Agent Design for Automatic Use of a Software System: A Case Study with a Soar Agent for Mathematica

Dhiraj K. Pathak

CMU-RI-TR-93-30

Submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy in Robotics at Carnegie Mellon University.

Advisors: Dr. Allen Newell and Dr. David Steier

The Robotics Institute
Carnegie Mellon University
Pittsburgh, Pennsylvania 15213

May 1993

This research has been supported by the Engineering Design Research Center, a National Science Foundation Engineering Research Center.

Carnegie Mellon

Robotics

Thesis

Agent Design for Automatic Use of a Software System: A Case Study with a Soar Agent for Mathematica

Dhiraj K. Pathak

Submitted in Partial Fulfillment of the Requirements
for the Degree of
Doctor of Philosophy
in the field of Robotics

ACCEPTED:

MAJOR PROFESSOR	DATE 5/6/93
DEAN	DATE 5/14/93
PROGRAM DIRECTOR	

APPROVED:

PROVOST	DATE 14 May 1993

Abstract

It is useful to build agents to mediate in the use of complex software systems. To understand some dimensions of mediating agents, this thesis presents an agent design for the automatic use of a computer algebra system (CAS). Such an agent must take a task specification from the user and produce a plan for achieving the task while also producing a plan for the CAS. In particular, I use the Soar architecture to build an agent for using Mathematica, guided by a study of protocols of human users of Mathematica. The general design principles for mediating agents suggested by the particular agent for Mathematica include a blackboard-style control structure using a production systems architecture, a dual-space planning function, integration of planning and execution, and mechanisms for automatic knowledge acquisition through environment interaction.

Acknowledgements

Dr. Allen Newell made this thesis possible by accepting to be my advisor. I could not have asked for a better guide. I tried to learn from Allen as much as I could. The research could not have been completed without the support of Dr. David Steier. David has been an advisor from the inception of my thesis research and did not allow Allen's absence to impact my thesis work. I benefited substantially from David's advice and calming influence during turbulent times. I had excellent thesis committee members: Dr. Tom Mitchell, Dr. Raj Reddy, and Dr. Herb Simon. Each member of my committee has helped me in my research. The only way to thank them is by making the best use of the skills they taught me. I thank Dr. Dana Scott for helping me with my thesis proposal. Dr. Jaime Carbonell provided advice and inspiration at various points during my thesis research. I thank Dr. Stephen Smith for initiating me into AI research and for always being there to provide counsel. Dr. Bruce Krogh introduced me to graduate-level research. Bruce gave me a solid foundation to build upon. I thank Dr. Takeo Kanade for advice on numerous occasions that helped me maintain course. I am very grateful to Dr. Fritz Prinz for supporting my research through the EDRC. Several members of the Soar group at CMU helped me understand and work with Soar: Bob Doorenbos, Rick Lewis, Gary Pelton, Thad Polk, Tom McGinnis, and Ajay Modi.

I thank Shri Raj K. Singh, Dr. Gurmukh Singh, and Dr. Manjeet Singh for invaluable help in adjusting to life in the US. I thank my friends for helping me on the journey through graduate school: B. Gurumoorthy, R. Rajkumar, R. Sreenivas, Harsha Baxi, Deepak Kulkarni, Ramesh Mahadevan, Dave Plaut, Shree Nayar, Reg Willson, Jim Rehg, Benli Pierce, Ted Tschang, Anurag Acharya, Anurag Gupta, Mark Maimone, Milind and Sonali Tambe, Michel Roboam, Ajai Kapoor, Alok Jain, Soumitra Bose, Kikuo Ohgaki, and Toshi Satomi. My office mates: Rich Goodwin, Jim Blythe, and Zoran Popovic provided abiding fellowship. I thank Ms. Jean Harpley, Ms. Elaine Atkinson, Ms. Nancy Serviou, Ms. Marce Zaragoza, and Ms. Kathy McNiff for their help.

I thank my parents and my brother and sister for non-trivial encouragement.

Contents

List of Figures

Chapter 1

Introduction

1.1 Thesis Definition

My thesis is that general principles of problem solving and learning underly the design of an agent meant to automatically use a software system. To find out some of these principles I have designed and implemented a Soar [52, 64] agent that mediates between a user and Mathematica [75].

1.2 Motivation

It is useful to build agents to mediate in the use of complex software systems. A user gives to a mediating agent a task that can be performed using a software system that the agent is capable of using. It is the agent and not the user that interacts with the software system. The user need not know how to use the software system and not incur the learning cost associated with any complex software system. Even if a software system is familiar to a user there may be overhead in using it for a given task. The user must produce a plan for the task while also planning for the use of the software system for the task. Therefore, the cognitive demands of using a software system for a task can be substantial. Lapses in attention, failure in memory retrieval or incompleteness in acquired knowledge of the software system can all increase the cognitive load. A mediating agent provides a means for users to escape from the tedium of software system use.

1.3 Methodology

The class of all software systems for which mediating agents might be useful is clearly large. In this thesis we have focussed on the design of an agent for Mathematica. Mathematica belongs to a class of software systems that provide a large number of individual functions capable of significant computation. This characteristic enables tasks to be performed by the execution of a sequence of individual software system functions. Any given task can either be mapped directly onto a software system function or the task is processed until subtasks that can be directly mapped onto software system functions are obtained. Tentativeness in both task knowledge and software system knowledge compel the agent to execute the functions as they are determined. Such a use of a software system is the essence of exploratory programming by people using an interactive software system

We use protocols of human users of Mathematica to guide our agent design. The agent is tested on simple problems from sensitivity analysis, trajectory planning, and geometric reasoning. The generality of the agent design is assessed by applying it to some other software systems. The implemented agent provides us the problem solving and learning principles sufficient for the class of interactive software systems exemplified by Mathematica.

1.4 Guidance from protocols

Protocols of subjects using Mathematica to perform engineering design tasks are obtained. A protocol is a record of the activities that a subject engages in while performing a task. Protocol data includes a record of all exchange with Mathematica and (sometimes) written deliberations recorded while performing the task. Protocols are analyzed [53, 23, 32] to characterize processes required to perform tasks with Mathematica. Let us analyze the protocol in Appendix A. The written deliberations in this case are limited to a statement about the time at which the protocol session was conducted and a statement of the task itself. The task is to calculate an interpolating polynomial, $th(t) = d + ct + bt^2 + at^3$, given that, $th(t0) = th0$, $th(t1) = th1$, $\partial th/\partial t(t0) = 0$, $\partial th/\partial t(t1) = 0$.

We term a user input to Mathematica, an *external action specification* (external action, for short), since it is an action specification formulated by

the user for external execution (by Mathematica). `In` and `Out` are labels used in Mathematica to correspond to a user input and the corresponding output generated by Mathematica. `In[1]` is an external action to differentiate the interpolating function, $th = d+ct+bt^2+at^3$, with respect to the independent variable, t. We assume that for each input to Mathematica there exists an *internal action specification* (internal action, for short) that is the user's reason for formulating an external action. An internal action is a function applied to some data. For `In[1]`, the internal action is (derivative $th = d+ct+bt^2+at^3$ t), which applies the function, derivative, to the interpolating polynomial, $th = d + ct + bt^2 + at^3$, and the independent variable, t. Let *o1* denote this internal action. Sometimes there is direct evidence in the written deliberations of a protocol for the internal action corresponding to an external action. When there is no direct evidence in the subject's protocol for an internal action, we must infer it in light of the external action. *o1* is assumed to have been specified because the input task specifies a constraint on the derivative of the interpolating polynomial. *o1* is termed an *input internal action* since it is obtained directly from the input task.

The input task specification and the internal and external actions are some of the elements of the subject's *cognitive state*. To characterize the constituency of the cognitive state is one objective of protocol analysis. The second objective is to characterize mappings on cognitive state sufficient to explain protocol data. Two mappings have been discerned thus far. One maps the input task onto internal actions. A second maps internal actions onto external actions. Additional elements of the cognitive state are identified by analyzing these mappings. The first mapping requires that the subject has access to task knowledge. For example, to process the task instance the subject must know about computing interpolating functions. The second mapping requires that the subject has access to knowledge of Mathematica functionality. For example, it must be known that the `D` function computes a derivative in Mathematica and that it takes two arguments, the expression to be differentiated and the variable of differentiation.

`Out[1]` is the output produced by Mathematica for `In[1]`. As in language understanding [68], we assume that the data is mapped onto a *meaning*. Part of the meaning is context-independent. For example, the context-independent meaning for `Out[1]` can include the information that it is an equation whose left hand side is zero and whose right hand side is a cubic polynomial. Another part of the meaning is context-dependent. For exam-

ple, the context-dependent meaning for `Out[1]` can include the information that it is the derivative of $th == d + ct + bt^2 + at^3$ with respect to t and that the derivative of th is 0 with respect to t. The next external action is `In[2]` that differentiates the interpolating polynomial with respect to t specifying that th is dependent upon t through the use of the `NonConstants` option for `D`. The internal action for `In[2]` is inferred to be, (and (derivative $th = d + ct + bt^2 + at^3$ t) (dependent th t)).[1] Presumably, the subject's mathematical knowledge flagged off the context-dependent meaning for `Out[1]` that had the derivative of th to be zero as being anomalous given access to task knowledge that had th dependent upon t. We assume that the response to the detected anomaly is to augment the internal action with the property violated in the external action execution. By accessing additional Mathematica knowledge, namely, the form of the `D` function for specifying dependence of a variable on the independent variable, the new internal action is mapped onto `In[2]`. (`In[2]` is seen to be unaccepted by Mathematica the first time around. This is not a problem of Mathematica but rather an idiosyncrasy of the computing environment we used to interact with Mathematica.) `In[2]` is properly interpreted by Mathematica the second time around. `Out[2]` is the data produced by Mathematica for `In[2]`. Using Mathematica knowledge, the context-independent meaning for `Out[2]`, includes the information that the left hand side of the equation is the derivative of th with respect to t.

We assume that when an external action produces an anomalous result and is followed by an alternative external action that produces an acceptable result the subject can acquire new knowledge of how to use Mathematica. For example, the new knowledge here is to include the dependency among variables when specifying an internal action to differentiate an expression. One way to confirm such acquisition actually having taken place is to await the recurrence of a problem context that would be mapped onto the first internal action in the absence of acquisition but to the second if knowledge had been carried forth from a previous interaction. By now we have identified the primary elements of the cognitive state and the primary mappings on these states. The rest of this protocol is analyzed in a similar manner. Protocol analysis is followed up with an implementation using Soar which requires us to devise representations for the cognitive states and operators

[1] ie. a conjunct of the previous internal action, (derivative $th = d + ct + bt^2 + at^3$ t), and the relation, (dependent th t).

for the mappings. An implementation serves to make concrete and complete the mappings that are required to simulate protocol activity. It also becomes the instantiation of an agent to automatically use Mathematica. These are some global considerations that guided our protocol analysis effort:

1. A protocol is data produced by the execution of the cognitive processes of the subject and to analyze a protocol is to determine the underlying process model much like system identification [69].

2. A lot of the human cognitive apparatus is revealed in a typical protocol and selectivity in analysis is called for keeping in mind the objectives of the study. Our objective is to demonstrate an implemented agent that can use Mathematica and possibly be a blueprint for the automatic use of other software systems as well. At the same time there is a methodological commitment to approach this goal via understanding human cognition in the use of Mathematica. We did not have as an analysis goal the complete explication of protocol data. This would have been called for had we wanted to either evaluate Mathematica as a software system from a usability standpoint or assess the cognitive skills of subjects using Mathematica. In particular, these portions of protocol data were summarily ignored:

   (a) Recovery from syntax errors. The assumption is that the agent will not commit syntax errors.

   (b) Extended periods of observable non-activity by the subject brought on by truly complex tasks. It is assumed that the agent presses on with the application of whatever relevant knowledge it has to perform the given task and does not exhibit non-activity.

   (c) Sometimes the user consults the manual for guidance. This is obviously true in the early stages of acquiring a new software system. This activity was not included in the protocol data collected. We did not want to set as a goal agent ability to acquire knowledge from the manual noting that non-trivial language processing capability is implied to support this activity in the agent.

   (d) Resolution of system software concerns are part of subject protocols. For example, using Mathematica remotely over the network means that sometimes the network will go down. A human user of

course recognizes this situation and responds appropriately. We assume that the agent does not have to contend with system software issues.

1.5 Contributions

These are the contributions of my dissertation:

1. Design principles underlying an agent that can automatically use a software system belonging to a class exemplified by Mathematica.
   (a) A blackboard-style control structure realized within a production system architecture.
   (b) An object-oriented knowledge representation augmented with a predicate-calculus-like language.
   (c) Dual-space planning function for planning for the input task and planning for the software system.
   (d) Automatic mapping of tasks onto software system functions.
   (e) Integration of planning and execution.
   (f) Automatic knowledge acquisition through environment interaction.
2. Demonstration of automation of task-level symbolic calculations in engineering design.

