching and generalizing algorithms described in the following sections arises from this reasoning about conjunctive templates.*

Each template states explicitly which layer of the complexity ordering it is in. The ordering within each layer can be trivially determined from the number of parameters. The complexity ordering on the nodes (instances of the templates) is implicit in the explicit relations between templates combined with predicates attached to the templates that determine the relation between instances of the templates.

Templates come in two varieties: basic templates, which are the simplest templates within each layer, and the derivative templates, whose defining predicate is derived from that of other templates by conjunction or projection. Each template must state whether it is a basic template, and if not, must state how it was derived, and what templates it was derived from.

The predicate of a basic template cannot be expressed as a conjunction of simpler predicates, and each parameter only occurs once in the predicate description. Each basic template must have a *self-include* predicate to determine whether one instance of the template is a strict generalization of another instance. For example, the `CIRCLE` template is a basic template, with two parameters—a position and a distance—and it has a self-include predicate that states that a circle node [`circle:` $|(x,y) - \langle center_1 \rangle| \leq \langle radius_1 \rangle$] includes another circle node [`circle:` $|(x,y) - \langle center_2 \rangle| \leq \langle radius_2 \rangle$] when

$$|(\langle center_1 \rangle - \langle center_2 \rangle)| \leq (\langle radius_1 \rangle - \langle radius_2 \rangle)$$

The basic templates must also have predicates specifying the relation between instances of it and instances of other templates. The final bit of information that basic templates must have is a function that will construct the most specific instance of the template that covers a given set of pattern values.

Other predicates can be constructed out of the basic templates by conjunction and projection. A template C is a *conjunctive derivative* of T if the predicate of C is a

* In retrospect, it would have been cleaner to separate the derivation of the predicates and functions of conjunctive templates from the use of those predicates and functions once derived. Future versions of NODDY should separate them more clearly.

conjunction, and one of the conjuncts is identical to the predicate of T. The self-include predicate of a template that is a conjunctive derivative of other templates can be simply derived from the conjunction of the self-include predicates of the constituent basic templates. P is a *projective derivative* of T if its predicate is identical to that of T except that two or more of the parameters of T have been replaced by a single parameter in P. Again, the self-include predicate of a projective derivative of a template T can be trivially derived from the self-include predicate of T. NODDY can perform both of these derivations.

For example, the [`vert-strip`:$x_1 \leq x \leq x_2$] template is a conjunctive derivative of the [`right-half-plane`:$x \geq x_1$] and the [`left-half-plane`:$x \leq x_2$] templates.

With these relations and predicates, NODDY can determine whether a node that is an instance of one template is a generalization or specialization of a node that is an instance of any other template. Each derivative template must have one final predicate that determines whether a particular set of parameter values represents a valid instance of the template. The following sections describe the way the condition matcher uses this template ordering and these predicates.

6.2 Matching Patterns and Conditions

There are two modes in which the structure matcher invokes the condition matcher simply as a matcher—first to determine whether two conditions are equal or one is a generalization of the other, and the second to determine whether two conditions conflict, *i.e.*, whether they have a common specialization (though the matcher does not have to construct the specialization, just determine whether it exists). Both tasks involve reasoning about the generality ordering on conditions which is implicit in the information in the templates.

To match (or generalize) the conditions, NODDY considers each component of the conditions separately. There are three trivial cases for matching two nodes (condition components)—(a) the nodes are identical; (b) one node is the always-satisfied node (*e.g.*, [`anywhere`], for the position component) which is more general than any other node, and conflicts with nothing; and (c) one node is a primitive pattern node, which conflicts with nothing, and is included by another node if it satisfies the basic predicate of the other node. When the nodes are neither primitive pattern nodes nor the maximally general node, the matching is more complex.

6.2.1 Matching Condition Nodes: Include

To test whether a node A includes another node B, NODDY first determines the relation of the templates of A and B (T_A and T_B).

If A and B have the same template, then NODDY applies the self-include predicate of the template to the two nodes. If the template is not a basic template, NODDY will have to derive the predicate from the self-include predicates of the basic templates of which the template is a conjunctive derivative. NODDY does not actually derive the predicate explicitly, but breaks the node down into its constituents, as described in the next paragraph, and checks that all the constituents of A include the corresponding constituents of B.

If T_B is a conjunctive derivative of (or a derivative of a derivative of ...) T_A, then T_B's predicate is a conjunction of T_A's predicate and some other predicate. NODDY constructs a new instance of T_A from the node B by dropping out the parameters of B that are irrelevant to T_A, then applying the self-include predicate of T_A to A and the new node. If the predicate is satisfied, then A includes B.

For example, let A be [`vert-strip`:$1 \leq x \leq 4$] and B be [`rect`:$2 \leq x \leq 3, 1 \leq y \leq 5$] (refer back to figure 6-1). The `RECTANGLE` template states that the first two parameters of the `RECTANGLE` predicate came from the `VERT-STRIP` template, and the second two from `HORIZ-STRIP`. Therefore, [`rect`:$2 \leq x \leq 3, 1 \leq y \leq 6$] is effectively a conjunction of the nodes [`vert-strip`:$2 \leq x \leq 3$] and [`horiz-strip`:$1 \leq y \leq 6$]. NODDY constructs [`vert-strip`:$2 \leq x \leq 3$], and uses the self-include predicate of the `VERT-STRIP` template to determine that [`vert-strip`:$1 \leq x \leq 5$] is a generalization of [`vert-strip`:$2 \leq x \leq 3$] and therefore of [`rect`:$2 \leq x \leq 3, 1 \leq y \leq 6$] also.*

A similar method would work for projective derivatives, but has not yet been implemented.

If T_B is not a derivative of T_A then NODDY looks in T_A for a predicate for determining the relation of nodes of type T_A and type T_B. For example, the `LEFT-HALF-PLANE` template might have a predicate for determining when a `left-half-plane` node includes a `circle` node. If there is no predicate for T_A and T_B, NODDY assumes that A does not include B. This assumption may be wrong, but it is the safe assumption because

* Note that nodes do not contain descriptions of the predicate, only a list of parameter values. The predicate is given here as a mnemonic aid.

the structure matcher takes more conservative actions if it believes the nodes do not match. It may miss some key pairs that it could have found had it known more, but it will not construct an invalid key pair. It will also think that some forks are nondeterministic and do extra work to resolve them when the fork was actually an (ordered) deterministic fork, but, again, this is a conservative choice which will not introduce errors into the new procedure.

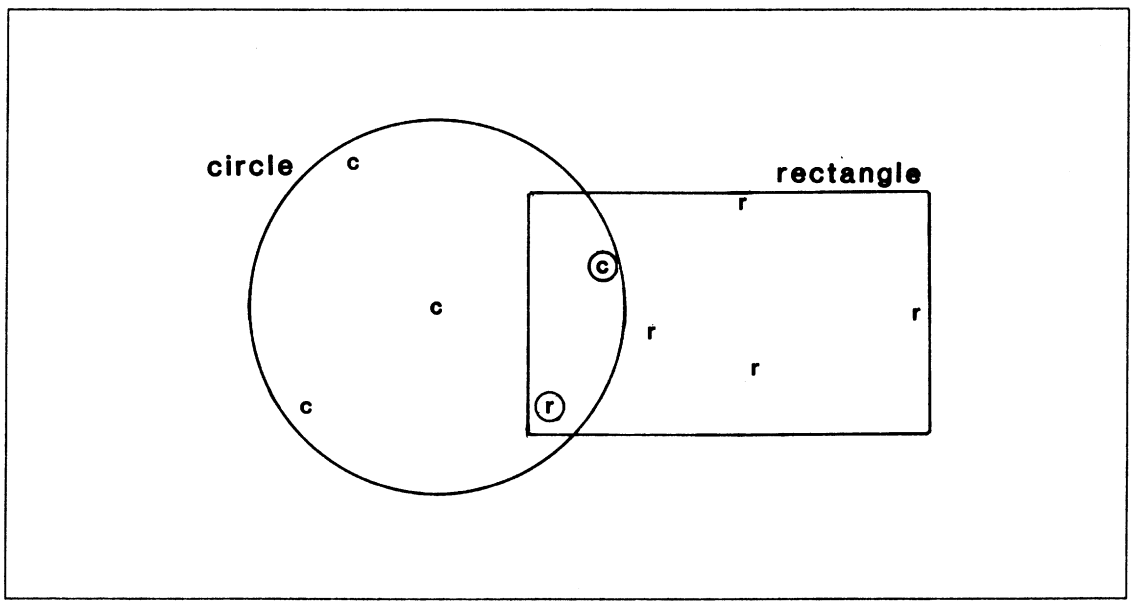
6.2.2 Matching Condition Nodes: Conflict

Two conditions conflict if there are any patterns that satisfy both conditions, and neither condition is a generalization of the other. There are two ways in which two conditions, C and D, can conflict. The first is that some nodes of condition C are more general than the corresponding nodes of condition D and other nodes are less general than their corresponding nodes in D. For example, the condition [`anywhere contact` 90°] conflicts with the condition [`at:` $(0,0)$ `any-contact-dir`]. NODDY can test for this type of conflict readily by comparing the nodes as in the previous section, and comparing the results for each pair of nodes.

