CIFE CENTER FOR INTEGRATED FACILITY ENGINEERING

# How WATCH Works: Documentation for the WATCH, NEWWATCH, and MetaWATCH Programs

Andrew Gans, Tony Confrey
and
Barbara Hayes-Roth

**TECHNICAL REPORT**
**Number 27**

April 1990

If you would like to contact the authors please write to:

*c/o CIFE, Civil Engineering,*
*Stanford University,*
*Terman Engineering Center*
*Mail Code: 4020*
*Stanford, CA 95305-4020*

# Contents

# 1. Introduction

## 1.1 Overview

The WATCH system derives its name from the way it "watches over the shoulder of an expert" who is trying to solve a problem. WATCH observes a sequence of actions performed by the expert, to inductively learn the intended control strategy. The actions, which solve a problem for some BB1 application, must be expressed in ACCORD or a similar BB1 language framework.

## 1.2 The programs

The WATCH system consists of three programs. The original WATCH program was designed by Jeff Harvey. It tries to generalize the action sequence into a single hierarchical explanation, based on the assumption that the expert had only a single strategy in mind in selecting all of the actions. The NEWWATCH extension was written by Andrew Gans to learn from action sequences with interruptions. In other words, parallel or opportunistic strategies that concurrently selected some of the actions can also be discovered. MetaWATCH, the practicum of Tony Confrey, is designed to learn from multiple problem-solving sessions. A number of different action sequences are examined, to discover the strategy they all have in common.

WATCH, which learns strategies for BB1 applications, is itself written within the BB1 framework. NEWWATCH is a revised version of this original BB1 application, with the added ability to discover interruptions. MetaWATCH is a separate application program, which takes the strategies learned by WATCH and NEWWATCH as its input.

## 1.3 Uses of the WATCH system

The WATCH system was designed to be a knowledge acquisition tool. It allows an expert to define all of the background domain knowledge for a BB1 application, and then automatically learn control information by observing the expert's problem-solving process. The expert merely takes the place of the

control strategy by selecting the actions to perform one at a time. While it may have been difficult to articulate a set of general rules about the control strategy, demonstrating this control by selecting actions can be an easier task. The expert is saved the need to explicitly define control knowledge for any problems solved in the same way as the learned cases.

WATCH can also learn if the actions were chosen by an existing control strategy. This may be the case if WATCH is being tested, to determine how much of the original strategy can be rediscovered merely by observing the actions performed. In this case, the strategies learned by the WATCH system can be compared to the real existing control; this gold standard is the best strategy that WATCH could possibly learn.

# 2. WATCH

## 2.1 Running WATCH

Although WATCH is conceptualized as a system that "watches" actions as they are chosen by the expert, for practical reasons it is normally run in batch mode. In other words, the expert's problem-solving behavior, in the form of a run of the application system, is first stored. WATCH is then executed using this saved information.

Therefore, the WATCH program consists of two sub-systems:

1. The first part of WATCH is an extension to BB1 Version 2. It allows the user to save a trace of the problem-solving at the completion of a run of BB1 on an application.
2. The main part of WATCH is itself is a BB1 application. It learns control strategies using the stored trace.

### 2.1.1 Running and saving the application

WATCH learns a strategy by observing a sequence of actions that solve a problem for some application. Therefore, the first step of WATCH's learning is obtaining a trace of this action sequence. This is done by running the application program within a modified version of BB1 2.1. This special BB1 system will allow all the necessary information for WATCH to be stored on two files, when the application run has completed.

The special version of BB1 2.1 for WATCH can be created by first normally loading BB1. All of the necessary modifications are stored in the file MYWATCHFNS, in the WATCH directory. This file, along with the auxiliary translation functions in IL-TO-CL, should be loaded before the completion of the application run in BB1. The revised BB1 will provide a new "LEARN" option will appear in the BB1 Run menu.

The "LEARN" option contains the menu items "Turn WATCH on" and "Turn WATCH off". These can be selected to toggle whether or not the application problem-solving session should be saved at the end of the BB1 run.

To prepare an application problem-solving session for use by WATCH, the application should be run normally; the user need only set the toggle on before the run is completed. To insure that WATCH obtains a complete trace of the actions, BB1's global variable *BB1-INFO-RETENTION-TIME* should be set to NIL, so that none of the necessary information is deleted.

If the "Turn WATCH on" option has been chosen, a trace of the problem-solving will be saved at the end of the application's run in BB1. Saving a session involves creating two files, one for the sequence of actions and another for the domain-specific background knowledge. The user will be prompted to choose names for the files. The data file, with a name like WATCHSESSION7.DATA, contains the sequence of actions that the expert or control strategy chose to be executed. The knowledge base file, such as WATCH-PROTEANSAVE-KB.LISP, contains all of the KBs in the application system, except for the CONTROL-DATA and CONTROL-PLAN blackboards. This saved knowledge is the necessary background information for the learning process.

### 2.1.2 Running WATCH on the application

WATCH runs like any other application in BB1. All of the necessary files are stored in a standard directory, currently KSL-Exp-25:WATCH and also a backup in SAFE:/gans/cl-watch/new. Loading into BB1 the WATCH-SETUP file from either of these directories will load into BB1 all the other necessary files. The files which make up the WATCH system are:

WATCH-CONTROLFNS
WATCH-DOMAINFNS
MYWATCHFNS
: functions that save the trace of the application's action sequence

WATCH-WRITEOUT
: functions that save the new control strategy learned by WATCH

WATCH-GANS
: functions belonging to the NEWWATCH extension

IL-TO-CL
: macros that translate parts of the WATCH code, written originally in INTERLISP, into the current COMMONLISP

WATCH-KS-KB

knowledge sources of WATCH

WATCH-LEARN-KB

blackboard on which WATCH stores its work

WATCH-NEWGENERIC-KB

versions of the GENERIC-CONTROL KSes modified specially for WATCH

[WATCH-EXTRACODE

functions written for the original WATCH by Jeff Harvey that are no longer used]

When loading WATCH, BB1 will display a message saying that it is unable to find a file named WATCH-WATCHSAVE-KB. There is no such file in the system; this is merely a dummy placeholder name. The user should substitute the name of the KB file saved at the conclusion of the application run (WATCH-PROTEANSAVE-KB in the above example). Once WATCH is loaded into BB1; running it will cause a prompt for the name of the other saved file, the data file containing the list of actions (in this case WATCHSESSION7). The only files that need to be loaded manually when running WATCH, if they are not already in memory, are the CONTROLFNS and DOMAINFNS files belonging to the application system.