1.6 Thesis organization

The rest of the thesis document is organized into these chapters:

- Chapter 2: presents a process model for using Mathematica consisting of seven processes: supervisory, formulation, interpretation, external memory, knowledge acquisition, input, and output. The process model is derived from the agent implementation described in Chapter 3. It is presented prior to the implementation because its purpose is to provide the reader with an overview of what sort of computations the agent performs in using Mathematica.

- Chapter 3: describes the implementation of a Soar agent for using Mathematica. First, the state representations are described. Second, the problem spaces and their operators are described. Six of the processes - supervisory, formulation, interpretation, external memory, input, and output - are concerned with the performance function of the agent. Their implementation is described in this chapter. The knowledge acquisition process is fundamentally different from these performance processes. It is concerned with explaining how new knowledge can be acquired through interactions with Mathematica. Because of its independent status its discussion is consigned to Chapter 4.

- Chapter 4: describes automatic knowledge acquisition through interaction with Mathematica.

- Chapter 5: describes various experiments with the agent for Mathematica. Also discussed are a retrieval task for a FrameKit [59]-based database system and a discrete-time linear system computation using SignalProcessing [25].

- Chapter 6: describes the abstraction underlying the agent design for Mathematica.

- Chapter 7: describes some related work in automated deduction, intelligent interfaces, software reuse, and automatic programming.

- Chapter 8: presents the conclusions of this dissertation and a discussion of future work.

Chapter 2

A process model for using Mathematica

In this chapter a process model for using Mathematica is described. The model, illustrated in Figure 2.1, consists of seven processes: supervisory, formulation, interpretation, external memory, knowledge acquisition, input, and output. The process specifications are obtained from the implementation of the processes found in the protocols of subjects using Mathematica to perform tasks. A process is specified as a set of mappings on elements of the cognitive state. [1] It is useful to note two types of data that serve to give shape to the process model. First, there is the protocol data whose analysis yields mappings on the cognitive state sufficient for task performance. Second, there is the implemented computer program that actually performs the tasks whose protocols have been analyzed. However, the mappings that we include in the process model for the task are only those that have a basis in the implementation. The utility of a process model lies in its providing us with a sense of what is the nature of the computation sufficient for performing tasks of interest [2]. Insistence on an implementation as the basis for the proposed process model is motivated by the assumption that concrete representations and algorithms provide one measure of the computational feasibility of the mappings making up the process model. The process model is then a description of the computation independent of the details of the

[1] The cognitive state consists of the representations used during problem solving.

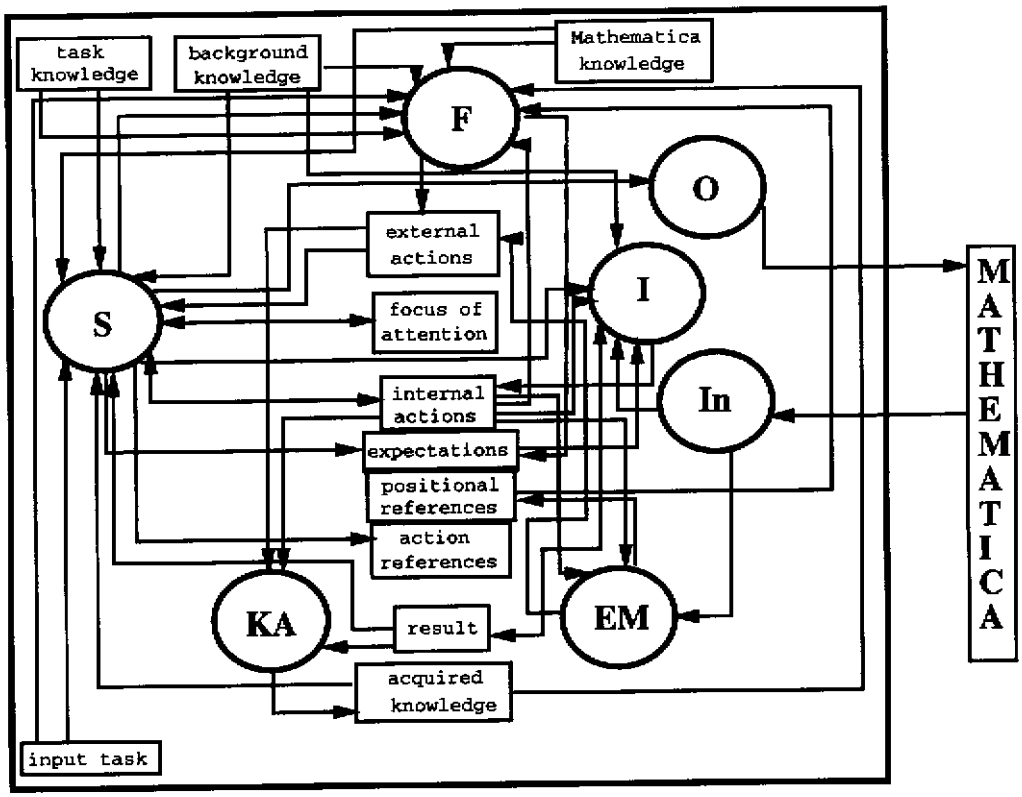
[2] Here, using Mathematica to perform symbolic calculations.

implementation and as such can be viewed as a knowledge level [50] specification of the agent or a computational theory for the task [43, page 24]. In the following sections each of the processes is specified in turn.

2.1 Supervisory process specification

The supervisory process is concerned with the synthesis and execution of a plan for the input task. It consists of these mappings:

1. Use task knowledge and background knowledge to specify function applications whose computation may produce a result for the input task. These function applications are termed internal actions. For example, if the task is to compute the sensitivity of a linear system, then an internal action can be $(\partial T/\partial K)(K/T)$ where T is the system transfer function and K is the open-loop gain.
2. Determine the initial internal actions by interpreting the input task representation. There are several ways to determine the initial internal actions. For example, a definition for the input task may be known from which the initial internal could be determined. So, for the input task to calculate the sensitivity of a linear system the definition of linear system sensitivity may be used to specify the initial action, $(\partial T/\partial K)(K/T)$ where T is the system transfer function and K is the open-loop gain.
3. Select an internal action to focus on for further processing.
4. Remove from focus an internal action.
5. Using Mathematica knowledge, direct the formulation process to determine an external action for the internal action in focus. For example, if the calculation is to find the derivative and it is known that Mathematica can perform differentiation then the supervisory process changes the status of the calculation to formulate. Here the decision to formulate a calculation is guided by direct knowledge associating the calculation with an appropriate Mathematica function.
6. Direct the formulation process to determine an external action for the internal action in focus without the knowledge of whether or not it is possible to produce a formulation for the internal action.
7. Select an external action and evoke the output process for transmission to Mathematica.



Input task: a function application
Task knowledge: knowledge of input functions
Background knowledge: knowledge relevant to tasks

S: supervisory process
F: formulation process
I: interpretation process
EM: External memory process
KA: Knowledge acquisition process
In: Input process
O: Output process

Figure 2.1: The process model for using Mathematica.

8. Choose to verify the result of an external action vis a vis an internal action by evoking the interpretation process.
9. Create expectations for internal actions. These expectations are properties of the result obtained by function evaluation. For example, if the computation is a product then the expectation is that the result will be a product. Expectations are not limited to the results of a calculation and can be arbitrary constraints on the outcome of a calculation. For example, if a differentiation with Mathematica produces an anomalous result then the next calculation may be created with the expectation that the anomaly observed in the previous instance be not obtained this second time.
10. Create action references for internal actions upon availability of results of corresponding external actions. Action references are a representation of an external action having been executed in service of an internal action.
11. Create a copy of an internal action if the external action execution for the internal action produces anomalous result.
12. Specify the post-condition of an internal action. For example, for an internal action, (simplify (o1)), where o1 is a polynomial object, the post-condition may be to compare the result of the simplification with o1. A post-condition is an intended action but not an internal action as yet and can be made into an internal action after the completion of the internal action that it was the post-condition of.

2.2 Formulation process specification

The formulation process is concerned with planning an external action for an internal action. It consists of these mappings:

1. Determine an external action specification for an internal action by using Mathematica knowledge. These function specifications are termed *external actions*. For example, for the internal action, (derivative (o1 o2)), where o1 and o2 are a polynomial and a variable respectively, the `D` function of Mathematica may be selected using the knowledge that this function computes a derivative.
2. Use task representation to determine arguments for external actions.
3. Determine an external action using background knowledge.

4. Recover from a wrong external action specification by selecting another external action specification.
5. Generate strategies for using Mathematica for internal actions.
6. Induce the use of a known function for an internal action.
7. Determine whether to use a Mathematica function for a calculation.
8. Create external action intentions. For example, two intentions corresponding to the internal action (factor (o1)) can be, (use `Factor`) and (use `FactorInteger`).
9. Determine the transformation to a calculation based upon the interaction experience.
10. Determine a program for computing input function. A program is the sequence of external actions executed by the agent during its attempt to compute the input function. The program is generated incrementally.
11. Create expectations for external actions. These expectations are properties of the result obtained by the execution of external actions. For example, an external action using the `Solve` function of Mathematica may have the expectation that the result will consist of an assignment of values to the unknown variables.

2.3 Interpretation process specification

The interpretation process is concerned with the mapping of the data received from Mathematica onto a meaning representation. It consists of these mappings:

1. Create a representation of the data obtained by the input process from Mathematica. Some of the meaning built up for the data is independent of the context in which it was produced. Other meaning is ascribable to the data because of the context of its generation. The expectations created by the supervisory and the formulation processes are part of the context-dependent meaning of the result.
2. Use background knowledge to recognize anomalies in the result.
3. Verify by matching expectations with the properties of the result.
4. The interpretation process is invoked when there is a data object received from Mathematica in response to an input to Mathematica.

2.4 External memory process specification

The external memory process is concerned with maintaining indices pointing to the results generated in the past and retrieving indices to past results by matching references to these results in subsequent actions. It consists of these mappings:

1. Create positional references to data produced by Mathematica.
2. Determine whether data corresponding to a result of a previously completed calculation is available externally.
3. Retrieve positional reference in order to specify an external action.

2.5 Knowledge acquisition process specification

The knowledge acquisition process is concerned with the addition of new knowledge to the long-term memory. It consists of these mappings:

1. When an external action produces an anomalous result then acquire the knowledge that the association between the external action and the corresponding internal action is inappropriate.
2. When an external action is determined that produces an acceptable result where previously an anomalous result was obtained using another external action then acquire the knowledge that the association of the new external action with the internal action is appropriate.

2.6 Input process specification

The input process is concerned with the processing of data received from Mathematica. It consists of these mappings:

1. Receive data from Mathematica and evoke interpretation process.
2. Receive data from Mathematica and evoke external memory process.

2.7 Output process specification

The output process is concerned with the transmission of data to Mathematica. It consists of these mappings:

1. Transmit data from supervisory process to Mathematica.
2. Record completion of transmission.

Chapter 3

A Soar-based agent for using Mathematica

In this chapter a Soar-based agent, illustrated in Figure 3.1, that automatically uses Mathematica for symbolic computations in engineering design, is described. The agent mediates between an engineerand Mathematica. The agent accesses knowledge of various tasks such as sensitivity analysis and factorization, knowledge of Mathematica functionality, and knowledge of mathematics to process an input task representation provided by the engineer and determine how to use Mathematica for the computations required by the input task. The chapter begins with a section describing Soar primitives of import to the agent design reported in this chapter. The next section lays out the overall organization of the Soar-based agent. Then, there is a discussion of the representations used in the agent. These representations include the input task representation, the representation of Mathematica functionality, the representation of mathematical knowledge, and the problem solving representations manipulated by the agent's processes.