Two conditions can also conflict if any pair of corresponding nodes conflict. For example, [`circle:`$|(x,y)-(0,0)| \leq 2$] conflicts with [`circle:`$|(x,y)-(1,0)| \leq 2$] since the two circles overlap. Checking for this type of conflict is more complicated, and would require a sophisticated algebraic or geometric reasoner to cope with the general case. However, NODDY can cope with many cases by using information in the templates.

NODDY first checks that neither of the nodes (A and B) includes the other. If the templates (T_A and T_B) of the two nodes are the same, the template may have a predicate for determining when two instances of the template conflict, which can be applied to the two nodes. T_A may also have a predicate for determining when an instance of T_A conflicts with an instance of T_B (though there are no such predicates in the current templates for the robot domain). If there is a template T_C which is a conjunctive derivative of both T_A and T_B, NODDY can form an instance of T_C using the parameters of the two nodes. If this new node is a consistent instance of T_C, then the two nodes have a common specialization, and therefore conflict.

If none of these methods are available, NODDY checks whether any of the past values of A are covered by B and whether any of the past values of B are covered by A. If either is the case, the nodes conflict (see figure 6-3). If neither node includes any of the



— — Figure 6-3 Determining Conflicts between Condition Nodes — —

Conflicting circle and rectangle nodes. The c's are the past values of the circle node—the positions that the circle accounts for—and the r's are the past values of the rectangle node. Each node covers one of the past values of the other node, therefore the nodes conflict, and the condition matcher must find new conditions that cover the same past values without conflicting.

past values of the other node, NODDY assumes, for lack of information, that the nodes do not conflict. There may, in fact, be an intersection of the conditions, but none of the pattern values NODDY has seen so far have occurred in the intersection region. This means that NODDY may leave a nondeterministic fork in the procedure without realizing it, but this is not disastrous since it may be that there will never be a pattern value in the intersection of the nodes, in which case the nodes are effectively disjoint, or a future example may include a distinguishing pattern value, in which case NODDY will discover the conflict, and will fix the fork.

When there is a conflict, NODDY checks that none of the past values of A are identical to any of the past values of B. As long as the past values are all distinct, the condition generalizer may resolve the conflict by finding new conditions. If any past value of A is identical to a past value of B, then the conflict is a more radical conflict: there is at least one pattern value that both conditions must cover. This means that each of the branches have been taken in the past with exactly the same pattern and it is not possible to construct a predicate on the patterns that will distinguish the two branches. NODDY

is not able to resolve such conflicts currently.

6.3 Generalizing Patterns and Conditions

When the structure matcher is trying to merge two events, and the conditions do not match perfectly, or when there is a conflict between the branch conditions of a fork, the structure matcher will invoke the condition matcher with the conditions and ask for a generalization that resolves the difference or the conflict. The search strategy of the condition matcher is to look for the most specific instance of the simplest condition that accounts for the data, if there is an unambiguous choice, and to abandon the search and wait for more data if there are several equally good choices.

There are two levels to the condition generalization. The first level, which is essentially trivial, is invoked with positive data only. That is, the structure matcher hands the condition matcher a pair of conditions and asks for a generalization of them. The second level is invoked with both positive and negative data so that the condition matcher must find a generalization of the positive data that excludes the negative data.

Both levels match the components of the conditions independently, and do not attempt to generalize components which already match (whether equal or one including the other). The rest of this section will speak as if the condition had only one component.

6.3.1 First Level Condition Generalization

The simplest condition is the always-satisfied condition—`[anywhere]` for the position component—which will account for any set of positive data. Therefore the first level generalization always returns the always-satisfied condition.

This means that the condition matcher can always find a generalization of two conditions, and, therefore, the conditions do not constrain the pairing of events at all during the propagation and grouping stages—all the constraint comes from the actions.

There are two problems with the current implementation of NODDY. The first is that the structure matcher is not sufficiently sophisticated during the grouping stage to distinguish between two groups of conditions: those that can be generalized to the always-satisfied condition because they are not branch conditions and do not have to be disjoint from anything; and those that are branch conditions, and cannot be so generalized without introducing conflicts with the other branch conditions. Section 4.4.1 described

this problem in more detail and outlined how NODDY should be modified. In particular, the structure matcher should hand the condition matcher a list of positive examples and negative examples when generalizing branch conditions.

In the current implementation of NODDY, the condition matcher tries to guess when the condition might be a branch condition that should not be generalized to the always-satisfied condition and do a more conservative generalization. This heuristic is not reliable, but when it guesses wrongly, the only result is that the resulting procedure may be less general than it needs to be.

When matching a generalized condition (other than the always-satisfied condition) to a primitive pattern that is not covered by the condition, NODDY assumes that the condition is a branch condition because the only reason for a generalized condition not being the always satisfied condition is that it would conflict with other branch conditions. NODDY guesses that the primitive pattern is not one of the branch conditions in conflict with the other condition, and therefore assumes that it should not generalize the condition to the always-satisfied condition, and, instead, searches for a more general instance of the same template that includes the primitive pattern. For example, if the condition were [`vert-strip`: $2 \leq x \leq 5$] and the primitive pattern were [`at`: $(6, 3)$], NODDY would construct the more general [`vert-strip`: $2 \leq x \leq 6$] which is the most specific instance of the `VERT-STRIP` template that accounts for the data.

This is consistent with the general search method because NODDY can assume that there are no simpler templates that would account for the data without causing conflicts with the other branch conditions. This generalization does not guarantee that the branch conditions will remain disjoint, since a more complex template may be necessary to accommodate the new pattern without conflicts, but there is a chance that it does, and the second level condition generalization will always have to do less work to resolve any remaining conflicts than if the first level had simply generalized to the always-satisfied condition.

The second problem with the first level condition generalization is that the current representation of pattern components that may have null values (*e.g.*, the contact direction component, which is null when the robot is not in contact with anything) does not allow the representation of a condition that is satisfied by a pattern of any value or a null value. This means, for example, that NODDY cannot currently find a generalization of the conditions [`at`: $(1, 1)$] and [at: $(1, 1)$ contact: $30°$] because there is no way of

representing an `any-or-null-contact-direction` node. The effect of this on the TURTLE procedure (see figure 1-11) would be that if there were an obstacle at the final position in any trace, NODDY would have to construct a procedure with two separate stopping conditions—one that stopped at $(0,0)$ with no contact and one that stopped at $(0,0)$ and any contact. A future version of NODDY could resolve this issue either by representing null pattern components explicitly or by extending the matcher to deal with implicit null pattern components correctly.

6.3.2 Second Level Condition Generalization

The second level condition generalization always has negative data, as well as the positive data. The action generalization did not require negative data because the determinacy constraint restricted the space of possible generalizations sufficiently. There is no such determinacy constraint on conditions, so the negative data is essential for constraining the candidate generalizations sufficiently to make an unambiguous choice.

When branch conditions of a fork conflict, the structure matcher invokes the condition matcher on the conditions involved in a conflict. The reason a conflict arises is that NODDY made an inadequate generalization at some earlier stage. In particular, some of the conditions are more general than they should be, so the condition matcher must backtrack in generalization space to find alternative generalizations that will not cause conflicts. Each condition is always the most specific instance of its template that covers the data, so it will not be possible to simply replace the conditions by more specific instances of the same templates. Rather, NODDY will have to find instances of more complex templates.

The structure matcher invokes the condition matcher separately on each condition involved in a conflict. Along with the condition, which we will refer to as the *current condition*, it also hands the condition matcher a list of all the other branch conditions with which the current condition conflicts—the *conflict conditions*. The task of the second level condition generalization is to find a new generalized condition to replace the current condition that covers all the past values of the current condition, but does not conflict with the other conditions.