It is important to note that the most recent version of the WATCH program is actually the newer NEWWATCH application. Therefore, instead of loading WATCH itself, the user instead should obtain the latest version of NEWWATCH and make the following modifications. After the NEWWATCH program has been loaded, manually remove the two Knowledge Sources (KSes) that are specific to NEWWATCH. These are called POSTULATE-SEQUENCE-WITHOUT-INTERRUPTION and HYPOTHESIZE-INTERRUPTION. These can be removed simply by changing their TRIGGER-CONDITION slots to NIL, either by editing in BBEDIT after NEWWATCH is loaded into BB1 or by manually editing the WATCH-KS-KB before it is loaded.

## 2.2 The WATCH program

### 2.2.1 The six phases

WATCH begins by placing a copy of the observed action sequence on the blackboard, to provide the input for the learning task. The knowledge source POST-THE-PROBLEM creates the appropriate object.

Once the action sequence is available, WATCH can perform the six phases of learning a control strategy. These are unfolded one at a time by LEARN-CONTROL-STRATEGY, the knowledge source that causes these tasks to be performed in the appropriate order.

1. Generalize the action sequence.
   Incrementally examine the sequence of actions, and build a tree of possible generalizations of various sub-sequences.
2. Choose possible strategies.
   Identify all possible strategies by finding the sets of sub-sequence generalizations in the tree that explain the whole action sequence.
3. Refine the strategies.
   Simplify strategies by removing unnecessary levels of abstraction.
4. Choose the best strategies.
   Rank the possible strategies using a set of heuristic preferences.
5. Hypothesize modifiers.
   Add appropriate modifiers that explain the order of actions within each sub-sequence of the strategy.
6. Write out control strategies.
   Output the strategies in two formats useable by BB1, as a set of control knowledge sources and as a skeletal plan.

### 2.2.2 The LEARN blackboard

The LEARN blackboard stores the partial solution while WATCH tries to learn the expert's control strategy. Included in this blackboard are a number of objects.

1. LEARN.PROBLEM.PROBLEM1

   This blackboard object is created by the first KS to run,

POST-THE-PROBLEM. It contains a copy of the action sequence from the saved data file.

2. generalization objects at the LEV0-S level

   Each action in the sequence is posted to the blackboard, by KS.DOMAIN.POST-INITIAL-STATEMENTS, after all possible generalization is done for the preceding action. The nth action in the sequence is stored in the object LEARN.LEV0-S.LEV0-Sn.

3. generalization objects at higher levels

   Domain KSes GENERALIZE-STMT-SEQUENCE and POSTULATE-STMT-SEQUENCE do inductive generalization and store their results in objects on higher levels, such as LEARN.LEV3S.LEV3S-13 or LEARN.LEV4-S.LEV4-S3. In this way, a tree of abstractions is built up in the LEARN blackboard.

Generalization objects on the learn blackboard each contain certain standard attributes and links, which show how they were inductively abstracted from object on lower levels. Some of the more important attributes are the ACTION-STMT and PARSED-VAL, which contain, respectively, readable and parsed expressions of the generalized action. GENERALIZED-VARS contains a list of parameter positions, in the parsed version of the action, that had to be abstracted to obtain this generalized action.

Each object has a links to generalizations at lower and higher levels of abstraction that it is GENERALIZED-FROM and GENERALIZED-TO. The generalizations at the same level which explain the immediate preceding and succeeding actions in the sequence are connected by POSSIBLE-LAST-STMT and POSSIBLE-NEXT-STMT links. If WATCH postulates a sequence containing a pattern, which requires a set of generalizations for a full explanation, these are connected by LAST-IN-SEQUENCE and NEXT-IN-SEQUENCE links.

### 2.2.3 The generalization phase

The WATCH control knowledge source GENERALIZE-THE-STATEMENTS invokes the generalization phase of WATCH's learning. Since generalization among the set of problem-solving actions can quickly explode

exponentially into an impossible task, two attributes of the KS limit the work that WATCH will do. The NUMBER-OF-GENERALIZATIONS attribute determines that only the *n* best generalizations will be made for any generalization object on the LEARN blackboard. The value of *n* is normally rather small, such as 1 or 2, to limit the combinatorial explosion. The GENERALIZATION-PARAMETERS attribute, described below, determines over which parameters the search for generalizations will be performed.

Since WATCH only does a small subset of the possible generalizations, it needs to rank the possibilities to choose which ones to perform. There are seven heuristics used to select this order. Some are based on knowledge about the ACCORD language, such as "avoid generalizing actions". Others, such as "prefer more statements" which prefers longer sequences of actions and "prefer generalizing over fewer levels of abstraction", are general to induction over any sequence. It is important to note that these latter two are contradictory, and act to balance out opposing concerns; if we add more actions to a sequence, we will have to generalize more to find one common abstraction. Two of the heuristics only influence the order of generalization, but do not cause any of the possibilities to be ignored due to a lower rating; these are "prefer rewrites" and "prefer lower level generalizations first".