3.1 Soar

Soar is a software system, motivated both as a unified theory of cognition [52], and as an integrated intelligent system [64], whose primitives are defined as

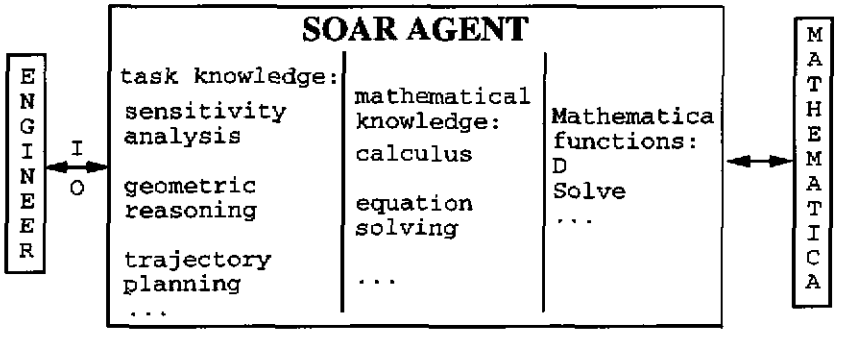


Figure 3.1: An agent for performing symbolic computations using a computer algebra system.

follows [1]:

1. Long term memory: is implemented as a production system [27].
2. Problem spaces: defined by a set of operators that can apply in these problem spaces. Soar selects a problem space for a goal based upon problem space preferences available in the preference memory. These preferences are retrieved by productions contained in Soar's long-term memory. Soar selects a state for a selected problem space based upon state preferences available in the preference memory retrieved by productions in long-term memory.
3. Operators: produce state change to achieve tasks. After having selected a state for a problem space Soar selects an operator based upon operator preferences available in the preference memory retrieved by productions in long-term memory. An operator is implemented as a set of long-term memory productions that change the state and/or

[1]The description of Soar's primitives provided here is motivated by their use in the agent design reported in this dissertation. Alternative descriptions of these primitives may be found in [51, 52, 55, 64]. Also, there is no attempt to describe all the behavior that Soar can manifest. Attention is restricted to giving the reader a sense of Soar behavior found useful in the agent design for automatic use of software systems. Further, it is not clear how a complete description of all the behavior producible by Soar can be provided since that would be like specifying all the programs that can be computed using a programming language. Version 5.2 [38] of the Soar system was used.

add preferences. Soar applies a selected operator by firing the productions implementing the operator. An operator proposal rule in Soar's long-term memory adds a preference for an operator to the preference memory. The preference for the operator remains in the preference memory as long as the working memory matches the conditions of the operator proposal rule. When the working memory fails to match the rule conditions the preference for the operator is withdrawn from the preference memory. This by itself will not cause Soar to remove the operator from the goal-context. Soar requires that a reconsider preference for the operator be present in the preference memory in order for operator to be removed from the goal-context. Another way to remove a selected operator is to have a reject preference for the operator in the preference memory. These are then two ways by which Soar terminates a selected operator.

4. Goals: are created by Soar to solve impasses. When a goal has been created Soar selects a problem space for it. If a problem space is found, Soar selects a state for the problem space. If a state is found, Soar selects an operator for the problem space. Nominal processing for a goal then consists of selecting a problem space, selecting a state, selecting an operator, and applying the selected operator to change the state. Subgoals are how Soar transitions from one problem space into another. For each goal, there is a goal-context comprised of a problem space for the goal, a state for the problem space, and an operator for the problem space. To process a goal, Soar attempts to instantiate the goal-context by selecting a problem space, a state, and an operator. Once these elements are obtained for a goal, Soar applies the selected operator to produce state change. The commitment to elements of a goal stack occurs at the end of a decision cycle. A single commitment is made at the end of a decision cycle. For example, say a goal's problem space has been determined previously. At the end of the next decision cycle Soar will expect to commit to a state for the problem space. If the desired commitment cannot be made then an impasse is detected and a subgoal is created to solve the impasse. It is possible for preferences to be added during a decision cycle that make a unique selection for a previously determined goal-context element impossible.
5. Impasses: occur when a. a selected operator cannot be applied b. there are multiple candidate operators to select among c. there is no

operator that can be selected for a problem space. [2] Soar responds to an impasse by creating a subgoal. If no operators can be applied to the state a subgoal, termed *state-no-change* (SNC), is generated. If a selected operator cannot be applied in the current state a subgoal, termed *operator-no-change* (ONC), is generated. A SNC is resolved when a candidate operator is proposed for the parent problem space for now further problem solving is possible in the parent problem space. An ONC is resolved when the operator has been applied by changing the parent state. Upon the resolution of either the SNC or the ONC the problem solving in the child problem space is automatically summarized to build a *chunk* that is added to the existing collection of productions. For the SNC, the chunk action is a preference for an operator and its conditions consist of the elements of the parent state that were matched to produce this preference. For the ONC, the chunk action consists of changes to the parent state made in the child problem space and its conditions consist of the elements of the parent state that were matched to produce these changes.

6. Working memory: consists of the goal hierarchy created by impasses, acceptable preferences for operators, and state representations. Long-term memory productions match working memory to retrieve preferences for problem spaces, states, operators, and state objects.
7. State: consists of a set of attribute value pairs. The working memory contains a symbol denoting a state. A state can be viewed as a semantic network. The state symbol is a node in the network. An attribute-value pair is then a link from the state node to a value node. Attribute values need not be restricted to links to the state node. These can be links among value nodes as well. Thus, value nodes can be indirectly connected to the state node. A constraint is for all value nodes in the network to be connected either directly or indirectly to the state node.
8. Decision cycle: fires all productions in long-term memory that match the working memory. Each production firing adds preferences to the preference memory for one or more problem spaces, states, operators, and state objects. Preferences can be added for a problem space, a state, an operator, or a state object of any goal in the goal hierarchy. Preferences for state objects change working memory. Changes in

[2] Refer to [38] for additional impasses created by Soar.

working memory can trigger further production firings. Finally, Soar reaches quiescence when no production firings are possible.

9. Decision procedure: is applied by Soar at the end of a decision cycle to select from among possible choices for problem spaces, states, and operators. It consists of a set of rules to order the preferences for problem spaces, states, and operators. There are two possible outcomes of the decision procedure. If the last decision procedure had selected an operator for a problem space then the current decision procedure tries to apply the selected operator. If the selected operator cannot be applied, an impasse is detected and Soar creates a subgoal to solve the impasse. If multiple choices exist for a problem space for a goal, or a state for a selected problem space, or an operator for a problem space then an impasse is detected and Soar creates a subgoal to solve the impasse. If no operators are available for a problem space then an impasse is detected and Soar creates a subgoal to solve the impasse.
10. Chunking: creates new long-term memory productions. A chunk (a name used to refer to a production created by Soar as opposed to being supplied by the programmer) is created when Soar finds a result for a subgoal. A subgoal result is a modification to a supergoal state produced by the application of a subgoal operator. A subgoal operator is an operator of the problem space selected for the subgoal. For a subgoal result, Soar determines the working memory elements that were matched by productions that led to preferences for the subgoal results. Those working memory elements that belong to the supergoal state and the supergoal, the supergoal problem space, and the supergoal state make up the conditions of the chunk created by Soar. The right hand side of the chunk is made up of the subgoal results. [3]
11. Preferences: are (symbolic) weights for problem spaces, operators, states, and state objects. Some examples of preferences are *acceptable* and *best*. Acceptable preferences for operators are part of Soar's working memory.
12. Lookahead search: when two or more operators are applicable in a problem space and there is not sufficient preference knowledge to make a unique selection Soar can make use of the lookahead search mechanism

[3]This is an accurate but necessarily abstracted description of chunking. A detailed description of chunking can be found in [38].

to make a selection among the applicable operators. Soar responds to a subgoal created for the impasse (called a tie impasse) brought on by a unresolved contention among applicable operators by selecting a lookahead space. Soar selects a state for the lookahead space that is a copy of the supergoal state where the impasse was detected. Soar then applies each of the contentious operators in the lookahead space in turn. If there is some long-term knowledge that can act on the representations obtained by the application of these operators in the lookahead space to generate discriminating preferences that enable Soar to make a selection among the contending operators in the supergoal problem space then Soar transits back into the supergoal space and commits to the suggested choice. Additionally, Soar creates a chunk that will allow it to make this very choice in subsequent problem solving.

13. Nominal behavior: consists of a cycle in which working memory is matched to productions in long-term memory. These matched productions fire in parallel adding preferences to preference memory. Changes to state objects are made once all productions have been fired. Two types of memory cues can be produced: (applicable) preferences for operators and state objects. These cues can cause further long-term productions to fire. The process of firing long-term productions and changing working memory continues until no further productions match the working memory. Now, Soar processes all preferences for problem spaces, states, and operators. For a given goal, Soar first commits to a choice for a problem space, then to state for the problem space, and finally to an operator for the problem space. Finally, a selected operator is applied. The resulting state changes will produce new preferences to be retrieved in the next decision cycle. These preferences are again processed to obtain choices for problem spaces, states, and operators.

3.2 The overall agent organization

The agent is implemented as a blackboard system [41, 24, 58, 56, 57]. Soar's initial default problem space, top-ps, is given an operator, task, whose application produces a transition into the task problem space. The task operator is meant to stand for the whole enterprise of using a software system (here Mathematica). The user introduces an input task representation into the

task problem space. The state for the task problem space is called the blackboard. It is shown embedded within the task problem space in Figure 3.2. The blackboard is a global state in which are present representations of the knowledge that the agent can access either delivered at the start of problem solving or during the course of problem solving. This knowledge is retrieved from Soar's long-term memory. The blackboard also contains the problem solving representations that are built by the application of operators. There are seven problem spaces: supervisory, formulation, interpretation, external memory, knowledge acquisition, input and output making up the system. In this chapter all except the knowledge acquisition space will be described. The consideration of this remaining space is the content of Chapter 4. Soar transits to and from the task space and these seven spaces. For each operator of a particular space there is an operator proposal rule in Soar's long-term memory that adds a preference for a process operator in the task space when the blackboard state matches the rule conditions. During a given decision cycle, multiple process operators can be applicable in the task space. The various possibilities include: a. a single process operator is applicable: in this case there is no contention b. multiple operators of a single process are applicable and c. operators belonging to multiple problem spaces are applicable. Soar's control knowledge consisting of long-term productions may deliver preferences that resolve the various contentions. Alternatively, Soar can transit into the lookahead space in order to resolve the impasse. Once Soar has selected a process operator in the task space it applies the selected operator. The operator implementation knowledge is not a part of the task space and therefore, Soar creates a subgoal to resolve the impasse. Soar selects the problem space to which the process operator belongs and selects a new (initially empty) state for this problem space. For every process operator in the task space there are one or more operators that implement that operator in the corresponding process problem space. Soar selects and applies these operators in the process problem state. Operator applications in the process problem spaces create new representations or modify existing representations on the blackboard. Thus, operator activity in a process space is visible to all other process operators through the blackboard. Once a process operator has been implemented in the process problem space Soar transits back to the task space and the cycle of selecting among applicable process operators in the task space and applying the selected process operator in the process space continues until the time that Soar has obtained a result

for the input task on the blackboard. At the outset, a bidirectional data channel is established between Soar and Mathematica using a client-server arrangement. Mathematica remains invoked throughout the interaction with it by Soar analogous to the situation that prevails when a person uses Mathematica directly. Data to be transmitted to Mathematica is added to the top-space by an output process operator from where it is automatically sent to Mathematica by Soar's output process. Data produced by Mathematica is received by Soar's input process and added to the top-space. An input process operator adds the data to the blackboard for further processing by the agent.

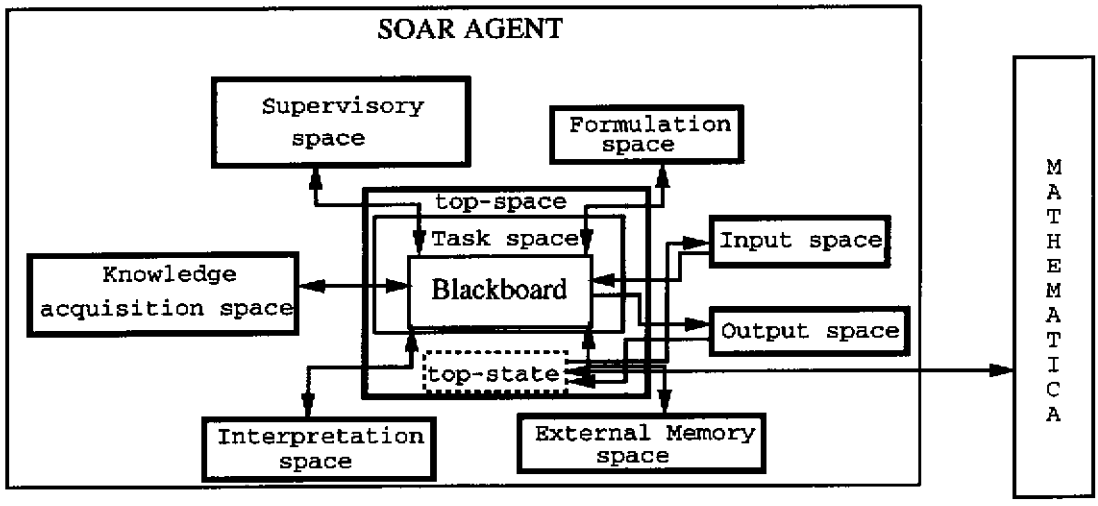


Figure 3.2: The agent's problem space organization.

3.3 Representations

In this section the various representations manipulated by the agent's processes are described. These include the input task representation, the representation of Mathematica functionality, the representation of mathematical knowledge, and the problem solving representations.

3.3.1 Input task representation

An input task is represented as a goal object that is added manually to the blackboard. A goal object has these attributes:

1. **type**, whose value is *goal.*

2. **purpose**, whose value is a function application.

3. **data**, whose value is a set of data objects.

4. **facts**, whose value is a set of predicates denoting relations among the data objects or among names of data objects.

A data object has these attributes:

1. **type**, whose value is *data.*

2. **description**, whose value is a set of predicates describing the data.

3. **child**, whose value is a set of data objects.

4. **parent**, whose value is a set of data objects.