The condition matcher does a breadth first search through the space of templates ordered by their complexity, starting at the template of the current condition. At each complexity level, there will be a set of candidate templates. For each candidate template, it

will construct the most specific instance that covers the same data as the current condition (*i.e.*, the past values of the current condition). It then rejects as inconsistent any of these new nodes that conflict with any of the conflict conditions, and rejects as redundant any that are a generalization of any other candidate. If there is just one remaining new node, this is the new condition, because it is unambiguously the most specific instance of the simplest template that accounts for the data. If there is more than one remaining new node, then the choice is ambiguous, so NODDY abandons the search and waits for more data. If there are no consistent new nodes, the search proceeds to the next complexity level.

The condition matcher need not consider all the templates in the same layer as the template of the current condition (the current template)—it need only consider the templates that are conjunctive derivatives of the current template. This is because the current template was the only template of its complexity that could satisfy the previous data, so that conjunctive derivatives of its neighbours alone could not satisfy the new data. If none of the conjunctive derivatives of the current condition satisfy the new data, the condition matcher searches in the next layer, which is unrelated to the current condition. For example, if the current template is `upper-half-plane`, the `lower-half-plane`, `left-half-plane` and `right-half-plane` must have been rejected as not satisfying the data on which the current condition was based. Therefore, there is no need to consider the the `VERTICAL-STRIP` template as a candidate for the new condition, because it could not satisfy the old data, and therefore, not the new data either. The candidates that the condition matcher would consider from this layer would be `horizontal-strip` and `rectangle` because these are both conjunctive derivatives of `upper-half-plane`

In terms of the generality ordering on conditions, this search method is actually backtracking from the current condition and choosing a more complex condition to explain the new data. However, because of the structure of the ordering on the templates, the search method looks as though it is overgeneralizing then specializing, until it cannot specialize any more. It will then make a new jump in generality space, corresponding to stepping to a new layer in the complexity ordering.

Chapter 7

Generalization Principles

This chapter discusses several key ideas that underlie the design of NODDY. Although NODDY is concerned with acquisition from examples, these ideas apply to any task which involves matching or generalizing descriptions. The difference between matching and generalizing is one of emphasis, rather than essence: Matching is concerned with finding a measure of the similarity of two descriptions, and perhaps determining their differences also. If the descriptions have structure, a matcher may also produce a correspondence between the parts of the descriptions. Generalization, on the other hand, is concerned with producing a description which captures all the similarities of the descriptions and ignores the differences. A matcher, therefore, must at least implicitly construct a generalization of the descriptions, and a generalizer must determine how well the two descriptions match, even if it does not place some measure on the goodness of the match. Therefore, the following ideas apply equally well to a task centered on matching, such as reasoning or learning from analogy, or a task centered on generalization, such as acquiring procedures from examples.

The first key idea is the principle of exploiting the constraints of the domain. This is a standard technique in AI, but it instructive to examine just how effective it can be in a learning task. The second key idea is a set of three constructive generalization techniques. The third key idea is the principle of using justification to limit generalization. This is the most significant principle arising out of NODDY.

7.1 Exploiting Constraints

It is generally accepted that a very powerful technique for solving many different problems is to determine the constraints inherent in the problem domain and construct a solution built around those constraints. (A classic example of the power of this technique was Waltz's line drawing program. By analyzing the constraints on line drawings representing real objects, Waltz was able to devise a simple algorithm that exploited the constraints to interpret line drawings.)

Exploiting domain constraints is especially important for tasks involving generalization since generalization itself is usually underconstrained. Acquisition tasks which must

generalize from positive instances only are particularly underconstrained, and one must provide very strong constraints on the possible generalizations to acquire anything.

Angluin, [1980] and Berwick, [1982], among others, have explored this question from a formal perspective in the area of language acquisition. They have shown that if the set of candidate grammars is constrained to satisfy a certain condition, then one can acquire a grammar from positive examples only just by choosing the most specific grammar consistent with the examples.

This degree of constraint, though obviously desirable, is not available in all domains, and a learning system must exploit less constraining constraints in different ways. NODDY illustrates at least three ways in which a learning system can exploit domain based constraints:

(a) Reduce the space of possible generalizations;
(b) Construct negative examples out of positive examples;
(c) Guide a structure matching process to eliminate combinatorial explosion.

7.1.1 Reducing the Generalization Space

If the major problem of generalization is how to choose between alternative candidates, then any reduction of set of candidates will make the choice easier. The choice become trivial if the set of candidates can be reduced to a single candidate or to a set of candidates which can be ordered by generality such that there is just one most specific candidate.

The constraints on natural language that Berwick [1982] exploited in constructing LParsifal were able to reduce the candidate rules to a single most specific rule in most cases.

The constraints on procedures are not as tight as those on the syntactic structures of natural language, and NODDY cannot exploit them quite as simply. However, determinacy constrains the space of generalized actions very considerably. The effect is that, when there are any at all, there is usually only one most specific generalization of a set of actions, and frequently there are no generalizations at all. This is in sharp contrast to generalizing conditions where there is always a generalization of any set of patterns, and there may be very many most specific candidate generalizations.

The determinacy constraint eliminates generalized actions that cannot be expressed either as a primitive action whose parameters are a function of a previous pattern, or as a primitive action whose guard condition is a generalization of the features of the domain

that are specified to be monitorable.

NODDY is critically dependent upon this constraint. Although the structure matcher chooses candidate pairs of events on the basis of the contextual justification, this is not sufficient evidence to guarantee that the events should be merged. The event matcher must provide further justification for merging them by finding a generalization of the events. If there were a generalization of any pair of actions, then any two events could be matched, and the new procedure would degenerate to a single loop containing one very general event. The determinacy constraint prevents this.

7.1.2 Constructing Negative Examples

There is no analogous constraint on conditions to restrict the space of candidate conditions, and so there are many possible generalizations of a set of patterns. NODDY resorts to a simplicity ordering on the space of conditions to govern the choice between candidates. But even this is not sufficient—any set of patterns has a generalization which is also the simplest condition, namely, the always satisfied condition. To find any more useful condition, the generalizer must have positive and negative examples.

A similar problem will arise in any domain where there is insufficient constraint to create a narrow generality ordering on the generalization space and there are simple, very general, generalizations. If the environment or the teacher do not provide any negative examples, the matcher/generalizer must create its own. When the problem arises for elements in structured descriptions, one method of creating negative examples is to exploit constraints on groups of the elements.

In the procedure domain, the determinacy constraint requires that not only each action, but also the choice at each fork be deterministic. This is a constraint on the group of branch conditions of the fork, which requires that each pair of conditions be disjoint.* Therefore, when it is trying to construct a branch condition, NODDY can exploit this constraint by using the patterns corresponding to the other branch conditions as negative examples for the condition it is constructing. With these negative examples, and the ordering discussed later, NODDY is able to find an unambiguously best choice from

* Actually, NODDY allows one condition to be a strict generalization of the other. It satisfies the determinacy constraint by assuming that the more general condition contains an *implicit exception* clause corresponding to the more specific condition. With the exception clause, the conditions are actually disjoint, and the conditions are simpler than if the exception clause were made explicit.

among the candidate conditions.

In general, any constraint on groups of elements of a structured description can constrain the generalization of those elements. If the constraint can be construed as requiring the elements in the group to be disjoint in some sense, then the constraint can generate negative examples for generalizing the elements.

7.1.3 Guiding Structure Matching

A third way that a matcher/learner can exploit constraints is in the matching of structured descriptions. A central problem with matching two structured descriptions is to determine the appropriate pairing of the nodes of the descriptions. There are exponentially many possible pairings, so it would be very expensive, computationally, to consider each possible pairing.

If there is some constraint on the descriptions such that there are "important" relations in the description which provide a "skeletal structure" to the descriptions, the structure matcher can exploit these relations to guide the choice of pairings. The *succeeder* relations in procedures impose a very tight structure on the events of a procedure, which NODDY exploits by tracing along the **succeeder** relations from paired events to find other candidate pairs.