The work of the generalization phase is done by three knowledge sources. POST-INITIAL-STATEMENTS adds each action to the LEARN blackboard. GENERALIZE-STMT-SEQUENCE generalizes actions with higher levels of abstraction, and moves them to higher levels of the generalization tree. POSTULATE-STMT-SEQUENCE hypothesizes that a set of generalizations might be a distinct phase of the strategy.

- POST-INITIAL-STATEMENTS adds a generalization object to the LEARN.LEV0-S blackboard level for each action in the problem-solving sequence. KSARs are triggered for each action as soon as the problem is selected at WATCH's first BB1 step, but they only become executable one at a time. As a result, the new action is only added to the blackboard, and the generalization process, after all the generalization has been done for the previous actions.

- GENERALIZE-STMT-SEQUENCE serves different purposes in the

generalization phase. It does solo generalization, moving objects to higher levels of abstraction so that they can be used in various levels of the hierarchical strategy. In addition, it does generalizations that will form BB1's REFINE-PARAMETERS type strategies (where a single strategic decision object is refined over a set of parameters in sequence to form an entire sub-tree of the control plan).

A KSAR is triggered for the generalization starting points found when SVB-GENERALIZATION-STARTING-POINTS calls CHECK-UPPER-LEVEL-STMT, CHECK-SAME-LEVEL-STMT, CHECK-LOWER-LEVEL-STMT, CHECK-SOLO-GENERALIZATION, and MODIFY-REQUIRING-REGENERALIZATION.

- POSTULATE-STMT-SEQUENCE generalizes over a sub-sequence of the problem-solving actions that have one or more parameters in common. It is triggered by the addition of a new generalization object to the blackboard, and it attempts to find all sequences that include that object.

The function FIND-PARAMETER-SEQUENCE calls GET-PARAMETER-SEQUENCE for each parameter of the parsed action in which we wish to look for sequences. Which parameters are examined to find a match depends on the value of the attribute GENERALIZATION-PARAMETERS.

For example, the new blackboard item could contain a parsed ACCORD ANCHOR action such as

*((ANCHOR) (RANDOMCOIL2) (HELIX1)*
*(PARTIAL-ARRANGEMENT1) (CONSTRAINT-SET-R2H1))*

based on the action template of ANCHOR

*((ACCORD.ACTION.ANCHOR) (OBJECT) TO (ANCHOR)*
*IN (PA) WITH (CONSTRAINT)).*

The various clauses of the GENERALIZATION-PARAMETERS attribute tell us which of the parameters in the parse to examine for a match. The clause (OBJECT TT) would tell WATCH always to attempt to postulate sequences that have OBJECTS in common. In other words, WATCH would look for a sequence of items that matched the second parameter, the OBJECT *RANDOMCOIL2*, of the ANCHOR action.

The clauses of the GENERALIZATION-PARAMETERS attribute can take on other values, such as (PA IF-NONE) or (ACTION TT). WATCH looks for matches within the parameters in the order that these clauses appear. The second item in each clause gives additional instructions about when to attempt this match:

- TT: always attempt to match this parameter
- NIL: don't do any matching on this parameter (this is equivalent to not including a clause for this parameter)
- IF-NONE: match on this parameter only if there are no matches for any other parameter
- HALT-IF-PRESENT: match on this parameter, and stop looking for matches of other parameters if this one is successful

GET-PARAMETER-SEQUENCE determines the position of the parameter in the parse, by examining ACCORD's action templates in GET-PARAMETER-NUM-FROM-ACTION. This position number is then passed to FIND-THE-SEQUENCES, which attempts to match a sequence containing the object. FIND-THE-SEQUENCES:

- collects all the possible sequences of objects at this level of abstraction (COLLECT-SEQUENCE-OF-STMTS)
- examines in reverse order all of these sequences that contain the current object
- finds those objects in the sequences that match the current object by having a matching parameter (PARAMETERS-MATCH?)
- checks whether all the statements in between the matching pair also have the corresponding match with the next objects in the sequence (INTERVENING-STMTS-MATCH)
- extends the sequence by looking back in the sequence for other sub-sequences that match the sub-sequence made up of the matching pair and intervening statements

For example, FIND-THE-SEQUENCES might be trying to find sequences of objects which match some parameter of object x. Object x is at a level of abstraction which is fully covered by sequences of objects (a b c d e x) and (a f g x), meaning that these sequences explain the whole action sequence. The first of these is examined in reverse to find that x and e do not match over the given parameter. Therefore, the match moves through the list in reverse, finding that x and d do succeed the test in PARAMETERS-MATCH?. The intervening object e is matched

with the next item back in the list c, to attempt to find a correspondence between the pairs (e x) and (c d). After the success of this test, the sequence is extended if (a b) also matches (e x), indicating that both ((a b) (c d) (e x)) and ((c d) (e x)) are possible sequences to postulate. FIND-THE-SEQUENCES returns all the possibilities found by fully checking both this sequence and the alternate one (a f g x). All of the resulting possible sequences are used to trigger new KSARs for POSTULATE-STMT-SEQUENCE.

### 2.2.4 The output

Once WATCH finds one or more acceptable strategies, its final task is to save them in a format that can be useful for other problems. The knowledge source CONSTRUCT-AND-WRITE-KSES outputs each strategy in two different formats, as a set of hierarchically linked KSes and as a skeletal plan.

CONSTRUCT-AND-WRITE-KSES creates a separate new blackboard, with a name such as NEWWATCHKS3432, to store the set of KSes which make up each possible hierarchical strategy. The user is immediately prompted for a name of a file in which to store this new knowledge base. At the same time, a new level is created on the blackboard SKELETALPLAN, to contain the objects that make up the skeletal plan representation of the same strategy. This level contains a PLAN object, as well as strategy and focus decision objects for each part of the hierarchy. It is important to note that WATCH does not automatically store the SKELETALPLAN blackboard on a file; the user must manually use BBEDIT to save this knowledge base after skeletal plans have been created for all strategies.