An example of an input task representation from trajectory planning [19] is illustrated in Figure 3.3. As shown in the figure, o1 is the input task object. Its purpose is to compute the coefficients, a_0, a_1, a_2, and a_3, of an interpolating polynomial, $y = a_0 + a_1 log(x^2) + a_2 x^2 + a_3 x^3$. The interpolating polynomial is represented by the data object, o2. The constraints on the interpolating polynomial are given by the data object, o7. Each of these constraints is represented as an implication. *value* is a predicate whose second argument represents the value of its first argument. For example, the first implication of o7 asserts that the value of the dependent variable is y_0, if the value of the independent variable is x_0. *derivative* is a function acting on two arguments. The first argument is the object to be differentiated and the second argument is the variable of differentiation.

object:
id: o1
type: goal
purpose: (interpolation (o2 o7))
data: o2 o3 o4 o5 o6 o7)

object:
id: o2
type: data
description: (equation) (independent-variable o5)
(dependent-variable o6)
(name ($y = a_0 + a_1 log(x^2) + a_2 x^2 + a_3 x^3$))
(interpolating-polynomial)
child: o3 o4

object:
id: o3
type: data
parent: o2
description: (lhs) (name (y))

object:
id: o4
type: data
parent: o2
description: (rhs)
(name ($a_0 + a_1 log(x^2) + a_2 x^2 + a_3 x^3$))

Figure 3.3: The input task representation of a trajectory planning task.

```
object:
        id: o5
        type: data
        description: (independent-variable)
                     (name (x))


object:
        id: o6
        type: data
        description: (dependent-variable)
                     (name (y))


object:
        id: o7
        type: data
        description: (function-values)
                     (if (value independent-variable x0)
                        (value dependent-variable y0))
                     (if (value independent-variable x1)
                        (value dependent-variable y1))
                     (if (value independent-variable x0)
                        (value (derivative (dependent-variable
                                            independent-variable)) 0))
                     (if (value independent-variable x1)
                        (value (derivative (dependent-variable
                                            independent-variable)) 0))
```

Figure 3.3 continued.

3.3.2 Representation of Mathematica functionality

In this subsection the representation of Mathematica knowledge is described. Currently, a subset of Mathematica functions for calculus, equation solving, and algebraic manipulation have been represented. There are two representations for a Mathematica function. The first is a declarative representation of the function using an object of type, *mma-function*. The attributes of a mma-function object specify the type of arguments of the function, the type of the output produced by the function, the mapping performed by the function, and the syntax of the function. The mma-function objects are a part of the blackboard representations. Two examples of mma-function objects are:

`Equal`:

object:
id: o14
type: mma-function
name: (λ x λ y Equal x y)
output: boolean
action:
(if (equal x y) (output true))
(if (not (equal x y)) (output false))
(if (output false) (not (equal x y)))
(if (output true) (equal x y))

`D` with `NonConstants`:

object:
id: o21
type: mma-function
name: ($\lambda x \lambda y$ D x y)
output: (derivative O1 O2)
input: (exists (O1 O2 O3) (and (child O1 O3) (dependent O3 O2)))
action:
(if (exists (O1 O2 O3) (and (child O1 O3) (dependent O3 O2)))
(derivative O1 O2))

The second representation of a Mathematica function is procedural. It uses a predicate, *computes*, to relate a description of a task to a Mathematica

function can be used to compute the task. These predicates are a part of the blackboard representations. Two examples of procedural representations of Mathematica knowledge are:

- (computes Mma derivative `D`)
- (computes Mma equal `SameQ`)

The first predicate asserts that Mathematica computes a derivative with the `D` function while the second asserts that equality is computed with the `SameQ` function. There are two advantages of having both declarative and procedural representation for Mathematica functions. In cases where procedural knowledge linking a task to a function is unavailable, the agent can reason with the declarative representation of the Mathematica functions to try and find plausible functions to use. Here, the declarative representation is a means of avoiding the brittleness associated with procedural knowledge alone. The second occasion for the use of declarative knowledge arises when multiple linkups between a task and Mathematica functions are suggested by the procedural knowledge. In such cases, by reasoning with the properties of the functions, contained in their declarative representations, an informed selection may be possible.

It is useful to contrast the representations of Mathematica knowledge in the agent with the representation of Unix knowledge used by Wilensky et. al [74] and Selfridge [66]. In the UC system [74] knowledge of UNIX commands includes a representation of the use of a command and a representation of the intended function for which the command is appropriate. These correspond to our procedural representation of function knowledge. The UC system does not include a declarative representation of the function knowledge. This limits the possibility for introducing new linkups between tasks and functions by reasoning with the knowledge of the functions. Selfridge [66] represents a UNIX command as a frame whose name is the name of the function. Our representations differ in that the object corresponding to a Mathematica function is not referred to by its name. Instead, the name of the function is an attribute of the command. The implication is that functions can be retrieved without knowledge of their name. In Selfridge's representation, the name of the function must be known in order to retrieve it from memory.

3.3.3 Representation of mathematical knowledge

1. **Domain-independent mathematical knowledge.** For example, *To compare two objects one may test if they are equal* is represented as:

```
(if (and
        object(id: O type: calculation-graph-node
                description: (compare *)))
    (add
        object(id: O description: (equal *))))
```

2. **Declarative representation of task knowledge.** For example, $S_K^T = \frac{\partial T}{\partial K}\frac{K}{T}$ is represented as:

```
(definition (S T K)
        (calculation
            (product
                (derivative transfer-function open-loop-gain)
                (div open-loop-gain transfer-function))))
```

3. **Procedural representation of task knowledge.** For example,

   If there is an internal action, (coordinates o1), then set up the internal action, (equal r1 r2), where r1 and r2 are points whose separation is known.

4. **Interpretation of mathematical data.** These mathematical objects are recognized:

   (a) Equation: If the Mathematica expression consists of Equal as the root function then it is recognized as an equation. This recognition is context-independent. Let us consider the case where two objects have been equated using the Equal function. Here the resulting equation will have as its left-hand and right-hand constituents recognizable objects. This recognition is context-dependent for the objects are recognized as being present in the current problem solving context.

(b) Sum: If the Mathematica expression consists of Plus as the root function then it is recognized as a sum of terms.

(c) Product: If the Mathematica expression consists of Times as the root function then it is recognized as a product of terms.

(d) Derivative: If the Mathematica expression consists of D as the root function then it is recognized as a derivative.

Further, if the Mathematica expression consists of D as the root function together with the expression, (Rule NonConstants (List y)), then it is interpreted as a derivative with y depending upon the variable of differentiation.

(D y x (Rule NonConstants (List y))) is interpreted as (and (derivative y x) (dependent y x)). Here y and x are names of some objects whose identity is established by matching names.

5. Mathematical facts are represented declaratively. For example, (number 1), is a fact about the symbol, *1*.

3.3.4 Problem solving representations

The blackboard contains these representations:

1. Objects: An object is a set of attribute value pairs. We have already seen the objects for the input task representation and Mathematica functionality. Additionally, there are objects for internal actions, external actions, and results. An internal action is a function application and has these attributes:

   (a) description, whose value is a set of S-expressions describing the internal action. These S-expressions can either be function applications or predicates.

   (b) parent, whose value is one or more internal actions.

   (c) child, whose value is one or more internal actions.

   (d) joins, whose value is one or more internal actions.

   (e) decomposition-of, whose value is one or more internal actions.

   (f) type, whose value is *calculation-graph-node*.

An external action is a function application executable by the software system used by the agent and has these attributes:

(a) type, whose value is *external-action.*

(b) description, whose value is the predicate, *name*, whose argument is the name of the external action.

(c) calculation, whose value is an internal action.

A result is the representation of data received by the agent when an external action is executed by the software system and has these attributes:

(a) ss-output, whose value is the data received from the software system.

(b) mma-input, whose value is an external action.

(c) type, whose value is *result.*

(d) description, whose value is a set of predicates describing the data received from the software system.

2. Facts: These are predicates. We have already seen the mathematical facts and Mathematica facts that might be present in the global state. Facts also represent relations among data objects. For example, (perpendicular o3 o5 o2), where o3, o5, and o2, are points, represents that the segment joining o3 and o5 is perpendicular to the segment joining o2 and o5.

3. calculation-graph-root: is an attribute of the global state whose value is an internal action that has an additional attribute, *input-task*, whose value is a goal object. The root node of the calculation graph can therefore locate the input task representation.

4. calculation-graph-focus-of-attention: is an attribute of the global state whose value is an internal action that the agent is currently reasoning about either in the supervisory space or in the formulation space.

5. calculation-graph-frontier: is an attribute of the global state whose value is a set of internal actions. These are internal actions that

have yet to be processed and from among which a selection is made for the calculation-graph-focus-of-attention. When an item enters the calculation-graph-focus-of-attention it ceases to be part of the calculation-graph-frontier.

6. The internal actions, external actions, and results collectively make up a *calculation graph* that is the central problem solving data structure. The internal action nodes of the calculation graph represent the task plan and the external action nodes of the calculation graph represent the software system plan. The calculation graph is built up both immediately prior to action and in anticipation of action. For example, task knowledge may suggest an internal action specification that is immediately followed with an external action specification using the knowledge of the software system functionality. This is the case of a lock-step plan and execution episode. On the other hand multiple internal actions might be specified at some point and followed up with external actions only in order. The internal actions that await their turn to be processed are an example of an anticipatory plan of action. Some useful properties of the calculation-graph are:

   (a) Links among internal actions can be traversed to influence the status of planned internal actions. For example, suppose at some point a set of internal actions are planned to succeed an internal action. The decision to continue on with these planned actions is could be based upon the outcome for the current action.

   (b) Links representing ordering among actions act as constraints on what internal actions may be selected to be in-focus. For example, an internal action that is preceded by another cannot be in-focus until the preceding action has been completed.

3.4 Agent's process operators

In this section the agent's process operators are described. These operators are proposed in the task spaces and implemented in the corresponding process problem spaces. To represent a process operator we need an operator proposal rule in the task space (that makes process operator applicable in

the task space), an operator proposal rule in the process problem space (after Soar has transited from the task space into the process space), operator implementation rules to implement the operator in the process space. Additionally, control rules (that order multiple applicable process operators with respect to each other in the task space) are needed. The Soar rules for the process operator, sp-10, are illustrated in Figure 3.4. We have labeled rule conditions and actions with numbers in parentheses to refer to them in the ensuing discussion. These labels are not part of the Soar code.

Let us consider the operator proposal rule, op-proposal-sp-10. The goal-context required by the rule is given by conditions 1, 2, and 3. The goal-context must be the task problem space whose state must be the blackboard. Two blackboard objects must be present. Condition 4 checks for an object linked to the blackboard by focus-of-attention and for a fact. Conditions 5 and 6 bind the variable, function, to the function application contained in the object's description. Conditions 7, 8, and 9 require that the fact be about the computation of the function bound in condition 6. The variable, mma-function, is bound to the Mathematica function that is suggested by the fact. Conditions 11, 12, and 13 specify that the proposal rule is not matched if the blackboard contains the fact, (sp-10 o). If the blackboard satisfies the conditions of op-proposal-sp-10, Soar will fire the rule and add an acceptable preference for the process operator, sp-10. The name of the process operator is the problem space that will implement it. So, for sp-10, the name is supervisory-process since that is the space to implement sp-10. If Soar selects sp-10 to be the operator for the task space then it will detect an impasse since there is no implementation for sp-10 in the task space. Soar responds by creating a subgoal to resolve the impasse. At this point, the three rules shown in Figure 3.6, enable Soar to transit into the appropriate process space. The first rule in Figure 3.6 makes the name of the subgoal the same as the name of the process operator whose application produced the impasse. Then, the second rule in Figure 3.6 can fire to add a preference for the supervisory problem space for the subgoal. After Soar selects the process space for the subgoal, the third rule in Figure 3.6 fires to add a preference for a new state for the process space that is linked to the blackboard. The subgoal state is initially empty although linked to the blackboard. The subgoal context can now match conditions 14, 15, and 16 of the second operator proposal rule, op-proposal-sp-10-1, shown in Figure 3.4. This rule adds an acceptable preferences for the operator, op-implement-sp-

10-1, in the subgoal. Conditions 18, 19, 20, 21, 22, 23, and 24 of op-proposal-sp-10-1 are similar to the conditions 5, 6, 7, 8, 9, and 10 of op-proposal-sp-10. The operator, op-implement-sp-10-1 is implemented by the third rule in Figure 3.4. The op-implement-sp-10-1 rule adds to the blackboard the fact, (sp-10 o), that causes Soar to withdraw op-proposal-sp-10. Another fact, (terminate sop), where sop is the symbol denoting the process operator in the task space that started off this whose processing, is added to the subgoal state. The addition of this fact enables the rule, 1-terminate-superop, shown in Figure 3.6 to fire and add a reconsider preference for sop. Since, the acceptable preference for sop has been withdrawn at the end of the current decision cycle Soar removes sop in the task space. However, op-implement-sp-10-1 adds a third fact, (formulate sop), to the status of the object in the focus-of-attention of the blackboard. This completes a description of how Soar makes the transition from the task space into a process space and implements a process operator in the process state to change the blackboard state. This basic processing is true for all the process operators. We do not describe process operators in terms of the Soar code any further in this document.