Winston [1980] uses in the same principle in constructing analogies, where the **cause** relation provides the "skeletal structure" to guide the pairing of nodes. The **cause** relation not only suggests pairs of nodes to match, but also selects the important nodes out of the description—the nodes not involved in **cause** relations are assumed to be of secondary importance, and the structure matcher does not pair them until it has paired as many of the nodes involved in **cause** relations as possible. Gentner [1983] argues that any higher order relations—relations between relations—could play this role, at least for analogical matching.

7.2 Constructive Generalization Methods

The ways of generalizing a description depend on the nature of the representation languages of the description and the generalizations. Michalski [1983] gives several generalization methods for a variety of cases in which the representation language for the generalizations is either identical to, or just a small superset of the representation language of the descriptions.

NODDY embodies three generalization methods which produce generalizations in a representation that has significant components not present in the descriptions being generalized. The first involves inferring groups in structured descriptions, the second involves inferring relations between nodes in structured descriptions, and the third involves generalizing parameterized objects.

7.2.1 Inferring Groups in Structured Descriptions

Groups are a very powerful representational device for structured descriptions. For example, without some type of group it is impossible to construct the general description of the class of tables with 1–6 legs, since one must be able to talk about the group of legs in order to talk about its cardinality.

Acquiring structured descriptions with groups poses a major problem for a learner when the examples do not contain explicit groups. The first problem, given an example description, (or several descriptions) is to determine the extent of the group—what items belong to the group. For example, given a description of a table and four legs, the learner must determine whether there is a group at all, and if so, whether it includes just two of the legs, all of the legs, or the legs and the top. The second problem is to determine what the typical member of the group is—is the group a group of pairs of legs (which would ultimately exclude all tables with an odd number of legs) or is it a group of single legs. The third problem is to infer the constraints on the group, for example, a limit on the cardinality of the group.

Groups arise in procedures in the form of iterative loops where the typical member of the group is the sequence of events in one iteration. NODDY does not have a full solution to the problem of finding groups. In particular, it has not addressed the issue of finding constraints on groups (for example, the number of iterations in a counted loop), and has only a partial solution to the problem of identifying the typical member of a group.

The existence and extent of a group can be found from matching two examples by allowing the matcher to pair a node in one description with more than one node in the other description. Each set of multiple pairings will generate a group of nodes—all the nodes paired with a single node, and all the nodes paired with them, and.... Several groups of nodes may be combined into a single group of objects if each node of one group of nodes is connected by the same relation to a corresponding node of the other group of

nodes. For example, in matching an office chair with five legs against another office chair with four legs, the matcher might construct a group of all the legs and another group of all the casters. It would then notice that each caster is connected to a leg and construct a group of caster-and-leg's.

Allowing multiple pairings increases the complexity of the matching task considerably* so that the matcher must use some method of controlling the matching to prevent an inordinately expensive match. One such method is NODDY's propagation technique which limits the pairings to those that can be justified by the context of the evolving match.

For finding loops, this method has a bias towards finding the largest typical member possible, (the loop with the most events). This means that the teacher must identify the boundary of the the intended group by presenting examples that differ by just one element of the group (one iteration of the loop). It would be possible to eliminate this requirement by using a simple exhaustive search to find the smallest typical element once the boundaries of the group have been found. Assuming the number of elements in the groups will be small, the search need not be expensive.

7.2.2 Inducing Relations

An essential component of structured descriptions is the relations between the nodes in the description. A difficult type of generalization is inferring relations that are not explicitly present in the input data. One class of relations are those that hold between parameterized objects—objects whose description includes a fixed set of slots whose values are taken from structured domains—and can be represented by expressions applied to the parameters of the objects.

Inducing these relations involves determining which nodes may be involved in a relation (discussed in section 7.3) and finding the relation by searching through the space of expressions. The expressions are built out of constants taken from the domains of the parameters of the parameterized objects, and a set of operators that can be applied to parameters, constants, or results of other operators.

If the set of operators is small (relative to the domains of the parameters) the space

* There are $n!$ ways of pairing each node of an n node graph with exactly one node of another n node graph, but $2^{n^2} \approx (n!)^{n/\lg n}$ ways of pairing each node with any number of nodes of the other graph!

of expressions *without* constants can be searched quite simply by incrementally composing operators and expressions into more complex expressions. When the space includes expressions with constants as well as operators, the search space becomes infinite and cannot be searched directly. The problem of finding constants for the expressions is another form of the "invisible objects" problem. For example, given a set of pairs of real valued parameters (x_i, y_i), which all happen to be related by a linear function $y_i = ax_i + b$ the values of a and b are not present in the (x_i, y_i) pairs, and are therefore "invisible" to the generalizer in the input data. To find the expression $y = ax + b$ the generalizer must either consider all possible values for a and b (impossible if the constants are real valued), or determine their values by some special means. It is not possible to have heuristics for finding these "invisible" constants in all possible expressions, but one could construct heuristics for a small set of common expressions.

NODDY addresses this problem in inferring functional relations between actions and earlier patterns. NODDY does a breadth first search through the space of expressions by building expressions "from both ends" of the data. It is able to find one invisible object for each expression by exploiting the restriction that all the operators must be invertible. It also uses the context from which the data was drawn to suggest good guesses for possible constant values which can be used to build expressions without having to search the entire domain of the constants. This algorithm is very general, and applies equally well to numerical and non-numerical parameters, given that the operators and their inverses can be provided.

An alternative method of finding the invisible objects is that of Langley [1981a,b 1983] which uses domain specific procedures for determining the values of the constants for specific types of expressions that are known to be likely for that domain. For instance, Langley has a heuristic that will determine whether the x_i and y_i are linearly related, and if so, what the values of a and b are. Clearly, this particular procedure would not apply to a non-numerical domain, and a different set of procedures would be needed. However, this method can find many more invisible objects, essentially by knowing exactly where and how to look for them.

This method of building expressions also applies to generalizing parameterized objects by constructing expressions representing predicates on the objects. NODDY does this kind of generalization in generalizing the branch conditions of procedures, but instead of constructing the expressions directly, it uses the method described in the next section.

7.2.3 Generalizing Parameterized Objects: Climbing the Template Hierarchy

Searching through expression space by building expressions incrementally is a very general method for generalizing parameterized objects, but it is also very expensive. A cheaper generalization method uses a hierarchy of templates representing parameterized classes of expressions. To find a generalization of a set of parameterized objects, the generalizer climbs the hierarchy to find a template containing an instance that is a generalization of the objects. The templates must contain not only the parameterized expression, but also the predicates or functions necessary for constructing an instance of the template that is a generalization of a set of examples. The selective generalization method of climbing an AKO hierarchy is superficially similar to this method, but the nodes of an AKO tree are descriptions rather than templates, so computing an instance is trivial.

NODDY uses the climb template hierarchy method for both generalizing conditions and for the first level action generalization. There are two issues to address in using the climb template hierarchy method: Ordering the templates and constructing the hierarchy. For constrained domains such as actions in the robot domain, a generality ordering on the templates is sufficient. For more complex domains, a generality ordering is not sufficient. The next section will discuss the template ordering used in the condition generalization.

The template hierarchies in NODDY are small and were constructed by hand. Once an ordering criteria and a language for the expressions have been established, constructing the template hierarchy by hand is straightforward. NODDY has not addressed the issue of constructing the hierarchy automatically. It is a relatively simple task to construct the parameterized expression represented by the template, by building expressions out of operators and variables, instead of operators and constants, but it is much harder to construct the various predicates and functions based on this expression that are necessary for actually constructing instances of the templates.

7.3 Justification

The third key principle embodied in NODDY is that of requiring justification to limit generalization. The task of a generalizer is to find a generalization of a given set of data. This task is underconstrained because there are always many possible generalizations of a

given set of data. A generalizer must have some basis for either choosing one of them or rejecting them all.*

A generalization of a set of data can be viewed as a theory of that data. A key consideration for choosing or rejecting a theory is whether the theory is *justified.* Intuitively, there are several ways in which a theory may be unjustified:

- If there are many equally good theories, an arbitrary choice of one of them would be unjustified. For example, a group of a suitcase and a watering-can could be explained by their both being made of plastic, both having handles, both containing stuff, both standing on the floor, or ... Since there are so many possibilities that all seem equally good (in the absence of any other information), there is no way to choose one over the others.
- If the best theory is so tailorable that it could have been made to fit any similar set of data, then the theory is unjustified by the data. For example, if the data consists of three points, then there is a circle that can be put through those three points, However, we have no basis for believing that the circle explains the data, since the parameters of the circle could have been tailored to fir any three points.