# 3. NEWWATCH

## 3.1 Introduction

The NEWWATCH program attempts to recognize the primary strategy that explains a sequence of actions by first discovering and removing any interrupting actions. This is done in two phases, each represented by a separate Knowledge Source. First, a sequence of actions is hypothesized to be an interruption if it is not explained by what might be the current strategy. Secondly, this hypothesized interruption is compared against other available strategies to determine whether it is likely to be the intended strategy.

## 3.2 Running NEWWATCH

NEWWATCH is a BB1 application that is an extension of the original WATCH program. Therefore, NEWWATCH is loaded and executed much the same way as WATCH. The user first runs the application program in the modified BB1 to save a trace of the action sequence. Afterwards, NEWWATCH can be loaded into BB1 and executed on the saved trace.

Loading the NEWWATCH program simply involves loading the latest version of the system from the WATCH directory. This process is described in Section 2.1.2. However, the two Knowledge Sources unique to NEWWATCH are not removed from the system.

## 3.3 The NEWWATCH program

### 3.3.1 Hypothesizing an interruption

All possible sequences of actions that may be interruptions are hypothesized by the NEWWATCH KS HYPOTHESIZE-INTERRUPTION. This KS is based on the WATCH KS POST-INITIAL-STATEMENTS which adds each action to the blackboard one at a time. As each new action is ready to be posted (meaning that all the inductive generalization for previous statements has been

completed), HYPOTHESIZE-INTERRUPTION will check to determine whether it may begin an interruption sequence. An interruption may be the intended strategy if the following criteria (PRECONDITIONS) are met:

1. Generalization has been completed for prior actions, resulting in a set of possible generalization strategies for these statements.
2. One of these generalization strategies does not explain the new action. In other words, the generalization is not a more general version of the action, and therefore could not have been a control strategy that prescribed it.
3. This generalization strategy does explain some action that comes later in the sequence.
4. The later action was already executable at this time, and so therefore could have been chosen instead.

The first two criteria check that there is some strategy of which the new action could not have been a part. However, this alone does not guarantee that an interruption has taken place, since the action may begin a new phase of the control strategy. For the action and its immediate successors to be a possible interrupting action sequence, the strategy must be active later. This is guaranteed by the latter two criteria, which show that some later action could have been prescribed by this strategy generalization.

The latter criteria also help to define which actions make up the interruption sequence. All actions from the first one which begins the interruption up to the one prior to the later action which is explained, are the actions considered to be part of the interruption.

This first interruption action may actually be hypothesized to interrupt more than one of the possible generalization strategies, as long as it fits all the criteria for each strategy it might be interrupting. For each of these hypotheses, the KS will build a new object on the INTERRUPTION level of the LEARN blackboard, with all the information about the possible interruption. This includes the list of actions preceding and following the interruption sequence which are explained by the interrupted strategy. The list of actions is crucial, because it is used to rate the likelihood of the hypothesis of an interruption.

### 3.3.2 Rating the interruption

Each interruption is rated when NEWWATCH is ready to do generalization for the last explained action following the interruption. The NEWWATCH KS POSTULATE-SEQUENCE-WITHOUT-INTERRUPTION basically acts the same with the sequence of preceding and following actions, minus the interruption, as the original WATCH KS POSTULATE-STMT-SEQUENCE behaves with a non-interrupted sequence. It is triggered by, and gets its information from, the object on the LEARN.INTERRUPTION level of the blackboard. The preceding and following statements are merged into one action sequence, and are rated with WATCH's same heuristic criteria to determine whether they are explained by a probable strategy. If this sequence minus the interruption rates highly enough, the sequence becomes one of the strategies we are considering. If not, the hypothesis was not a success, because the possibly interrupted strategy was not a reasonable one.

# 4. MetaWATCH

## 4.1 Introduction

MetaWATCH is a BB1 application that examines multiple sets of WATCH output in the context of the problems they solve to incrementally converge on a common "super-strategy". This super-strategy is closer to the experts intended strategy and can be used to solve a wider range of problems. It is output as a BB1 skeletalplan. The basic premise of MetaWATCH is that given multiple problem solving sessions we are more likely to see all branches of the experts strategy fully exercised. This can make the learning process more accurate.

## 4.2 Running MetaWATCH

As with other BB1 applications loading the set-up file for MetaWATCH will load all required files. The MetaWATCH-setup file is contained in the directory "x25:TONY.WATCH.MetaWATCH", this directory contains all MetaWATCH specific code. The set-up file will load files from the BB1 generic directory and from the WATCH directory as well as from the MetaWATCH directory.

The MetaWATCH specific files and their contents are as follows:

WATCH-INPUT-FNS
: Functions used by MetaWATCH to input saved data and store it in the appropriate places.

DOMAIN-FNS

METAWATCH-KS-KB
: The MetaWATCH knowledge sources

COMPARE-STRATS
: This file contains all the functions that perform the strategy tree comparisons that are the basis of MetaWATCH's functioning.

A major problem with inputting data to MetaWATCH is the fact that all the data saved from the application program for use by WATCH, and all the

WATCH output is intended to be used once only. Consequently all the saved KBs from multiple application program and WATCH runs have the same names. To avoid confusion some conventions have been established and the input to MetaWATCH must be edited to conform with these conventions.

Contained in the MetaWATCH directory and loaded from the start-up file is a file called "PROBLEM-CONTEXT-KB". This file should be set up to contain all the domain knowledge to be input the MetaWATCH. These are the blackboards that are common to all the data saved from the different application program runs for WATCH.

On running MetaWATCH the user will be prompted to input the names of files containing the problem specific data for each problem being examined by the system. Each such file should be set up as follows. It should consist of one knowledge base named "Problem*x*" where x is a unique integer differentiating this KB from others. The KB should contain all the problem specific data for this problem and should have at least the following blackboards:

A blackboard "Problem*x*" which in general will contain the objects specific to this problem.