Instead, a more succinct description of the process operators is used. As an illustration of this alternate descriptive device consider Figure 3.7. First, there is an English paraphrase of the operator that tells us under what conditions the operator will be applied and what changes the operator will produce. There is no strict format for the English paraphrase as our intent is simply to provide the reader with a sense of what purpose the operator serves. Secondly, there is rule-like description for the operator. Comparing this rule with the rules in Figure 3.4 the reader can see the abstraction in the rule-like description. Since the conditions and the actions of the rule are in terms of the representations that Soar manipulates we see that the rule shows us all the information in the Soar rules of Figure 3.4. However, our use a rule-like description for operators is not meant to suggest that Soar actually interprets this rule directly. The mapping of the rule to Soar is demonstrated by our previous discussion of the Soar rules shown in Figure 3.4. In sum, the description offered in Figure 3.7 is simply for the ease of exposition. Some words about the notation used in Figure 3.7 are in order. The conditions are predicates true in the blackboard. All the conditions must be true for the rule to apply. Capital letters are used to denote variables that get bound to items in the blackboard. The action part of the rule consists of changes

to the blackboard. Add and delete are keywords used in the rule action to specify these modifications. In the following subsections a representative set of process operators are described to give the reader a sense of the agent implementation.

3.4.1 The supervisory operators

The supervisory operators create internal actions and build the task plan. Some representative supervisory operators are given below:

- sp-1: *If a definition is known for an input goal then initialize the calculation graph with the defined calculation as the root node and add the root node to the calculation graph frontier.*

  (if (and
  object(id: O1 type: goal purpose: (X Y))
  (definition (X Z) (calculation C)))
  (add
  object(id: O2 type: calculation-graph-node description: C)
  (calculation-graph-root O2)
  (calculation-graph-frontier O2)))

  For example, if the input goal has the purpose, (S T K), and a definition, (product (derivative transfer-function open-loop-gain) (div open-loop-gain transfer-function)), for (S T K) is known, then the calculation graph is initialized with the defined calculation.

- sp-2: *If an input goal is performable using a given Mathematica function then initialize the calculation graph with the input function application.*

  (if (and
  object(id: O1 type: goal purpose: (X Y))
  fact(computes mma X Z))
  (add
  object(id: O2 type: calculation-graph-node description: (X Y))
  (calculation-graph-root O2)
  (calculation-graph-frontier O2)))

```
(sp op-proposal-sp-10
   (goal <g> ^name task ^problem-space <p> ^state <s>) (1)
   (problem-space <p> ^name task) (2)
   (state <s> ^name blackboard) (3)
   (state <s> ^fact <f> ^focus-of-attention <o>) (4)
   (object <o> ^description <d>) (5)
   (sexp <d> ^value <function>) (6)
   (sexp <f> ^value computes ^next <m1>) (7)
   (sexp <m1> ^value mma ^next <m2>) (8)
   (sexp <m2> ^next <m3> ^value <function>) (9)
   (sexp <m3> ^next nil ^value <mma-function>) (10)
 -{(state <s> ^fact <f1>) (11)
   (sexp <f1> ^value sp-10 ^next <m4>) (12)
   (sexp <m4> ^next nil ^value <o>)} (13)
 -->
   (operator <op> ^name supervisory-process ^input-object <o>
                  ^entry sp-10)
   (goal <g> ^operator <op> +))

(sp op-proposal-sp-10-1
   (goal <g> ^problem-space <p> ^state <s> ^object <sg>) (14)
   (problem-space <p> ^name supervisory-process) (15)
   (goal <sg> ^state <ss> ^operator <so>) (16)
   (operator <so> ^entry sp-10 ^input-object <o>) (17)
   (object <o> ^description <d>) (18)
   (sexp <d> ^value <function>) (19)
   (state <ss> ^fact <f>) (20)
   (sexp <f> ^value computes ^next <m1>) (21)
   (sexp <m1> ^value mma ^next <m2>) (22)
   (sexp <m2> ^next <m3> ^value <function>) (23)
   (sexp <m3> ^next nil ^value <mma-function>) (24)
 -->
   (operator <op> ^name op-implement-sp-10-1 ^input-object <o>
                  ^calculation <o>)
   (goal <g> ^operator <op> +))
```

Figure 3.4: An example of Soar rules for a supervisory operator.

```
(sp op-implement-sp-10-1
   (goal <g> ^state <s> ^operator <op> ^object <sg>) (25)
   (operator <op> ^name op-implement-sp-10-1 ^calculation <o>) (26)
   (goal <sg> ^state <ss> ^operator <so>) (27)
   (operator <so> ^entry <entry>) (28)
 -->
   (state <ss> ^fact <f> + &)
   (sexp <f> ^value <entry> ^next <n>)
   (sexp <n> ^value <o> ^next nil)
   (state <s> ^fact <f2> + &)
   (sexp <f2> ^value terminate ^next <n1>)
   (sexp <n1> ^value <so> ^next nil)
   (object <o> ^status <f3> + &)
   (sexp <f3> ^value formulate ^next nil))
```

Figure 3.5: Figure 3.4 continued.

For example, if the input goal is, (factorization o1), where o1 is a polynomial object, then knowing that the Mathematica function *Factor* factorizes expressions, the calculation graph is initialized with an internal action whose description is (factorization o1).

- sp-3: *If an input goal is known to be a computable function then initialize the calculation graph with the input function application.*

  (if (and
  object(id: O1 type: goal purpose: (X Y))
  (computable-function X)
  (add
  object(id: O2 type: calculation-graph-node description: (X Y))
  (calculation-graph-root O2)
  (calculation-graph-frontier O2)

  For example, if the input goal is, (coordinates o1), where o1 is a point object, and it is known that coordinates is a computable function then initialize the calculation graph with an internal action whose description is (coordinates o1).

```
(sp 1-goal-elaboration-task
   (goal <g> ^impasse no-change ^attribute operator ^object <sg>)
   (goal <sg> ^operator <so>)
   (operator <so> ^name <x>)
 -->
   (goal <g> ^name <x> +))

(sp 1-propose-space-task
   (goal <g> ^impasse no-change ^attribute operator ^object <sg> ^name <x>)
   (goal <sg> ^operator <so>)
   (operator <so> ^name <x>)
 -->
   (goal <g> ^problem-space <p> +)
   (problem-space <p> ^name <x> +))

(sp 1-propose-state-task
   (goal <g> ^impasse no-change ^attribute operator ^object <sg> <sg>
             ^name <x> ^problem-space <p>)
   (goal <sg> ^state <ss>)
   (problem-space <p> ^name <x>)
 -->
   (goal <g> ^state <s> +)
   (state <s> ^superstate <ss> +))

(sp 1-terminate-superop
   (goal <g> ^name <x> ^problem-space <p> ^object <sg> ^state <s>
             ^operator <op>)
   (problem-space <p> ^name <x>)
   (goal <sg> ^operator <so>)
   (operator <so> ^name <x>)
   (state <s> ^fact <f>)
   (sexp <f> ^value terminate ^next <n1>)
   (sexp <n1> ^value <so>)
 -->
   (goal <sg> ^operator <so> @))
```

Figure 3.6: The Soar rules that enable transition between the task space and the process spaces.

If a calculation in the calculation-graph-frontier is such that the function to be computed is directly available as a Mathematica built-in function then change the status of the calculation to formulate.

```
(if (and
        (calculation-graph-focus-of-attention O1)
        (description O1 (Y *))
        (computes mma Y Z))
   (add
       (status O1 (formulate))))
```

Figure 3.7: A succinct description for the rules shown in Figure 3.4.

- sp-4: *If a definition is known for an internal action then add the definition to the description of the internal action.*

```
(if (and
        object(id: O1 type: calculation-graph-node description: (X Y))
        (definition (X Y) (calculation C)))
   (add
        object(id: O1 description: C)))
```

For example, if a definition, (equal o1 o2), is known for (compare o1 o2), for an internal action, then (compare o1 o2) is added to the internal action's description.

- sp-6: *If a calculation is compound then split the calculation into subcalculations corresponding to each of the compound terms of the calculation. Then add another calculation that applies the top-level function to the results of the subcalculations. If there are n compound terms then* $n + 1$ *subcalculations are produced.*

```
(if (and
        object(id: O1 description: (F1 (A1 A2)))
        fact(compound A1)
        fact(compound A2))
    (add
        object(id: O2 description: A1)
        object(id: O2 description: A2)
        object(id: O2
                description: (F1 (result A1) (result A2)))))
```

- sp-8: *If there is a calculation in the calculation-graph-frontier then consider focusing on this calculation.*

```
(if (or
        (and
            (calculation X)
            (element X calculation-graph-frontier)
            (forall (Y)
                    (and
                        (predecessor X Y)
                        (not (element Y calculation-graph-frontier))
                        (status Y (verified Y Z)) (result Y Z))))
        (and
            (calculation X)
            (element X calculation-graph-frontier)
            (not (exists (Y)
                    (predecessor X Z)))))
    (add
        (calculation-graph-focus-of-attention X))
    (delete
        (calculation-graph-frontier X)))
```

- sp-9: *If a result is available for a calculation then verify that result.*

```
(if (and
        object(id: C type: calculation-graph-node mma-input: O1)
        (mma-output O1 O2)
        (result O2 O3))
    (add
        (verify O3)))
```

- sp-10: has already been described in Section 3.4.

- sp-16: *If there is a calculation being attended to then try to formulate the calculation.*

```
(if (and
        (calculation-graph-focus-of-attention X))
    (add
        (status X (formulate))))
```

- *Propagate status of a calculation to its parent.*

```
(if (and
        (calculation c)
        (mma-input c o1)
        (mma-output o1 o2)
        (result o2 o3)
        (not (next-object c *))
        (parent-object c c1))
    (result c1 o3))
```

- sp-14: *If a result is obtained for an external action then create an internal action to simplify the data obtained and splice this internal action to precede any internal actions that may have otherwise succeeded the first internal action.*

```
(if (and
        object(id: O1 result: O2 next-object: O3))
```

```
(add
    object(id: O4 type: calculation-graph-node
           description: (simplify (result O1))
           next-object: O3
           post-condition: (compare (result O1) (result O4)))
    object(id: O1 next-object: O4)))
```

- sp-15: *If result for an internal action is available and there is a post-condition calculation for this internal action then set up the post-condition calculation.*

```
(if (and
        object(id: C post-condition: (F X) result: O1))
    (add
        object(id: O2 type: calculation-graph-node
               description: (F X)
               previous-object: C)))
```

- sp-17: If the internal action in focus has been refined into other internal actions then remove the internal action from focus.

```
(if (and
        (calculation-graph-focus-of-attention O1)
        object(id: O1 type: calculation-graph-node)
        fact(refined O1))
    (delete
        (calculation-graph-focus-of-attention O1)))
```

- sp-18: If the internal action in focus has been decomposed into other internal actions then remove internal action from focus.

```
(if (and
        (calculation-graph-focus-of-attention O1)
        object(id: O1 type: calculation-graph-node decomposition: O2))
    (delete
        (calculation-graph-focus-of-attention O1)))
```

Notation:

- (next-object c o4) means that the old value of (next-object c *) is overwritten.

3.4.2 The formulation operators

The formulation operators build the software system plan. Some representative formulation operators are given below:

- fp-1: *If there is a Mathematica function that computes an internal action then use that function to specify an external action for the internal action.*

```
(if (and
        object(id: O1 type: calculation-graph-node
                       status: formulate
                       description: (F X))
        fact(computes mma (F Y) MF))
   (add
       fact(use O1 MF)
       object(id: O1 status: {(use O1 MF), (formulated O1)})))
```

fp-1 matches an internal action description to an external function description. An internal action description is a ground formula. An external function description is a quantified formula. To match an internal action description to an external function description means to obtain an instance of the latter that is syntactically the same as the former. Another example of a fp-1 rule is the following:

```
(if (and
        object(id: O1 type: calculation-graph-node
                       status: formulate
                       description: {(X1 X2 X3), (X4, X3)})
        fact(computes mma (and (X1 X2 X3) (X4 X3)) MF))
   (add
       fact(use O1 MF)
       object(id: O1 status: {(use O1 MF), (formulated O1)})))
```

It differs from the first in the syntactic match achieved. An example of the use of this rule occurs when the internal action description, {(derivative o1 o2), (dependent o3 o2)} is matched to (computes mma (and (derivative X1 X2) (dependent X3 X2))).