To account for these intuitive perceptions of when a theory is justified, we must give a basis for the idea of the "goodness" of a theory, and an account of what justifies or does not justify adopting a theory. We claim that these are both found in the degree to which the data constrains a theory, and how much data a theory needs to constrain it.

7.3.1 An Ordering on Generalizations

Any theory or generalization must be described in some language which will have a set of terms and relations and a set of conventions for composing the terms and relations. We can partition the space of all possible descriptions into classes based on the form of the description—a description will belong to the class corresponding to that description with each term or relation replaced by a formal parameter. Choosing a generalization involves choosing a class of descriptions, then choosing values for the parameters. The number of parameters (and their types, if they are typed) determines how tailorable the theory is—how many degrees of freedom it has. In the absence of any special ordering on

* In rejecting all the possible generalizations, the generalizer need not throw away all the information it has gained in the process—it could keep the partial results around to use when it acquires more data.

the terms, all the generalizations in the class would be considered equally good, and the choice of a generalization from this class could not be justified unless there were sufficient data and other information to select just one member of the class by determining the parameters uniquely. The more degrees of freedom a class has, the more data will be required to determine values for all the parameters, and the less the justifiability of the member of the class.

This measure of justifiability immediately leads to an ordering on the space of generalizations—the generalizations with few degrees of freedom, and hence high justifiability, should occur earlier in the ordering than those generalizations with many degrees of freedom, and hence low justifiability. This ordering is closely related to syntactic complexity orderings. In fact, if the degrees of freedom of a class of generalization are directly proportional to the number of parameters in an expression (which may not be the case in"fuzzy" non-mathematical domains), the justifiability ordering will be equivalent to a syntactic ordering based on the number of terms in an expression. However, the motivation is not based on syntactic complexity but on the difficulty of constructing and justifying generalizations.

7.3.2 A Search strategy

To use this ordering for constructing a generalization of a set of data, we must define a search strategy and also elucidate the different sources of justification that the strategy will use.

The basic search strategy using this ordering is to consider each level of justifiability in turn, starting with the highest justifiability. The search prunes the generalizations that are not compatible with the data (or, alternately, generates the ones that are compatible). If there is a unique generalization consistent with, and justified by the data, the search halts with this generalization. The search will halt without an answer if there are many possible generalizations at the current level because there is no justification for choosing one over the other. If there are no generalizations consistent with the data in that level, the search will proceed to the next level.

If the justification from the data and other sources can be quantified and compared to a measure of the justifiability of the next level, it may be possible to halt the search before proceeding to the next level when there is insufficient justification to justify any theory of the next level. A less efficient method is to search the next level and discover that there

is insufficient justification.

7.3.3 Sources of Justification

There are several sources of justification which are either necessary for the basic search strategy, or which modify the strategy:

- **The data**: The examples, positive and negative, which the generalization must account for.
- **The context**: The occasion that generated the set of data.
- **Previous Theories**: For incremental acquisition, the theory that explained the data before the last bit of data was added.

There may also be additional constraints on the space of generalizations derived from other sources of information about the particular generalization task:

- **Perspective**: Assumptions about what is relevant or important.
- **Purpose**: What the generalization will be used for.

Data

The data is the primary justification for adopting a given theory. A prerequisite for even considering a theory or generalization is that the theory or generalization must be consistent with the data. If the data is just positive examples, the theory must cover all the examples. If the theory includes both positive and negative examples, the theory must also exclude the negative examples.

The amount of data also provides an important justification, in conjunction with the tailorability of the theory: a set of data can only justify theories from a class if the amount of data can completely constrain the values of the parameters of the class. For some domains, this can be quantified very exactly as a relation between the number of parameters of the theory and the number of data points.

For example, in fitting polynomials to sets of points on the plane, the data over-constrains the theory when the number of data points is greater than the number of coefficients of the polynomial (the parameters of the theory), and just constrains the theory when the number of data points equals the number of coefficients. In this borderline case (for example, fitting a quadratic to three points), the theory is only partially justified because the theory could have been tailored to account for any values of the data points

by adjusting adjusting the coefficients.

For domains other than this very precise, mathematical domain, the amount of constraint that the data places on the theory will not be as clear cut. Even if we just extend the set of possible curves in the curve fitting problem to include all curves expressible as differentiable functions of x, one would now need many more than three data points to justify a quadratic curve over a sine wave, hyperbola, *etc.*. Extending the representation language to include these other curves has extended the set of terms in the language to include not only the constant coefficients of a polynomial, but also the operators such as addition, and raising to a power. The quadratic is now a member of a class with more parameters, and therefore more data points are required to justify it.

In common sense domains, such as describing groups of household objects, the representation language may be very large and the boundaries on the classes of theories are less precise than in a formal, mathematical domain. The borderline at which a set of data just constrains the theory is therefore more fuzzy. It is not well defined how large the class containing the theory is, and therefore it is not well defined how much the theory could be tailored to account for different sets of data.

Nevertheless, one can still identify cases where the data clearly does or does not constrain the theory. For example, in most situations, 25 household objects of different functions, sizes and materials that are all bright purple would clearly provide sufficient constraint to justify the theory that the group consists of bright purple objects. The parameters of this theory (which cannot be much more than the property being considered—the colour—and the value of the property—bright purple) could not have been tailored to account for any set of 25 objects.

The borderline cases are those in which there is enough data to determine a unique choice of values for the parameters of the theory, but the theory could have been tailored to account for any variations in the data. There is therefore a question whether the theory captures a real regularity of the data or whether the data is just a random set and the class of theories was large enough to cover any random set, regardless of whether there was a real regularity or not. In the borderline case, the generalizer must obtain additional justification from other sources in order to justify a particular theory.

Context

The set of data does not arise in a vacuum—there must have been some context that

generated the set of data. One context is that of a teacher presenting a set of examples to the generalizer. In this case (assuming the teacher is not trying to confuse the generalizer), the teacher guarantees that the examples have something in common and therefore there is a generalization of the examples to be found. This is justification for believing that there is a genuine generalization of the data. A very different context arises in matching tasks, for example, matching the current situation against many remembered situations in a large database. In trying to match the current situation to a remembered situation, the generalizer looks for a partial match, *i.e.*, a generalization of the two situations. The context then gives no guarantee that the situations will match—no justification for believing that there is a genuine generalization to be found.

The context may also provide partial justification, between the guarantee provided by a teacher and the uncertainty of the database task. For example, in matching two structured descriptions, there is more justification for considering a generalization of two nodes if all the nodes to which they are connected match well (suggesting that the two nodes occupy the same role in the description) than if none of the nodes to which they are connected match at all.

Justification from the context is particularly important for the cases in which the data itself provides only borderline justification for a theory. If the context gives no additional justification, the generalizer must assume that the theory is a spurious result of a tailorable class of theories. If, on the other hand, the context does give justification for believing that a genuine generalization exists, the generalizer will have sufficient justification for guessing that the theory it has found may represent a genuine regularity to the data.

For example, if the teacher guarantees that there is a polynomial that fits a set of 3 points on the plane, the generalizer would be justified in choosing the parabola that fits them. Of course, if there were a straight line through the points, it would have even more justification for choosing that because the three points overconstrain the line.

Perspective

A second, very significant, type of information that the generalizer may use to justify its choice of a generalization is the perspective from which the generalizer should view the data. The context, as used above, is the local setting of the data. The perspective is provided by the more global setting of the whole task and domain in which the data arises. The perspective selects certain properties, terms, or relations in the representation language as relevant or important. This may have two effects on the generalization. By

declaring certain properties or terms as irrelevant, the perspective will eliminate many possible generalizations from consideration. This will simplify the generalization task by reducing the search space, in the same way as exploiting constraints from the domain.

The second effect is to create an additional ordering criterion on the space of generalizations according to the relevance or importance of the terms. properties, and relations of the generalizations.

For example, from the perspective of putting away the items from a grocery bag, a can of baked beans and a can of caviar are both instances of the general class of canned items, and are quite different from a wedge of brie cheese. However, from the perspective of preparing a meal, the property of being in a can or needing refrigeration is much less important than the type of food, so that the brie and caviar now match best as instances of the class of gourmet foods.

The generalizer can use this additional ordering to decide on generalizations that could not be justified without it. In particular, if there are many generalizations of equal justifiability that account for the data, the data alone could not justify a choice between them. However, the relevance ordering imposed by the perspective may distinguish one of the generalizations as the most relevant generalization, and the generalizer would then by justified in choosing it.