A blackboard "Solution*x*" - the solution blackboard for this problem. A blackboard "Skeletalplan*x*" which contains the skeletalplan blackboard saved from the WATCH run on this problem.

Any other blackboards necessary to define the problem specific knowledge, each of which should be suffixed with an integer x to differentiate them from similar blackboards in other problem KB's.

For each problem the user will also be prompted to input the associated action sequence. This should be the same file as loaded by WATCH for that particular problem - it will automatically be stored on the problem blackboard in the KB associated with the problem in question.

The output from MetaWATCH is of the same form as the skeletalplan output from WATCH. A new blackboard will be created and the MetaWATCH skeletalplan(s) stored on it. The user is then prompted for the name of a file to which this plan should be written.

## 4.3 The MetaWATCH Program

### 4.3.1 Knowledge sources

MetaWATCH consists of the following domain knowledge sources:

Input-data

This KS is the first KS run. It inputs all the data to be used by MetaWATCH. The function *Input-problem-bbs* is called. This function will continually input a KB specified by the user and will create a level on the problem BB called action-sequence on which it stores the action sequence which is also specified by a prompt to the user.

Compare-strategies

This KS generates one KSAR for each comparison that can be made between alternate strategies for different problems. It does this continually until all comparisons have been made, the possible comparisons are generated using a call to *Uncompleted-correspondance-chain*.

The work of this KS is performed by *Compare-strat-trees*, this function will be described in detail below.

Isolate-best-strategies

Creates a new blackboard for output and writes all super strategies to it using a different level for each super strategy.

Write-out-plan

Writes the newly created output blackboard to a user specified file.

### 4.3.2 Comparing strategies

In this section, the functions used to perform strategy comparison are explained. For further explanation on the high level ideas behind this method, refer to Section 5 of the MetaWATCH practicum paper.

Two important points to keep in mind while examining the code are:

1) The new superstrategy is built up as the comparison proceeds. So if the comparison succeeds the new superstrategy is already built, if not the nodes and links created in building the superstrategy must be deleted.

2) Each node in the superstrategy is linked to a node in one or both of the WATCH more specific strategies it combines. When a node in the superstrategy combines nodes from both more specific strategies it is linked via corresponds-to links to both those nodes. If one node in a specific strategy is more general or does not correspond to any node in the other specific strategy the corresponding node in the superstrategy is linked using corresponds-to links to only the more general node. These corresponds-to links are used to navigate around the strategy trees, finding the corresponding parent nodes of superstrategy nodes etc.

The Compare-strat-trees function is the top level function for comparing strategies. It sets up the new super strategy and deals with the results of a positive or negative comparison. However all the real work of comparison is achieved by the Match-children function. It operates as follows: Starting at the top node of each strategy to be compared (child strategies) it recursively checks each of several possible situations stepping through the appropriate action-sequences as it goes. The remaining action sequence and uncompared parts of the strategy trees are passed along with each function call. If at any stage we need to go back over information already compared we can easily fetch this information from the appropriate blackboard. The situations checked by the Match-children function are illustrated in the flow-chart below. These situations are checked by the following functions:

Match-node: Are these nodes exactly the same?

Match-sub-to-super: Match a superaction against a subaction or an action with a super object against the same action but on a subobject.

Both-ok? Check if the nodes at this point are different but both are ok in context.

Match-children: Match two leaf nodes against each other.

Build-super-obj: This function returns an action that is a superset of both the subactions passed. This is then matched against each of the subactions in the context of their problem using Match-sub-to-super. This deals with the case below where one node has a more general action but the other has a more general object.

The various other functions used to compare objects, actions etc are self explanatory.

The only other point to note is that the superstrategy developed is then to be compared to further WATCH strategies. It thus needs an action sequence to give it a problem context for the purposes of comparison. The function Make-super-action-sequence performs this task by amalgamating the two action sequences corresponding to the strategies that were compared to form this superstrategy.

# References

Confrey, Tony, and Hayes-Roth, Barbara, Knowledge Based Strategy Generalization. Knowledge Systems Laboratory Report KSL 89-45, Stanford University, March 1990.

Gans, Andrew, and Hayes-Roth, Barbara, NEWWATCH: Learning Interrupted Strategies by Observing Actions. Knowledge Systems Laboratory Report 89-44, Stanford University. March 1990.

Garvey, Alan, Hewett, Michael Johnson Jr., Vaughan M., Schulman, Robert and Hayes-Roth, Barbara, BB1 User Manual - Common LISP Version 2.0. Knowledge Systems Laboratory Report KSL 86-61, Stanford University. August 1987.

Harvey, Jeff, WATCH - Inductive Learning of Control Abstractions. Knowledge Systems Laboratory, Stanford University. Unpublished 1987.

# Appendix 1
# An Action Sequence

At the end of the application run, WATCH saves a trace of the action sequence in a data file. For each action, it stores:

(action-number action-sentence parsed-action-sentence
BB1-cycle-that-action-became-executable cycle-that-action-was-executed)

WATCH also saves a list of all the KSARs that were not executed, so that MetaWATCH will have a complete list of all available actions.