- fp-2: *Determine the arguments for the selected Mathematica function by interpreting the task specification.*

```
(if (and
        object(id: O1 type: calculation-graph-node
                       description: (F (X1 X2))
                       status: (use O1 MF))
        (calculation-graph-root O2)
        object(id: O2 type: calculation-graph-node
                       task-object: O3)
        object(id: O3 type: goal data: D1 D2)
        object(id: D1 type: data description: {X1, (name (N1))})
        object(id: D2 type: data description: {X2, (name (N2))}))
    (add
        fact(arguments O1 MF N1 N2)
        object(id: O1 status: (arguments O1 MF N1 N2))))
```

- fp-3: *If a Mathematica function and its arguments are available then obtain the external action syntax.* Each Mathematica function syntax is represented as a lambda expression that is applied to the names of arguments to obtain the external action syntax. For example, if the argument to Factor is, $a^2 + b^2 + 2ab$, then the external action syntax, $Factor[a^2 + b^2 + 2ab]$, is obtained by applying, $\lambda x Factor[x]$ to $a^2 + b^2 + 2ab$.

3.4.3 The output process operators

An example of an output process operator is given below:

- out-1: *If an external action is available then it is added to the top-state for transmission to Mathematica.*

```
(if (and
        object(id: O1 type: mma-input description: (name (N))))
    (add
        (top-state (output N))))
```

3.4.4 The input process operators

An example of an input process operator is given below:

- in-1: *If an external action has been transmitted to Mathematica and data from the execution of this action by Mathematica has been received then add the data to the blackboard.*

```
(if (and
        (transmission-complete O1)
        (top-state (input N)))
    (add
        object(id: O2 type: mma-output description: (name (N)))))
```

3.4.5 The interpretation operators

A representative set of interpretation operators are given below:

- ip-1: *If data is received from Mathematica then create a result object and build meaning representations for the data.*

```
(if (and
        object(id: O1 type: mma-output
                    description: (name (Plus X))))
    (add
        object(id: O2 type: result mma-output: O1
                    description: (sum))))


(if (and
        object(id: O1 type: mma-output
                    description: (name (Times X))))
```

```
   (add
        object(id: O2 type: result mma-output: O1
                       description: (product))))


(if (and
         object(id: O1 type: mma-output
                        description: (name (Equal X Y))))
   (add
        object(id: O2 type: result mma-output: O2
                       description: (equation) child: O3 O4)
        object(id: O3 type: data result parent: O2
                                        description: (lhs) (name (X)))
        object(id: O4 type: data result parent: O2
                                        description: (rhs) (name (Y)))))

(if (and
         object(id: O1 type: mma-output
                        description: (name (0-9))))
   (add
        object(id: O2 type: result mma-output: O1
                       description: (number) (constant))))


(if (and
         object(id: O1 type: mma-output
                        description: (name (a-z))))
   (add
        object(id: O2 type: result mma-output: O1
                       description: (constant))))


(if (and
         object(id: O1 type: mma-output
                        description: (name (Plus X))))
   (add
```

```
        object(id: O2 type: result mma-output: O1
                       description: (polynomial))))

(if (and
        object(id: O1 type: mma-output
                       description: (name (False))))
    (add
        object(id: O2 type: result mma-output: O1
                       description: (false))))

(if (and
        object(id: O1 type: mma-output
                       description: (name (True))))
    (add
        object(id: O2 type: result mma-output: O1
                       description: (true))))
```

- ip-2: *If there is an imperative to verify a result and no expectation failure is detected then assume that the result is verified.*

```
(if (and
        object(id: O1 type: calculation-graph-node result: O2)
        fact(verify O2)
        (not object(id: O2 description: (expectation-failure O2))))
    (and
        fact(verified O1 O2)))
```

- ip-3: *If there is an imperative to verify a result and an expectation failure is detected then add to the result's description that there is an expectation variance.*

```
(if (and
        fact(verify O1)
```

```
        object(id: C type: calculation-graph-node result: O1
               expectation: D)
        (not object(id: O1 type: result description: D)))
    (add
        object(id: O1 description: (expectation-failure))))
```

- ip-4: *If there is an imperative to verify a result and a match between the expectation and the result is detected then add to the result's description that there is an expectation match.*

```
(if (and
        fact(verify O1)
        object(id: C type: calculation-graph-node result: O1
               expectation: D)
        object(id: O1 type: result description: D))
    (add
        object(id: O1 description: (expectation-match))))
```

- ip-5: *If an internal action for an external action is to apply a function to the result obtained for another internal action then the result for the second internal action is annotated with a relation, output, between the data for the second external action and data of the first external action.*

```
(if (and
        object(id: C mma-input: O1 description: (F (result C1)))
        object(id: O1 type: mma-input mma-output: O2)
        object(id: O2 type: mma-output result: O3)
        object(id: C1 mma-input: O1')
        object(id: O1' type: mma-input mma-output: O2'))
    (add
        object(id: O3 description: (output (F O2') O3))))
```

3.4.6 The external memory operators

A representative set of external memory operators are given below:

- em-1: *If a calculation requires the data of a previously completed calculation then retrieve the positional reference to the data stored with the result of that calculation.*

```
(if (and
        fact(argument F (result C))
        object(id: C result: O1 description: (index I)))
    (and
        fact(positional-reference C I)))
```

- em-2: *If an external action produces a result for an internal action then add the positional reference of the result to the internal action's description.*

```
(if (and
        object(id: C result: O1)
        object(id: O1 mma-output: O2)
        fact(counter V))
    (add
        object(id: C description: (index V))
        fact(counter (increment (V)))))
```

- em-3: *If an internal action's result is computed by another internal action, then update the positional reference for the first internal action to be the positional reference for the second internal action.*

```
(if (and
        object(id: C1 result: (result C2)
                description: (index I1))
        object(id: C2 description: (index I2)))
    (add
        object(id: C1 description: (index I2)))
    (delete
        object(id: C1 description: (index I1))))
```

3.5 The control knowledge

The control among operators belonging to different processes and among multiple operators of the same process is provided by control rules in the task space. The execution trace of the Soar agent is a sequence of process operators. After the execution of an operator control comes back to the task space prior to the execution of another operator. Processing in the various problem spaces is coordinated through a shared state, called a blackboard, as illustrated in Figure 3.2. The blackboard is a collection of objects and facts and initialized by the input task representation. Process operators modify existing blackboard objects or add or delete new blackboard objects and facts. The basic control cycle consists of selecting among applicable operators and applying the selected operator until the input function is computed. The choice among applicable operators is made either by explicitly represented knowledge or through lookahead search. In the latter case, knowledge is acquired to govern similar decision-making in subsequent problem solving by the agent. A representative set of control rules are given below:

1. *If one operator is more specific than another then prefer it over the other and withdraw the more general operator.* For example, sp-1 is more specific than sp-2 and is selected preferentially. Similarly, between sp-10 and sp-16, sp-10 is selected as it is more specific than sp-16.

2. *If a supervisory operator and sp-10 are both applicable to the same internal action then select the former.* This heuristic encodes the preference for a move using task knowledge over a move using Mathematica knowledge.

3. *If two applicable operators cannot be ordered then apply them in turn in a lookahead space. Prefer an operator that can be applied to one that cannot.* For example, the choice between fp-3 and out-1 is resolved in this manner.

4. *If two applicable operators cannot be ordered then apply them in turn in a lookahead space. If both operators can be applied then make them indifferent with respect to each other.* For example, the choice between sp-15 and sp-9 is resolved in this manner.

5. *If two applicable operators have the same intentionality then reject one.* Operator intentionality is defined as the mapping that the operator application will perform. For example, sp-10, will apply to an internal action if there are multiple ways for a Mathematica function to be associated to the internal action.

6. *If an interpretation operator and a supervisory operator are in contention then select the former over the latter.* This heuristic suggests that interpretation may cause a modification in the intended action represented by the supervisory operator and therefore should be completed first.

7. *Withdraw an operator that applies to objects produced by itself.* For example, if a supervisory operator is proposed in order to apply a function to the result of a calculation such that the calculation itself had caused the application of the same function to some data then retract the proposal of the supervisory operator.

8. *If direct knowledge associating a calculation to a Mathematica function is available then decide to reject acting upon the intention to formulate this calculation irrespective of the availability of direct knowledge of such an association.* This ensures that the weaker intention is suppressed in the face of a stronger intention based upon more knowledge.

9. *Suppose there are two contentious operators such that the intention of the first is to reject the second. Then prefer the first operator to the second operator.*

10. *If an operator intends to remove an internal action from the focus and another operator intends to specify an external action for this internal action then prefer the first operator over the second.*

11. *If an operator intends to add a property to an internal action that is inconsistent with an existing property of the internal action then this operator is rejected.*

Chapter 4

Automatic knowledge acquisition

A multiplicity of knowledge is acquired by a human user of Mathematica. In this chapter we describe how the agent acquires some of the same knowledge. There are two forms of automatic knowledge acquisition. One is termed deliberate and the other reflective. Deliberate knowledge acquisition occurs when some new knowledge is generated and stored for use in subsequent problem solving. Reflective knowledge is acquired simply by the act of problem solving. Typically, reflective acquisition has the effect of making the processing more efficient and deliberate acquisition adds to the knowledge content of the agent.

4.1 Deliberate knowledge acquisition

In this section several types of deliberatively acquired knowledge are described. These include the acquisition of know-how rules, acquisition of control knowledge by doing, acquisition of strategies, acquisition of semantics of software system functions, and acquisition of control rules for strategy rules.

4.1.1 Acquisition of know-how rules

Internal actions are created by supervisory operators. The internal action description is a function application, that is, a function applied to some data

objects. For example, (derivative (o1 o2)), applies the function, derivative, to the data objects, o1 and o2. Supervisory operators also determine when to initiate planning for an external action specification for an internal action. An external action is also a function application, executable by the software system. For example, `D[y == a x + b, x]`, is an external action specification for Mathematica that applies the `D` function to an equation, `y == a x + b`, to differentiate it with respect to `x`. The planning for an external action is done by the formulation operators. The formulation operators take as the starting point the internal action description that was specified by the supervisory operators. The internal action description is not only a function application but can include other properties as well. For example, properties of the data to which the function is applied may be specified. So, an internal action description is a requirement for an external action specification to be obtained by the formulation process.

Sometimes, this requirement can be incomplete. Consequently, the external action specification obtained by the formulation process may not produce the desired results for the internal action. By interpreting the execution results it may be possible to determine how to modify the internal action description so that a proper external action specification may be produced by the formulation process. We call a rule that modifies an internal action description a *know-how* rule. In this section a method for the acquisition of know-how rules is described. The general schema for the acquisition of know-how rules is shown in Figure 4.1. In this figure $\triangle$ denotes the addition to *internal-action-1* suggested by *result-1*. $\square$ denotes the contingency for the acquired know-how rule. Both $\triangle$ and $\square$ depend upon the heuristic knowledge of the agent used in a given instance of the general schema. The effect of the acquired know-how rule is to modify the internal action description prior to the attempt to produce an external action specification for it.

To illustrate the know-how acquisition let us consider the processing shown in Figure 4.2. t1 is an internal action node of the calculation graph whose description is, (derivative (o1 o2)), where o1 is a polynomial equation and o2 is the variable of differentiation. t1 may be a part of some larger calculation graph that is not germane to our discussion. The formulation process plans the external action specification, f1, for t1. When f1 is executed, the data, d1, is produced by Mathematica. Interpretation operators build a result object, r1, for d1. The result object's description corresponds to the meaning of d1. It may include such information as the result is an

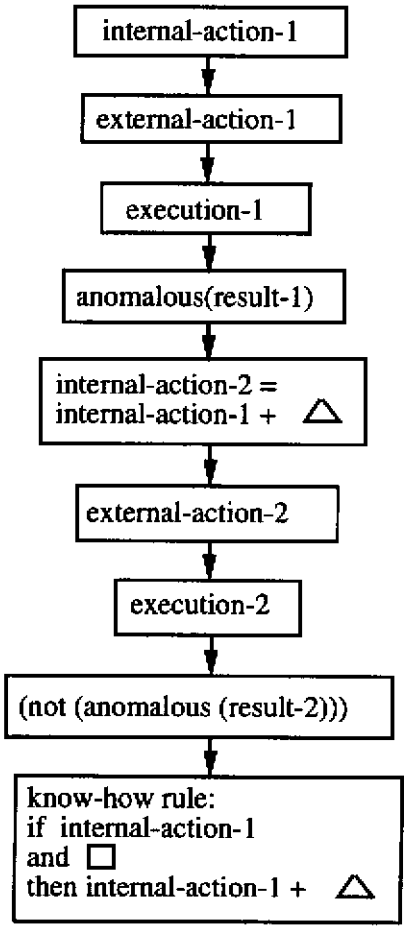


Figure 4.1: The general schema for know-how rule acquisition.

equation and that the left hand side of the equation is zero. Additionally, context-dependent interpretation rules may also apply. One such rule is to infer that if a function is applied to an equation and the result is also an equation then the left hand side of result equation is the value of the function applied to the left hand side of the input equation. In other words, the derivative of y is zero. A supervisory operator that corresponds to the rule that if the derivative of an object is zero then that object is a constant can now apply. Thus, the result, r1, has now added property that y is a constant. However, it is known that y is a variable. So we have an inconsistency with the result, r1, suggesting that y is a constant, while it is known that y is a variable. A supervisory operator recognizes this inconsistency and creates a copy of the internal action, augmenting its description with the property that was violated. In our example, the violated property is given by the two predicates, (child o1 o3) and (dependent o3 o2). The first predicate says that o3 is a part of o1. Since o1 is an equation it has two parts, the left side and the right side. o3 is the left hand side of o1. The second predicate says that o3 is a function of o2. That is, the left side of the equation is a function of the variable of differentiation. So, the copy of t1 that is created has a description that includes the function application of t1 and the two predicates that represent the violated property.