Purpose

One other kind of information that the generalizer may use is the purpose or function of the generalization or theory. This may also place an ordering on the generalizations. For example, NODDY finds generalizations of sets of patterns whose function is to be the branch conditions of the procedure. It is important that the branch conditions at a fork be disjoint so that the choice of which branch to take is deterministic. It is also safer for the procedure to not specify what the robot should do in unknown circumstances than to specify the wrong thing to do. This consideration places a preference on more specific conditions. Therefore, when the set of patterns will justify many possible conditions, the generality ordering will justify the choice of the most specific of those conditions, if there is just one such condition. This is the basis of the algorithm of NODDY's condition matcher which finds the most specific instance of the simplest template, if there is just one such template.

Many common sense tasks share this property of placing a preference on the most

specific condition. A monitoring task, such as looking for a particular restaurant while driving down the street, on the other hand, may place a preference on the most general condition if it is safer to have many false alarms than to miss a real alarm.

7.3.4 Summary

In summary, any system that generalized from examples based on the principles above would:

- Define languages to represent the examples and the generalizations.
- Exploit any constraints from the domain to restrict the space of valid generalizations.
- Further restrict the space, if possible, according to constraints from the perspective of the current situation (in some domains, this might change according to the situation).
- Determine what terms/properties/relations of the representation language can be viewed as parameters and order the generalizations into levels of justifiability according to their number of parameters.
- Order the generalizations within each level according to any bias from the purpose of the generalizations (*e.g.*, according to generality) or from the perspective (*e.g.*, according to relevance or importance).
- Given a set of examples, search each level of the ordered space for generalizations that are consistent with the examples until a level with consistent generalizations is found.
- If there is a single best generalization (using the additional ordering from the purpose or perspective if appropriate), choose this generalization.
- If there is not a single best generalization, abandon the search and wait for more data (possibly remembering the set of generalizations to guide the search when more data is presented.

Chapter 8
Conclusion

8.1 Summary

This thesis addressed a task and an issue. The task was to acquire procedures from examples, and the issue was how to constrain generalization in rich domains. The thesis has described an implemented system called NODDY that is able to acquire procedures from examples.

One significant feature of the procedure acquisition task is that the procedures to be acquired have several components that are not present in the the examples: loops; forks and branching conditions; and generalized actions involving dependencies on other components of the procedure. NODDY is able to infer procedures with all these components from examples which are linear traces of an execution of the procedure and have no loops, forks, conditions or dependencies. Inferring each of these components from the traces involves a different kind of generalization.

NODDY employs **three generalization methods** to infer procedures from examples:

- NODDY infers loops and forks by matching a procedure to a new trace and allowing multiple pairings of the events of the procedure and trace. It finds the appropriate match of the procedure and trace events by identifying a skeleton of *key pairs*—pairs of events with unique positions in the procedure and trace—and propagating through the structure of the descriptions from the skeleton to find new pairs.
- NODDY finds generalized actions involving dependencies on previous patterns by searching a space of expressions built out of operators and constants. It uses information about the inverses and the types of the range and arguments of an operator to prune the search to manageable size and to find expressions involving constants without having to search though all possible values of the constants.
- NODDY generalizes conditions (described by a condition type and a list of parameter values) by climbing a hierarchy of templates—parameterized descriptions of families of generalized conditions. The ordering on the templates is determined by their justifiability. This search method enables an efficient search of a very large space of conditions, and also provides a way of halting the search when there is insufficient

evidence to justify any generalization.

Another significant feature of the procedure acquisition task is that a procedure must be acquired from positive examples only. NODDY is able to acquire procedures under these circumstances because it exploits the **determinacy constraint** on procedures in two ways:

- The constraint limits the search space of generalized actions to a manageable size.
- The constraint enables NODDY to construct negative examples for acquiring conditions by finding groups of conditions that do not satisfy the determinacy constraint and using elements of the group as negative examples for each other.

The key new idea arising out of NODDY is the principle of **justified generalization**: one should only choose a generalization of a set of data when there is sufficient justification from the data and the context of the data to determine the choice uniquely and untailorably—when there is only one best generalization and it could not have been tailored to fit an arbitrary set of data.

Generalizations with many parameters are more tailorable than generalizations with few parameters. Therefore, generalizations with many parameters require more justification from the data. The appropriate ordering on the search space is therefore to consider the generalizations with few tailorable parameters before those with many parameters.

The principle also provides a halting criterion by which a generalizer can determine when to abandon a search for a generalization: search the space according to the justifiability order until there is insufficient justification from the data and the context to justify generalizations from later in the ordering. The justification for a choice is primarily from the data—the positive and negative examples—but the context generating the data, the perspective determined by the task and the domain, and the purpose to which the generalization will be put, may all provide additional justification for searching further in the search space or for choosing one generalization over another.

8.2 Future Directions

8.2.1 Justification

One important direction for further research is to explore the application of the justified generalization principle to other domains. In particular, it is important to consider non-mathematical domains in which it is not as clear how one can identify the tailorable

parameters of a description, since this is essential for determining where a generalization should be placed in the justifiability ordering.

A related issue is determining the appropriate use of the context of a set of data to provide additional justification for a generalization. In the procedure domain, the context for generalizing a set of events consisted of the surrounding events that had already been matched and generalized. This type of context would apply to any generalization task that involved matching structured descriptions. However, it is not as clear how this context should be used. NODDY used a simple criterion for determining what context would justify what kind of generalization. The criterion relied on the fact that there is only one kind of link between the events in a procedure. In other domains where the descriptions may have many kinds of relations, the criteria for evaluating a context will be more complicated.

8.2.2 Extensions to NODDY

Another area for future work is to extend the current version of NODDY in various directions. Several possibilities are:

- Enabling NODDY to construct its own template hierarchies.
- Extending the loop induction method to find counted loops as well as DO UNTIL loops.
- Applying NODDY to procedures in other domains, both robot domains (for example, procedures for robots with visual input), and non-robot domains (for example, procedures for common sense tasks such as washing the dishes).

Constructing Template Hierarchies

Constructing a new template for the action or condition hierarchy involves two distinct problems. The first problem is to construct the defining expression of the template—a function from patterns to action parameters for an action template, and a predicate on patterns for a condition template. The second problem is to construct the predicates and functions associated with the template that determine the relation of instances of the new template to instances of the same and other templates.

Constructing the defining expressions involves making explicit the generalization methods for generalizing actions and conditions, using these methods to construct generalized actions or conditions, then parameterizing these generalizations to obtain descriptions of families of generalizations.

For the action templates, the functional generalization method is already explicit in expression building algorithm of the second level action generalization. NODDY could create new action templates simply by parameterizing any local functions it found in the second level action generalization. Creating action templates for guard generalization and condition templates could be done with a modified form of the expression building algorithm which would build expressions representing predicates on patterns.

The second problem of deriving the other predicates and functions associated with the template is more difficult because it involves reasoning about the defining expressions. The particular kind of reasoning would vary according to the domain. For the robot domain, it would involve algebraic manipulation and reasoning about inequalities.

One issue to address in augmenting the hierarchies is when a template should be added. When the hierarchies are small, they represent a bias in favour of certain "common" or "expected" generalizations which can be found easily and require less justification than the generalizations not in the hierarchy. Indiscriminately adding templates would remove this advantage. It would probably be appropriate to add a new template to the hierarchy only when instances of the template had been found to be useful in several different procedures.

Guard Generalization

NODDY has not addressed the issue of guard generalization significantly. In a domain with a rich representation for guarded actions (*e.g.*, [Lozano-Perez *et. al.* 1984]) it is possible to express all or most of the control structure of a procedure in the guards, eliminating the need for forks and and most loops. Such procedures are essentially plans described by a sequence of subgoals to achieve. The guard of each action is the subgoal, and the executor will apply the specified action until the goal is reached.

Acquiring this kind of procedure from examples will require more sophisticated reasoning than NODDY has. In particular, it involves inferring the goal or purpose that the teacher intended for a particular action. This will almost certainly require some knowledge of the domain and the types of tasks that are possible in the domain. Combining the syntactic generalization methods of NODDY with more semantic generalization methods appropriate for inferring goals and subgoals from examples is a very interesting direction for future research.