(ksar-number ksar-event-sentence ksar-parsed-event-sentence
final-ksar-status cycle-that-ksar-became-executable
cycle-that-ksar-became-obviated)

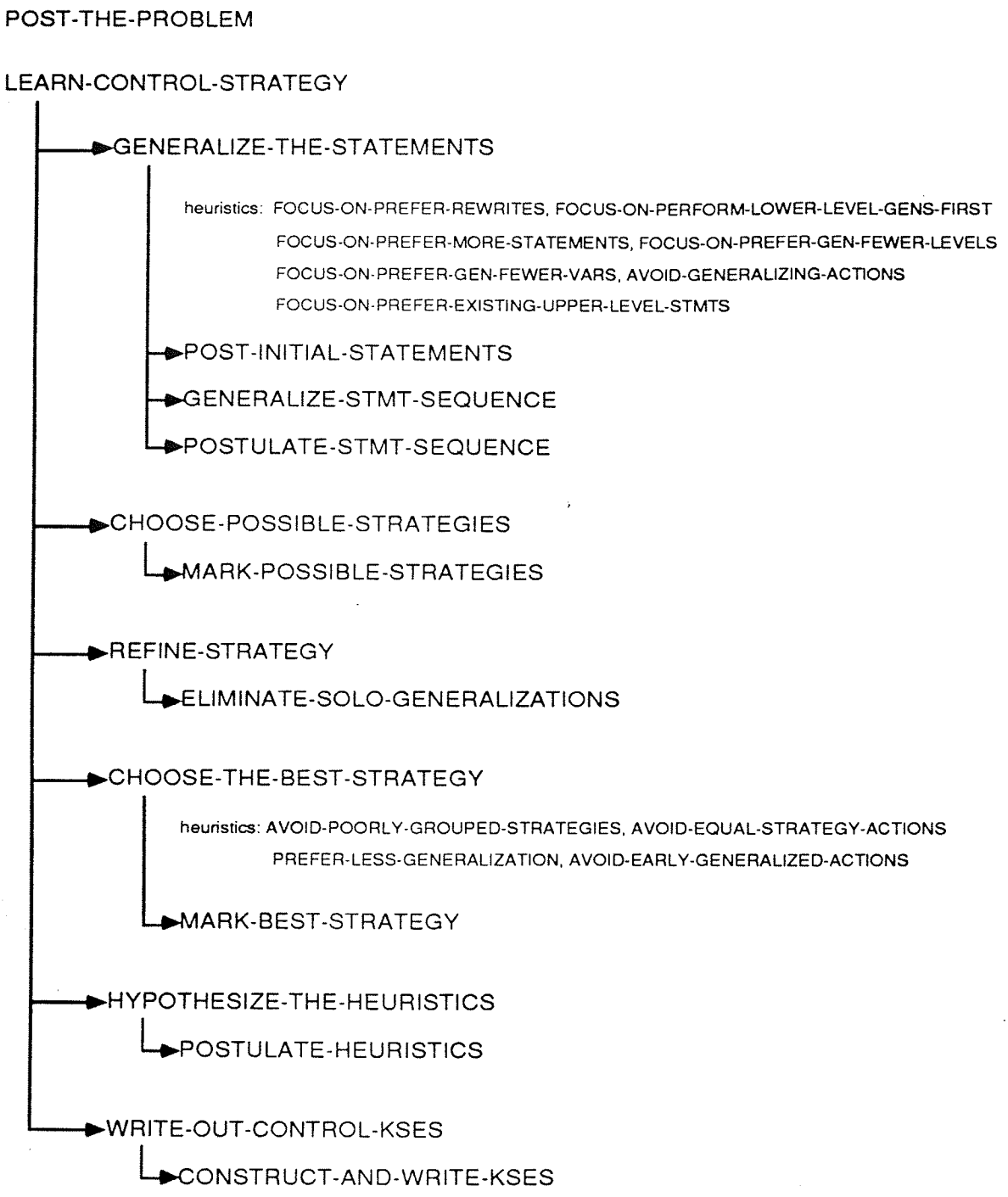
This example is from the PROTEAN application system.

```
(bb1::add-data-to-system 'action-sequence
    '((1 (accord.event.did-create solution.partial-arrangement.pa1)
        ((accord.event.did-create) (solution.partial-arrangement.pa1))
        1 6)
    (2 (accord.event.did-include problem.solid.helix3 in solution.partial-arrangement.pa1)
        ((accord.event.did-include) (problem.solid.helix3) (solution.partial-arrangement.pa1))
        6 10)
    (3 (accord.event.did-include problem.solid.helix1 in solution.partial-arrangement.pa1)
        ((accord.event.did-include) (problem.solid.helix1) (solution.partial-arrangement.pa1))
        6 11)
    (4 (accord.event.did-include problem.solid.helix2 in solution.partial-arrangement.pa1)
        ((accord.event.did-include) (problem.solid.helix2) (solution.partial-arrangement.pa1))
        6 12)
    (4 (accord.event.did-orient solution.partial-arrangement.pa1 about problem.solid.helix1-1)
        ((accord.event.did-orient) (solution.partial-arrangement.pa1) (problem.solid.helix1-1))
        11 20)
```