The various operators are summarized as inference rules in Figure 4.3. Rules, I1, I3, and I4, are domain-independent. Rule I2 is domain-dependent. f2 is the external action specification for t2. Since the internal action description has changed, the formulation process finds an alternative form of the D function to use. The execution result for f2, r2, does not contain the anomaly found for r1. Therefore, a know-how rule can be acquired for this example. To acquire the know-how rule, we must have a contingency and a modification to an internal action. The modification to the internal action in this case is the addition of the property that was violated in the execution result of the first internal action. That is, the property given by the predicates, (child o1 o3) and (dependent o3 o2). A knowledge-acquisition operator builds the following know-how rule for this example,

If the internal action description is, (derivative (O1 O2)), and O1 has a child object, O3, and O3 is dependent upon O2 then add to the internal action description the predicates, (child O1 O3) and (dependent O3 O2).

This operator implements the heuristic that if the modification to an internal action was a property of the data objects involved in the function

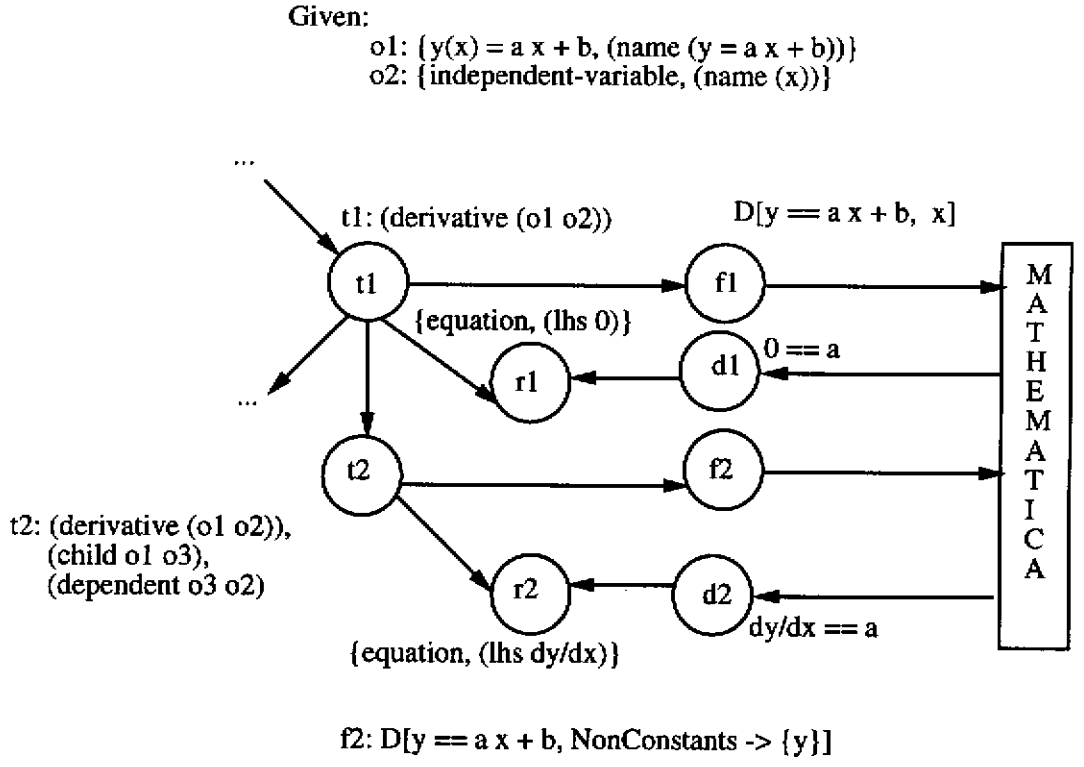


Figure 4.2: The calculation graph to illustrate know-how rule acquisition.

application of the internal action then make the contingency of the acquired rule to be the same property and make the actions of the rule to add the property to the internal action description. With the acquired know-how rule, the agent will complete internal action descriptions with the two predicates whenever the data objects of the internal action's function application have the property that is the rule's contingency. With the internal action description thus modified, the agent will avoid obtaining an anomalous execution result. The pseudo-code algorithm for the acquisition of know-how rules is given in Figure 4.4. The procedure used to add a given rule to Soar's long-term memory is provided in Appendix B.

The schema for the acquisition of know-how rules is general. However, it requires a set of heuristics that determine that an execution result is anomalous, what modification to the internal action description is sufficient to make

I1: If a function is applied to an equation and the result is also an equation then the left hand side of the result equation is the value of the function applied to the left hand side of the input equation. Similarly, the right hand side of the result equation is the value of the function applied to the right hand side of the input equation.

I2: If the derivative of an object is zero then that object is a constant.

I3: If a property, P, is asserted of an object in an execution result, and the negation of P, is known to be true of that object, then the result is anomalous. The negation of P is the property that is violated.

I4: If a property violation is obtained then create a copy of the internal action augmenting its description with the violated property.

Figure 4.3: The inference rules for the example in Section 4.1.1.

1. Execute an external action specification, ϵ, for an internal action, ι.
2. Look for inconsistency between an observed property of the execution result and an expected property.
3. If an inconsistency is found then assume that the expected property was violated by the execution of ϵ.
4. Create a copy of ι, adding to its description the violated property.
5. Retry the changed internal action.
6. If the execution result does not lead to the detection of an inconsistency then acquire a rule that adds the violated property to an internal action specification every time that internal action is created and the property is satisfied.

Figure 4.4: The algorithm for the acquisition of know-how rules.

the internal action description more complete, and on what is such modification contingent. In the above example the domain-dependent heuristic I2 is used to determine that the execution result for the first internal action is anomalous. The domain-independent heuristics, I1, I2, and I3, establish the modification to the internal action as the addition of the property that is violated in the execution result. The knowledge-acquisition operator that builds the know-how rule is also domain-independent since its application depends only on the fact that the modification should be the addition of a property to the internal action description. It must be noted that these heuristics are not limited to the particular processing instance that was discussed. However, it is also true that these heuristics are not the only ones for the general know-how rule acquisition schema. I4 is the only heuristic that we identified so far to instantiate the know-how schema. Our contribution then is the specification of a generic schema for a type of deliberate knowledge acquisition, namely, the acquisition of know-how rules, and the demonstration of the use schema for a specific instance of know-how rule acquisition.

The anomaly in the result is neither due to an error in the external action specification not attributable to Mathematica. So, it is not an execution failure. The problem is that the internal action description was incomplete. Both Shen [67] and Gil [30] have proposed methods for refining plan knowledge through environment interaction. Their methods are based on a Strips-like representation of the domain operators and the refinement occurs through a modification in the pre-conditions or the post-conditions of these operators. The know-how acquisition schema presented in this section offers an alternative approach. Instead, of structurally changing operator representations, new knowledge is added to the long-term memory. In subsequent problem solving, this knowledge is retrieved to change the problem solving behavior of the agent. So, in contrast to a change in the operators (ie. a change in the problem space) we suggest adding to the memory rules that will add representations to the state that can be processed to change the problem solving behavior. Also, the acquired rule is determined by a domain-independent acquisition schema that uses heuristic knowledge to identify the particular rule in a given instance of the schema. This is another contrast with the syntactic methods used in [67, 30] to identify changes to operator representations.

It is quite possible that the heuristics guiding the know-how rule acquisition cause a wrong rule to be acquired. For example, the modification to an

internal action may not really be the correct one to rectify the problem in formulation. In this case the wrongly modified internal action may again be incorrectly formulated. In other words, the acquisition of a know-how rule is not guaranteed to make the agent more accurate. The value of an acquired know-how rule is critically dependent upon the nature of the heuristics that are used in its acquisition.

4.1.2 Acquisition of new strategies

A strategy is defined as a pattern of activity for a given goal. For the representations we have used, a strategy is the function application of an internal action. An internal action is created for the goal of solving the input task. Its function application determines what further planning activity will occur. There are two possibilities to consider. The first possibility is that the internal action is directly mapped onto an external action specification or refined into internal actions that are themselves mapped onto external action specifications. In either case, the strategy represented by the original function applications has simply been carried out. This may happen, for example, for the function application, (derivative (o1 o2)). The second possibility is that the function application is executed by the adjoining of function applications that were not part of the original function application. When this happens we say that a new strategy has been created for the original function application. If the new strategy can be recognized and stored for use in subsequent problem solving, we say that there is a new strategy has been acquired for a function application. An example of strategy creation and acquisition will make clear how this type of knowledge can be acquired.

Let us consider the task instance shown in Figure 4.5. Our discussion of strategy creation can begin with the internal action, t5, shown in Figure 4.6, that is created during the course of the problem solving for this task instance. t5's function application is, (apply (interpolating-polynomial (substitute (th th0)))). The *apply* function represents the application of the substitution, (substitute (th th0)), to the interpolating-polynomial, $th(t) = d+ct+bt^2+at^3$. When t5 is in-focus, there are two applicable supervisory operators, sp-80 and sp-116. These operators have been specified in Figure 4.7. sp-80's intent is to create an internal action to write an equation in Mathematica's environment. sp-116's intent is to refine the compound function application of t5. t5's function application is compound because its second argument is itself a

function application. Let us assume that the contention between sp-80 and sp-116 is resolved in favor of sp-80. sp-80 applies to extend the task plan by the addition of t6. t6's function application represents the internal action to write the interpolating polynomial into Mathematica's environment. Next, sp-116 is selected for application. When sp-116 applies, it extends the task plan by adding t7 and t8, that are the refinement of t5. t7's function application is the substitution function while t8's function application is the original *apply* function applied to the interpolating-polynomial and the result of the substitution function application.

Calculate an interpolating polynomial, $th(t) = d + ct + bt^2 + at^3$, given that:

1. $th(t0) = th0$.
2. $th(t1) = th1$.
3. $\partial th/\partial t(t0) = 0$.
4. $\partial th/\partial t(t1) = 0$.

Figure 4.5: An instance of an interpolation calculation.

Next we find that both t6 and t7 can be selected to be in-focus. sp-8 is the supervisory operator that tries to select an internal action to be in-focus. Suppose the contention is resolved in favor of t6. As shown in Figure 4.8, t6 is mapped onto the external action specification, `th == d + c t + b t^2 + a t^3`, whose execution result, r6, has the description, interpolating-polynomial because the act of writing something into Mathematica's environment does not modify what was being written. Next, t7 is made the in-focus internal action. It is mapped onto the external action specification, f7, whose execution produces the result, r7. Notice that the description of r7 uses the substitute relation while the function application for t7 had referred to the substitute function. With t7 having been executed, t8 is made the in-focus internal action. At this point, we find that sp-82 and fp-1 are in contention. sp-82 is a supervisory operator whose intent is to replace data object in a function

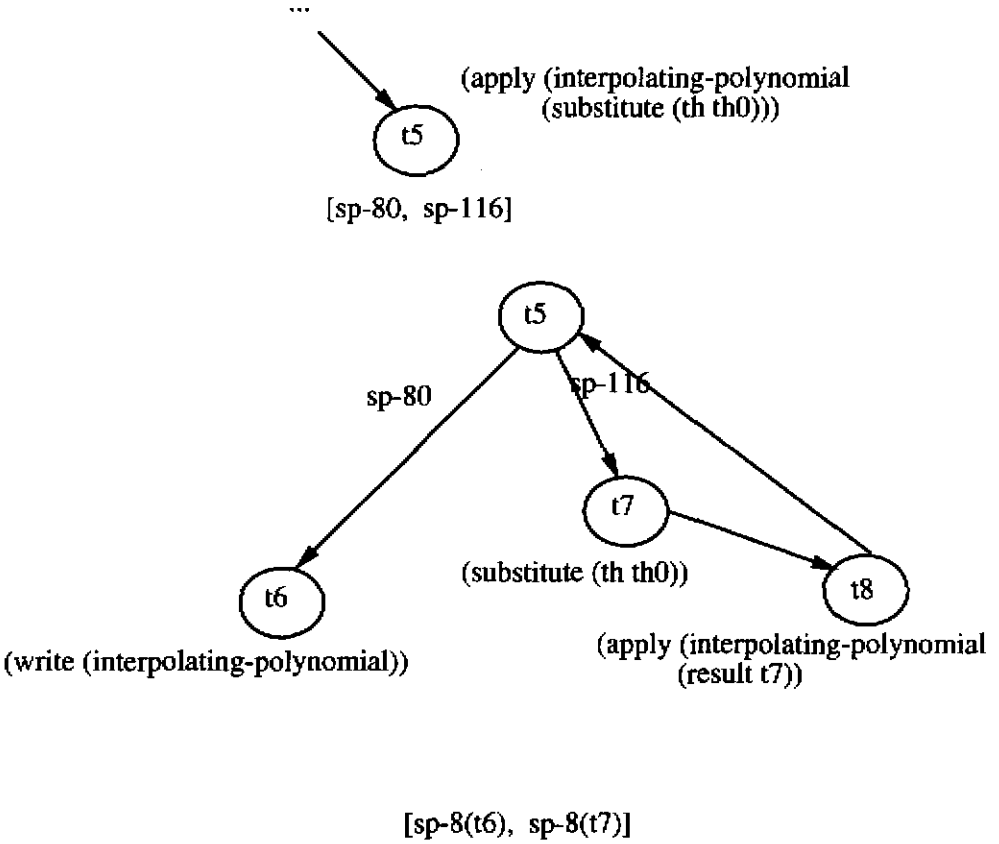


Figure 4.6: The calculation graph to illustrate strategy creation.

application with an external reference to that data object. An external reference is possible here because the interpolating-polynomial is available in Mathematica's environment due to the execution of f6. fp-1 is a formulation operator that initiates planning for an external action specification for t8. Suppose that the contention is resolved in favor of sp-82. When sp-82 is applied it replaces the first argument of the function application of t8 with the external reference, (result t6), as shown in Figure 4.9.