Loops

In the current version of NODDY, DO UNTIL loops "fall out" of the structure matching algorithm. However, this method of finding loops is dependent on carefully chosen examples to identify the boundaries of the loop. The problem is that NODDY does not look for groups explicitly. Future work on this problem should develop more powerful loop generalization methods that deal with loops explicitly to overcome the limitations of the current algorithm.

A more difficult problem is finding counted loops--loops in which the exit condition is the number of times around the loop rather than a condition on the input pattern. The major problems are deciding that a loop may be a counted loop and determining what the count depends on if it is a counted loop. This is an instance of the "invisible objects" problem, because the count is not present in the examples, so that one must guess at both its existence and its properties. One may be able to exploit the determinacy constraint again to solve this problem: a loop is a candidate for being a counted loop when NODDY is unable to construct a deterministic fork to exit the loop (the exit condition does not appear to depend on anything in the current pattern). NODDY would then search for some other part of the procedure on which the count could depend. The way in which it could do this is an open question.

Interactive Procedure Acquisition

The current version of NODDY is not an interactive learner in that it does not interact significantly with the teacher and not at all with the world. The acquisition of procedures is a particularly good task for developing an interactive learner, because the learner can interact very easily with the world simply by executing the procedures it has inferred and observing the effects. This interaction could reduce the burden on the teacher of generating all the examples, and would introduce other kinds of learning tasks, such as learning by discovery and learning by analysis.

There are several ways in which NODDY could be turned into an interactive learner. The following scenarios suggest some of them.

- Scenario 1: Partial Trace Generation.

The teacher leads NODDY through a task once or twice for NODDY to acquire a basic procedure, as it does at present. The teacher then allows NODDY to execute this procedure in several different situations. When NODDY's current procedure does not

handle some situation—the pattern it receives does not match the conditions of any of the succeeding events—NODDY either guesses the event with the "'closest" condition, or simply waits for the teacher to take over. If NODDY makes a mistake—performs an action when it should not have, the teacher intervenes, backs up and leads NODDY through the task until it is back at a point that matches NODDY's procedure.

NODDY would not have to be extended significantly to do this, since all the generalization is identical, but the teacher would only have to lead NODDY through the new parts of the traces rather than having to generate a complete trace, even when most of it was identical to previous traces.

- Scenario 2: Ascribing Blame.

After the teacher has shown NODDY the first few examples, it attempts the task in different situations. When the procedure does not match the situation, it either guesses the next move or asks the teacher for help. If NODDY has made a wrong move, the teacher tells it that it has made a mistake. If NODDY just does not know what to do next, the teacher leads it through enough action to get it back to a part of the procedure that it knows. If NODDY completes the procedure, the teacher tells NODDY whether it did it right or not. If it was wrong, NODDY uses the trace it has just generated as a negative example of the procedure, to specialize the procedure so that it will not make the same mistake again.

Using negative examples involves the very difficult problem of assigning blame—the negative trace matches the procedure perfectly (because that is how the trace was generated) and NODDY must determine which of the events in the procedure were wrong or too general and should be modified so they do not match the trace. Assigning blame is an open research question.

- Scenario 3: Goal Based Reasoning.

The most extensive extension of NODDY would be to introduce goal based reasoning into NODDY. The teacher would again present one or two traces of the procedure. NODDY would then analyze the examples to determine what the *goal* of the procedure was. It would then generalize the procedure by reasoning backwards through the procedure from the goal. This would involve all the components of the instructable robot described in section 1.4.

References

Anderson, J. R. (ed), [1981], *Cognitive Skills and Their Acquisition*, Erlbaum: Hillsdale, N.J..

Andreae, J. H., [1981], *Thinking With the Teachable Machine*, Academic Press.

Andreae, J. H., [1972–84], *Man-Machine Studies, Progress Reports*, Dept. Elec. Eng., U. Canterbury, New Zealand.

Andreae, P. M., and Andreae, J. H.,, [1979], *A Teachable Machine in the Real World*, Int. J. Man Machine Studies.

Angluin, D., [1980], *Inductive Inference of Formal Languages from Positive Data*, Information and Control, **45** 2.

Angluin, D. and C. H. Smith, [1982], *A Brief Survey of Inductive Inference*, TR 250, Dept. Comp. Sci., Yale University.

Berwick, R., [1982], *Locality Principles and the Acquisition of syntactic Knowledge*, PhD Thesis, M.I.T..

Bierman, A. W. and R. Krishnaswamy, [1976], *Constructing Programs from Example Compuations*, IEEE Trans Software Eng, **SE-2**, 3, pp 141–153.

Bierman, A. W., Baum, R. I. and Petry, F. E., [1975], *Speeding up the Synthesis of Programs from Traces*, IEEE Trans. Comput. **C-24** pp 122–136.

Biermann, A. W., [1978], *The Inference of Regular LISP Programs from Examples*, IEEE Trans on Systems, Man and Cybernetics, **SMC-8**, 8, pp 585–600.

Burstall, R. M., and J. Darlington, [1977], *A Transformation System for Developing Recursive Programs*, J. ACM, **24**, 1, pp 44–67.

Dietterich, T. G., [1982], , "Learning and Inductive Inference", chapter 14 of *Handbook of Artificial Intelligence*, Volume 3, William Kaufman, Los Altos, CA.

Dietterich, T. G. and Michalski, R. S., [1981], *Inductive learning of Structural Descriptons: Evaluation criteria and comparative review of selected methods.*, Artificial Intelligence, **16**, 257–294.

Dufay, B. and Latombe, J-C., [1983], *An Approach to Automatic Robot Programming Based on Inductive Learning.*, Robotics Workshop, M.I.T..

Dufay, Bruno, [1983], *Apprentissage par induction en robotique. Application a la Synthese de Programmes de Montage.*, Thesis, l'Institue National Polytechnique de Grenoble.

Fikes, R. E., Hart, P. E., and Nilsson, N. J., [1972], *Learning and Exectuing Generalized Robot Plans.*, Artificial Intelligence, 3, pp 251–288.

8, pp337–365;

Gentner, D., [1983], *Structure-Mapping: A Theoretical Framework for Analogy*, Cognitive Science, 7, 2.

Gold, E. M., [1967], *Language Identification in the Limit*, Information and Control. 10 pp447–474.

Goodman, Nelson, [1973], *Fact, Fiction, and Forecast*, Bobs-Merrill, Indianapolis, 3rd Edition.

Grossman, D. D., [1977], *Programming a Computer Controlled Manipulator by Guiding Through the Motions*, IBM Research Report, RC6393.

Grossman, D. D. and Summers, P. D., [1984], *XPROBE: An experimental System for Programming*

Robots By Example, Int. J. Robot Research, 3 1.

Hayes, P., [1979], , "The Naive Physics Manifesto", in *Expert Systems in the Micro-Electronics Age,* edited by D. Michie, Edinburgh University Press.

Hayes-Roth, B. and Hayes-Roth, F., [1977], *Concept learning and the recognition and classification of exemplars,* J. of Verbal Learning and Verbal Behaviour, **16** pp321–338.

Hayes-Roth F. and McDermott, J., [1976], *Learning Structured patterns from examples.,* Proc. of the third Int. Joint Conf. on Pattern Recognition, Stanford, CA, 419–423.

Hayes-Roth F. and McDermott, J., [1978], *An interferance matching techniques for inducing abstraction,* Com. of ACM, **21**, 401–411.

Iba, G. A., [1979], *Learning disjunctive concepts from examples,* AI Memo 548, M.I.T..

Langley, P., [1981a], *Data-Driven Discovery of Physical Laws,* Cognitive Science, **5**, pp31–54.

Langley, P., [1981b], *BACON.5: The Discovery of Conservation Laws,* Proc. 7th IJCAI..

Langley, P., Bradshaw, G. L., and Simon, H. A., [1983], , "Rediscovering Chemsitry with the BACON System", in *Machine Learning,* Michalski, Carbonell and Mitchell (eds), Tioga Press, Palo Alto CA.

Latombe, J-C., [1981], *Etat d'avancement des recherches au 1er Octobre 1981,* Rapport de Recherche IMAG, 291, Grenoble, France.

Lenat, D. B., [1982a], , "AM: Discovery in Mathematics as Heuristic Search" in *Knowledge-based Systems in Artificial Intelligence* R. Davis and D. B. Lenat (eds), McGraw-Hill, New York. Based on 1977 PhD thesis, Standord University.