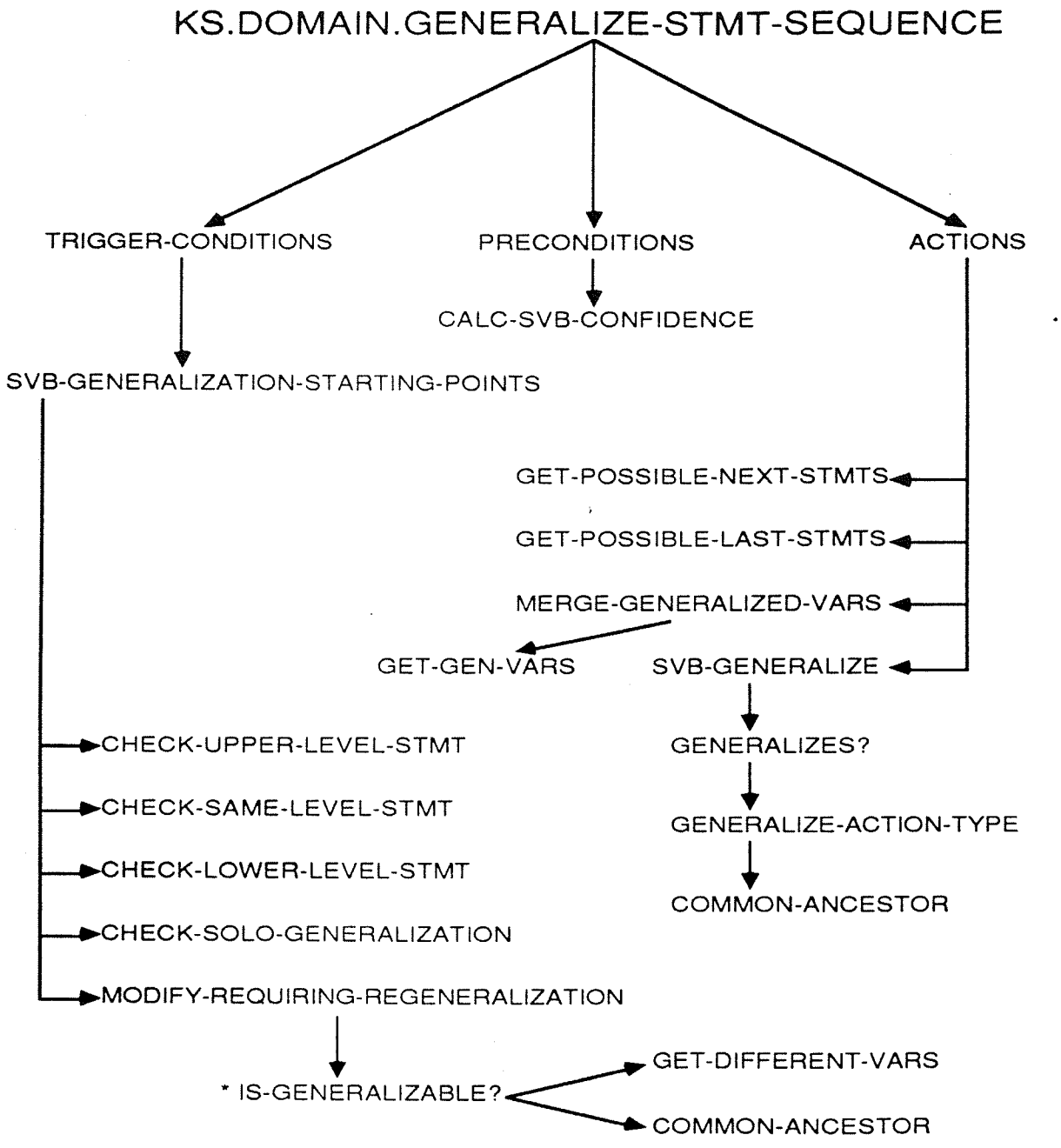
```
(5  (accord.event.did-anchor solution.solid.helix2-1 to solution.solid.helix1-1
      in solution.partial-arrangement.pa1 with problem.constraint-set.cseth1h2)
    ((accord.event.did-anchor) (solution.solid.helix2-1) (solution.solid.helix1-1)
      (solution.partial-arrangement.pa1) (problem.constraint-set.cseth1h2))
    20 28)))

(bb1::add-data-to-system 'nonexecuted-ksars
  '((control-data.ksar.ksar26
      (accord.event.did-orient solution.partial-arrangement.pa1 about solution.solid.helix3-1)
      ((accord.event.did-orient) (solution.partial-arrangement.pa1) (solution.solid.helix3-1))
      obviated 11 20)))
```

# Appendix 2
# The WATCH KSes

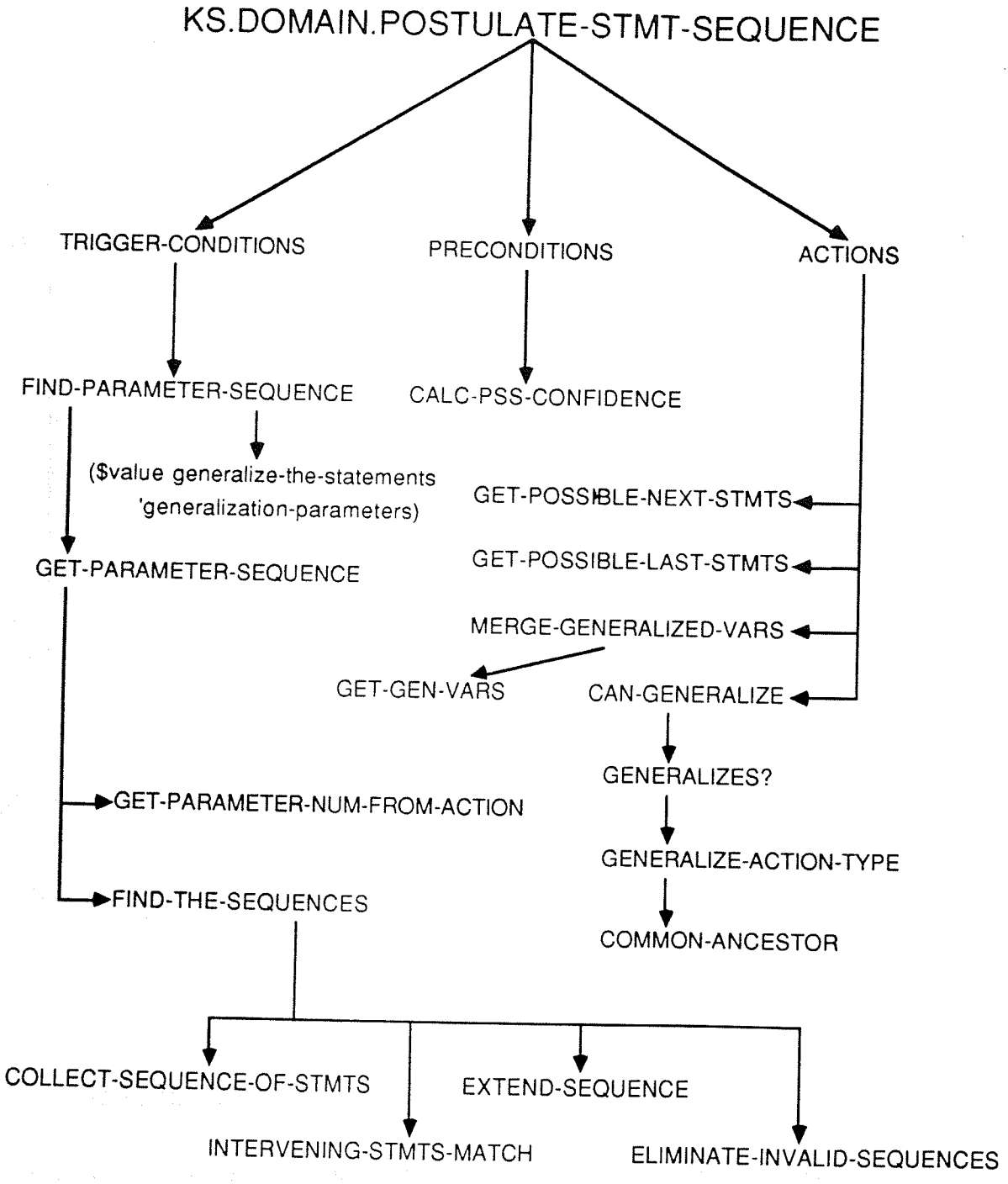


# Appendix 3
# KS GENERALIZE-STMT-SEQUENCE

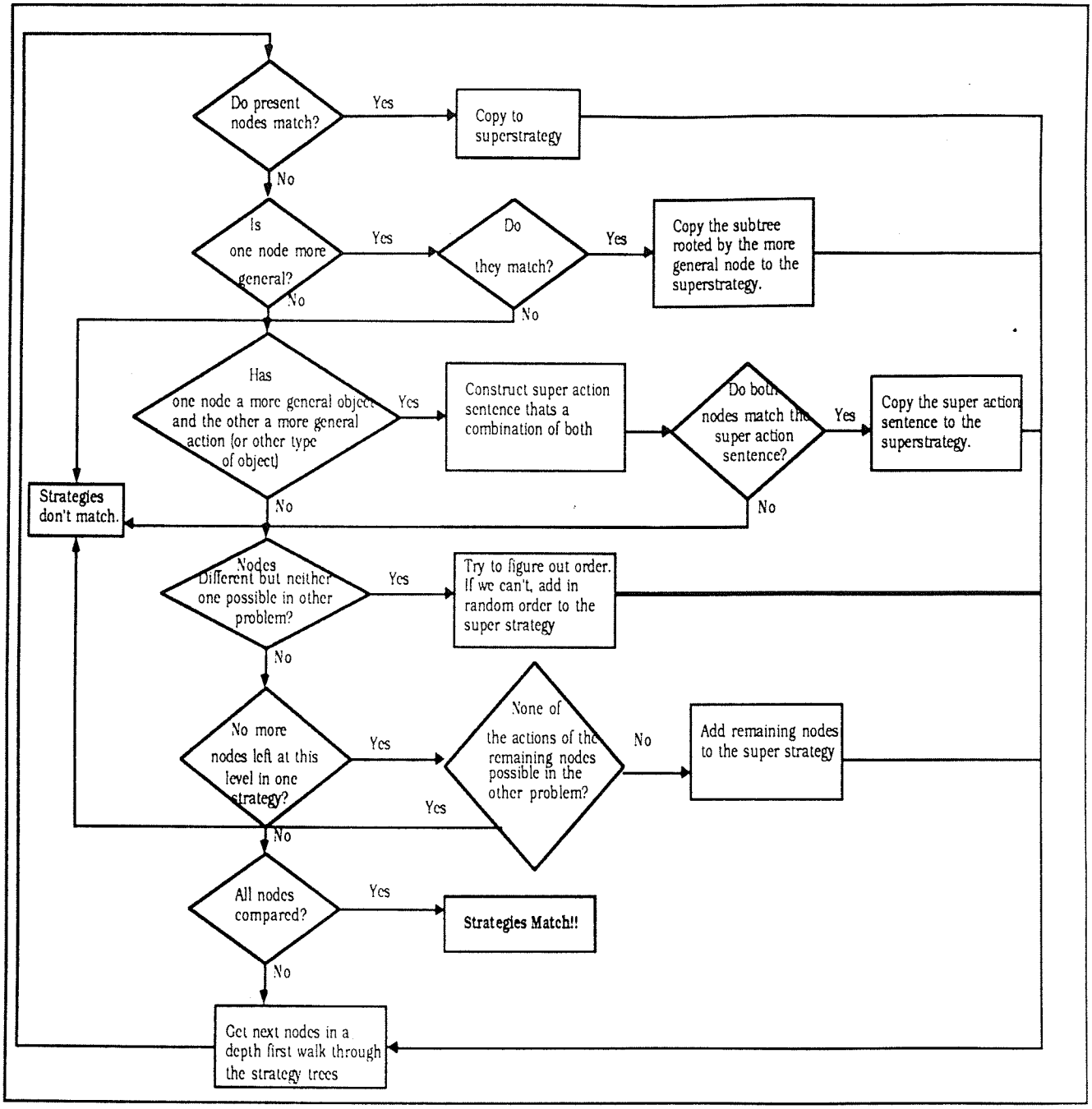



* all functions are in WATCH-DOMAINFNS except IS-GENERALIZABLE? which is in WATCH-WRITEOUT

# Appendix 4
# KS POSTULATE-STMT-SEQUENCE



# Appendix 5
# MetaWATCH Algorithm




The algorithm used to compare two strategies. See text (Section 4.3.2) for an explanation of how some of the steps are implemented.