Lenat, D. B., [1982b], *The Nature of Heuristics,* Artificial Intelligence, Cambridge, Mass.

Lieberman, Henry, [1980], *A Session With Tinker: Interleaving Program Testing With Program Design,* Memo 577 Artificial Intelligence Lab, M.I.T., Cambridge, MA.

Lozano-Perez, T., [1980], *Automatic Planning of Manipulator Transfer Movements,* Memo 606, Artificial Intelligence Lab, M.I.T. Cambridge, MA.

Lozano-Perez, T., Mason, M. T., Taylor, R. H., [1984], *Automatci Synthesis of Fine-Motion Strategies for Robots,* Int. J. Robotics Research, **3**, 1, pp3–24.

Manna,Z., and R. Waldinger, [1979], *Synthesis: Dreams to Programs,* Memo AIM-302, Stanford University Artificial Intelligence Lab, CA.

Manna, Z., and R. Waldinger, [1979], *Synthesis: Dreams to Programs,* Shorter version IEEE Trans SE, SE-5, 4, pp 294–328.

Marcus, M., [1980], *A Theory of Syntactic Recognition for Natural Language,* M.I.T. Press, Cambridge, MA.

Mason, M., [1979], *Compliance and Force Control for Computer Controlled Manipulators,* Technical Report 515, Artificial Intelligence Lab., M.I.T., Cambridge, MA.

McLaughlin, J. R., [1982], *TRIG: An Interactive Robotic Teach System,* Working Paper 234, MIT AI Lab.

Michalski, R. S., [1980], *Knowledge Acquisition Through Conceptual Clustering: A Theoretical Framework and an Algorithm for Partitioning Data into Conjunctive Concepts,* Int. J. of Policy Analysis and Information Systems, **4**, 3.

Michalski, R. S., [1983], , "A Theory and Methodology of Inductive Learning" and "Learning

from Obseration", Chapters 4 and 11 of *Machine Learning* (eds) Michalski, Carbonell, Mitchell. Tioga Pub. Co. Palo Alto, California.

Minsky, M. L., and Papert, S., [1969], *Perceptrons; An Introduction to Computational Geometry*, MIT Press, Cambridge, Mass..

Mitchell, T. M., [1982], *Generalization as Search*, Artificial Intelligence, 18, 203–226.

Mitchell, T. M., [1983], *Learning and Problem Solving*, IJCAI 83, Computers and Thought Lecture.

Rich, C., [1981], *A Formal Representation for Plans in the Programmer's Apprentice*, Proc. IJCAI 7, pp 1044–1052.

Sacerdoti, E. D., [1974], *Planning in a Hierarchy of Abstraction Spaces*, Artificial Intelligence **5**, pp115–135.

Samuel, A. L., [1959], *Some Studies in Machine Learning using the Game of Checkers*, IBM J./ Research and Development, 3 pp210–229.

Shapiro, E. Y., [1982], *Algorithmic Program Diagnosis*, J. ACM.

Summers, P. D., [1977], *A Methodology for LISP Program Construction from Examples*, J. ACM, **24**, pp 161–175.

Sussman, G. J., [1975], *A Computer Model of Skill Acquisition*, Elsevier.

Van Lehn, Kurt, [1983], *Felicity Condition for Human Skill Acquisition: Validating an AI-based Theory*, Technical Report CIS-21, Xerox PARC, Palo Alto, CA.

Winston, P. H., [1970], *Learning Structural Descriptions From Examples*, PhD Thesis, M.I.T.

Winston, P. H., [1980], *Learning by Understanding Analogies*, AI Memo 520, M.I.T..

This blank page was inserted to preserve pagination.

CS-TR Scanning Project
Document Control Form

Date : 1/25/96

Report # AI-TR-834

Each of the following should be identified by a checkmark:
Originating Department:

☒ Artificial Intellegence Laboratory (AI)
☐ Laboratory for Computer Science (LCS)

Document Type:

☒ Technical Report (TR) ☐ Technical Memo (TM)
☐ Other:_____

Document Information

Number of pages: 157 (164-IMAGES)
Not to include DOD forms, printer intstructions, etc... original pages only.

Originals are:
☒ Single-sided or
☐ Double-sided

Intended to be printed as :
☐ Single-sided or
☒ Double-sided

Print type:
☐ Typewriter ☐ Offset Press ☒ Laser Print
☐ InkJet Printer ☐ Unknown ☐ Other:_____

Check each if included with document:

☒ DOD Form (2) ☐ Funding Agent Form ☒ Cover Page
☐ Spine ☐ Printers Notes ☐ Photo negatives
☐ Other: _____

Page Data:

Blank Pages (by page number): _____

Photographs/Tonal Material (by page number): _____

Other (note description/page number):

Description :	Page Number:
(A) IMAGE MAP; (1-157) / JUNK'ED BLANK	2-156
(158-164) SCANCONTROL, COVER, 000(2), TRGTS (3)	
(B) CUT&PASTE FIGS ON PAGES	2, 12-15, 17-26, 44-45, 47, 51, 58-60, 63-67, 69-72, 74, 76, 78, 81-82, 84-85, 87-90, 92, 97, 102, 106, 109, 115, 120, 128

Scanning Agent Signoff:

Date Received: 1/25/96 Date Scanned: 1/30/96 Date Returned: 2/1/96

Scanning Agent Signature: Michael W. Cooper

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE *(When Data Entered)*

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER AI-TR-834	2. GOVT ACCESSION NO.	3. RECIPIENT'S CATALOG NUMBER AD-A156408
4. TITLE *(and Subtitle)* Justified Generalization: Acquiring Procedures From Examples		5. TYPE OF REPORT & PERIOD COVERED Technical Report 6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s) Peter Merrett Andreae		8. CONTRACT OR GRANT NUMBER(s) N00014-80-C-0505
9. PERFORMING ORGANIZATION NAME AND ADDRESS Artificial Intelligence Laboratory 545 Technology Square Cambridge, Massachusetts 02139		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS
11. CONTROLLING OFFICE NAME AND ADDRESS Advanced Research Projects Agency 1400 Wilson Blvd Arlington, Virginia 22209		12. REPORT DATE January 1985 13. NUMBER OF PAGES 161
14. MONITORING AGENCY NAME & ADDRESS*(if different from Controlling Office)* Office of Naval Research Information Systems Arlington, Virginia 22217		15. SECURITY CLASS. *(of this report)* UNCLASSIFIED 15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT *(of this Report)*

Distribution of this document is unlimited.

17. DISTRIBUTION STATEMENT *(of the abstract entered in Block 20, if different from Report)*

18. SUPPLEMENTARY NOTES

None

19. KEY WORDS *(Continue on reverse side if necessary and identify by block number)*

machine learning
constraining generalization
justification of generalization

20. ABSTRACT *(Continue on reverse side if necessary and identify by block number)*

This thesis describes an implemented system called NODDY for acquiring procedures from examples presented by a teacher. Acquiring procedures from examples involves several different generalization tasks. Generalization is an underconstrained task, and the main issue of machine learning is how to deal with this underconstraint. The thesis presents two principles for constraining generalization on which NODDY is based. The first principle is to exploit domain based constraints. NODDY demonstrates how such constraints

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE
S/N 0102-014-6601

UNCLASSIFIED (cont. over)

SECURITY CLASSIFICATION OF THIS PAGE *(When Data Entered)*

Block 20 continued:

can be used both to reduce the space of possible generalizations to manageable size, and how to generate negative examples out of positive examples to further constrain the generalization. The second principle is to avoid spurious generalizations by requiring justification before adopting a generalization. NODDY demonstrates several different ways of justifying a generalization and proposes a way of ordering and searching a space of candidate generalizations based on how much evidence would be required to justify each generalization.

Acquiring procedures also involves three types of constructive generalization: inferring loops (a kind of group), inferring complex relations and state variables, and inferring predicates. NODDY demonstrates three constructive generalization methods for these kinds of generalization.

Scanning Agent Identification Target

Scanning of this document was supported in part by the **Corporation for National Research Initiatives,** using funds from the **Advanced Research Projects Agency** of the **United states Government** under Grant: **MDA972-92-J1029.**

The scanning agent for this project was the **Document Services** department of the **M.I.T Libraries.** Technical support for this project was also provided by the **M.I.T. Laboratory for Computer Sciences.**

Scanned

Date: 1/31/1996

M.I.T. Libraries
Document Services