At this point a new strategy has been created. The original function application of t6 has been changed by adjoining of the function application of t6 to t8. The difference between the old strategy for t5 and the new strategy for t5 is that the object to which a substitution is to be applied is

sp-80: If an in-focus internal action applies a function to an object that is an equation then write that object.

sp-116: If an in-focus internal action has a compound argument then split that internal action.

sp-82: If the in-focus internal action applies to a data object that is available externally then replace that argument with a reference to the result of the internal action that produced the externally available data.

Figure 4.7: The inference rules for the strategy creation example.

made available externally and an external reference to this object is used. In the task instance of Figure 4.5 we see the need for two substitutions to be applied to the interpolating-polynomial. Under the old strategy each time the interpolating polynomial will be written out as part of the external action specification. The new strategy will simply write the interpolating-polynomial into Mathematica's environment and use the externally available data in each of the two substitutions. It must be noted that the creation of a new strategy is critically dependent upon the control knowledge used. In our discussion of the example we assumed that exactly the right kind of control knowledge was available. This can easily not be so. For example, Figure 4.10, shows three examples of reasonable control knowledge that would prevent the creation of a new strategy in the above example. Since the creation of a strategy is dependent upon the particular control knowledge used it is profitable to acquire new strategies for use in subsequent problem solving. This way the re-creation of a useful strategy will be impervious to vagaries of control knowledge as the strategy will be available fully formed already.

The general schema for the acquisition of new strategies is shown in Figure 4.11. It consists of three steps. First, the new strategy has to be recognized. A new strategy is also a function application. Upon recognition of a new strategy a rewrite rule is added to the blackboard whose left side is function application that represents the original strategy and whose right side the function application that represents the new strategy. The third step is to add a rule to the long-term memory that is contingent upon the original function application and whose action is the rewrite rule. Strategy

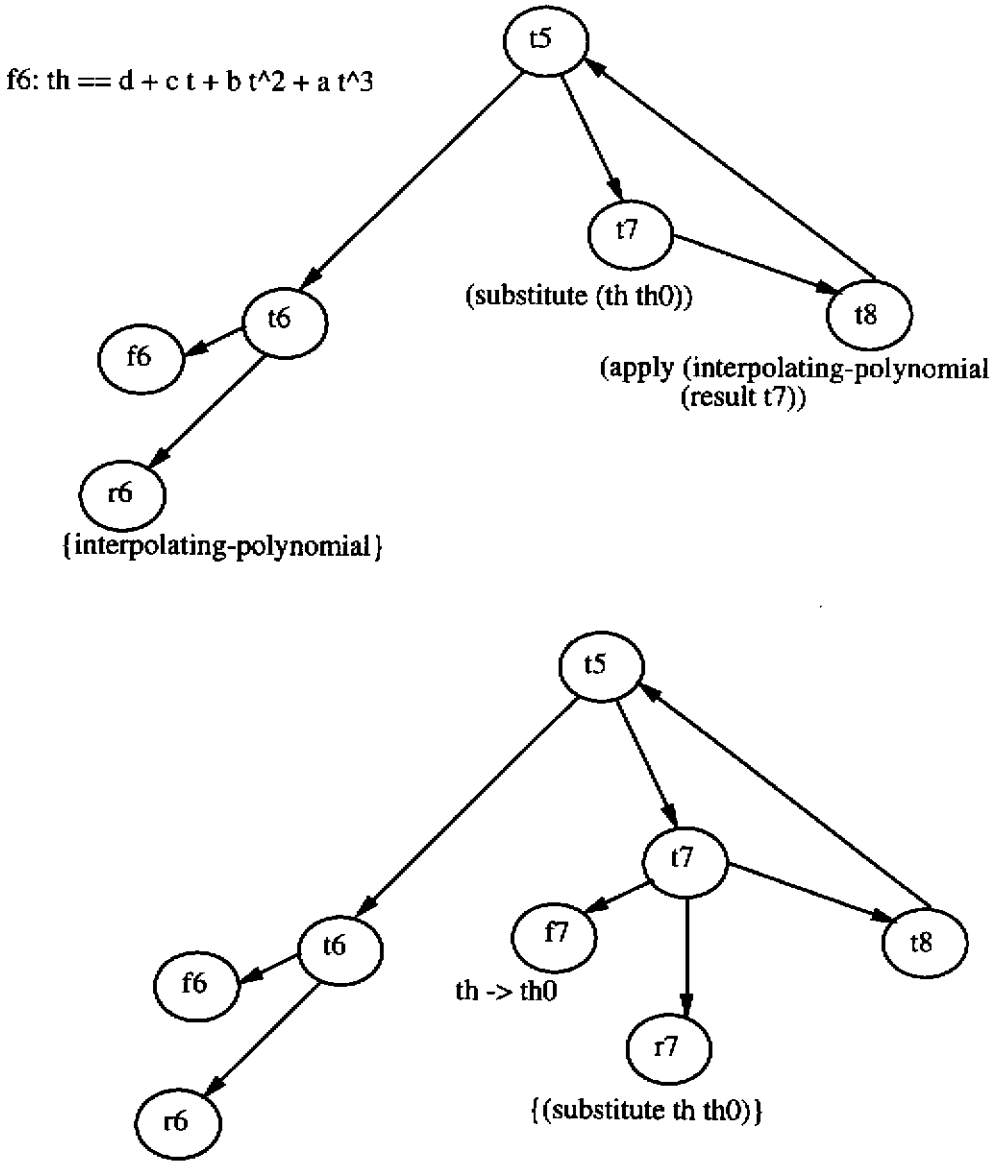


Figure 4.8: The continuation of the calculation graph to illustrate strategy creation.

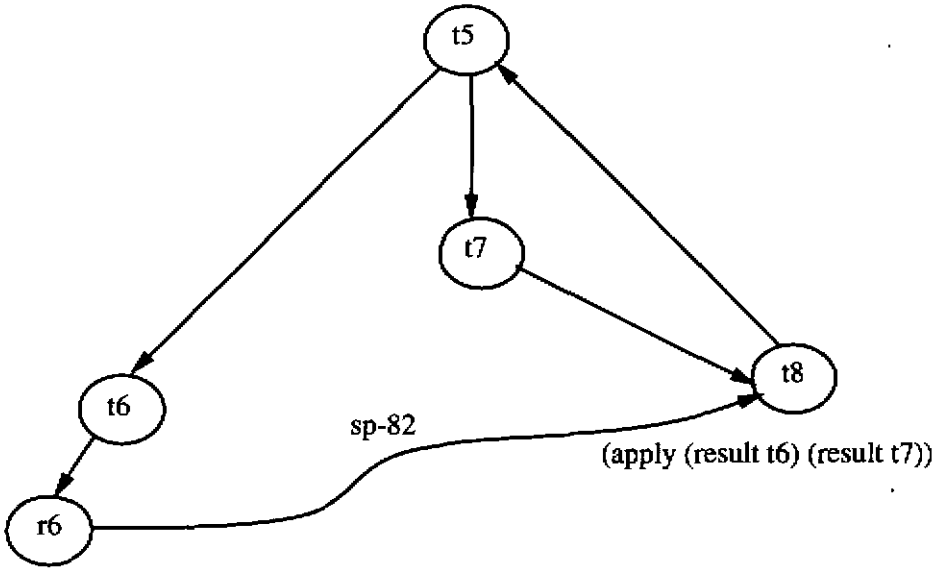


Figure 4.9: The continuation of the calculation graph to illustrate strategy creation.

1. sp-116 > sp-80; sp-8(t7) > sp-116. [sp-116 never given a chance ...]
2. sp-80 > sp-116; sp-8(t7) > sp-8(t6); sp-8(t8) > sp-8(6). [t6 remains suspended.]
3. sp-80 > sp-116; sp-8(t6) > sp-8(t7); sp-8(t8) > sp-82; fp > sp. [sp-82 never given a chance ...]

Figure 4.10: Three examples of control knowledge that will prevent strategy creation.

recognition occurs through the use of domain-independent heuristics that use the structure of the calculation graph. Observe that to recognize a new strategy is sufficient to create a concrete instance of the rewrite rule that links the original function application to the new function application. This is because a strategy *is* a function application. When the strategy rule is created in long-term memory, the constants in the rewrite rule are changed into variables. This is how a generalization of the concrete rewrite rule is obtained. Let us consider the acquisition of a strategy rule for our example. The following strategy recognition heuristic is used,

> If the end-refinement, O1, of an internal action, O2, refers to an internal action that was not part of the refinement schema then (rewrite (F1 ...) (F2 ...)), where F1 is O2's function application and F2 is O1's function application.

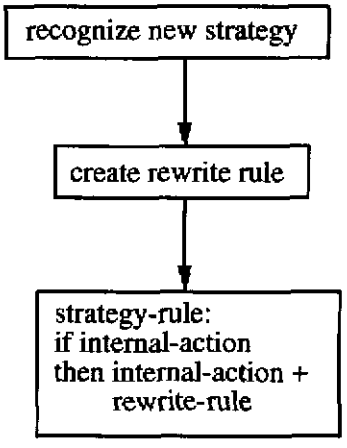


Figure 4.11: The general schema for the acquisition of new strategies.

The heuristic is applicable to the calculation graph of Figure 4.9 because t8 is the end-refinement of t5 and its argument refers to t6 that was not a part of the refinement schema for t5. The refinement schema for t5 is the ordered sequence comprised of t7 and t8. The heuristic creates the concrete rewrite rule,

```
(apply (interpolating-polynomial (substitute (th th0))))
→
(apply (write (interpolating-polynomial)) (substitute (th th0)))
```

Now, the third step of strategy acquisition schema applies. The condition of the acquired is taken to be t5's function application, (apply (interpolating-polynomial (substitute (th th0)))). However, the arguments are generalized to be variables. So, the condition of the acquired strategy rule is, (apply (A1 (substitute (A2 A3)))). The action of the acquired strategy rule is a generalization of the concrete rewrite rule. The generalization is obtained by replacing with variables the constants in the rewrite rule. For the above concrete rewrite rule this yields,

```
(apply (X1 (substitute (Y1 Z1))))
→
(apply (write (X1)) (substitute (Y1 Z1)))
```

So, the acquired strategy rule is,

```
If an internal action description is (apply (A1 (substitute (A2 A3))))
then add to the internal action description the rewrite rule,
(apply (X1 (substitute (Y1 Z1))))
→
(apply (write (X1)) (substitute (Y1 Z1)))
```

In subsequent problem solving if an internal action is created whose description matches the condition of the acquired strategy rule then that internal action's description is augmented with the rewrite rule that gives another strategy for that internal action. A supervisory operator can then apply to instantiate the rewrite rule for the internal action. In the above example, this replaces the function application, (apply (interpolating-polynomial (substitute (th th0)))), with the function application, (apply (write (interpolating-polynomial) (substitute (th th0)))). Figure 4.12 shows the processing of the internal action that ensues. In the figure, sp-90, is the supervisory operator that instantiates a strategy rewrite rule for an internal action. The supervisory operator, sp-116, directly refines t21 into t22, t23, and t24. Now there is no uncertainty about the execution of the new strategy for t21 because the only order in which these internal actions can be processed is t22 followed by t23 followed by t24.

It is useful to contrast the strategy acquisition described here with the learning of macro operators [26]. The difference is that the strategy representation is a new function application and not a composition of the operators that were used in creation of the strategy. A new strategy can also be seen as a proof plan [15] though the precise correspondence is a little hard to es-

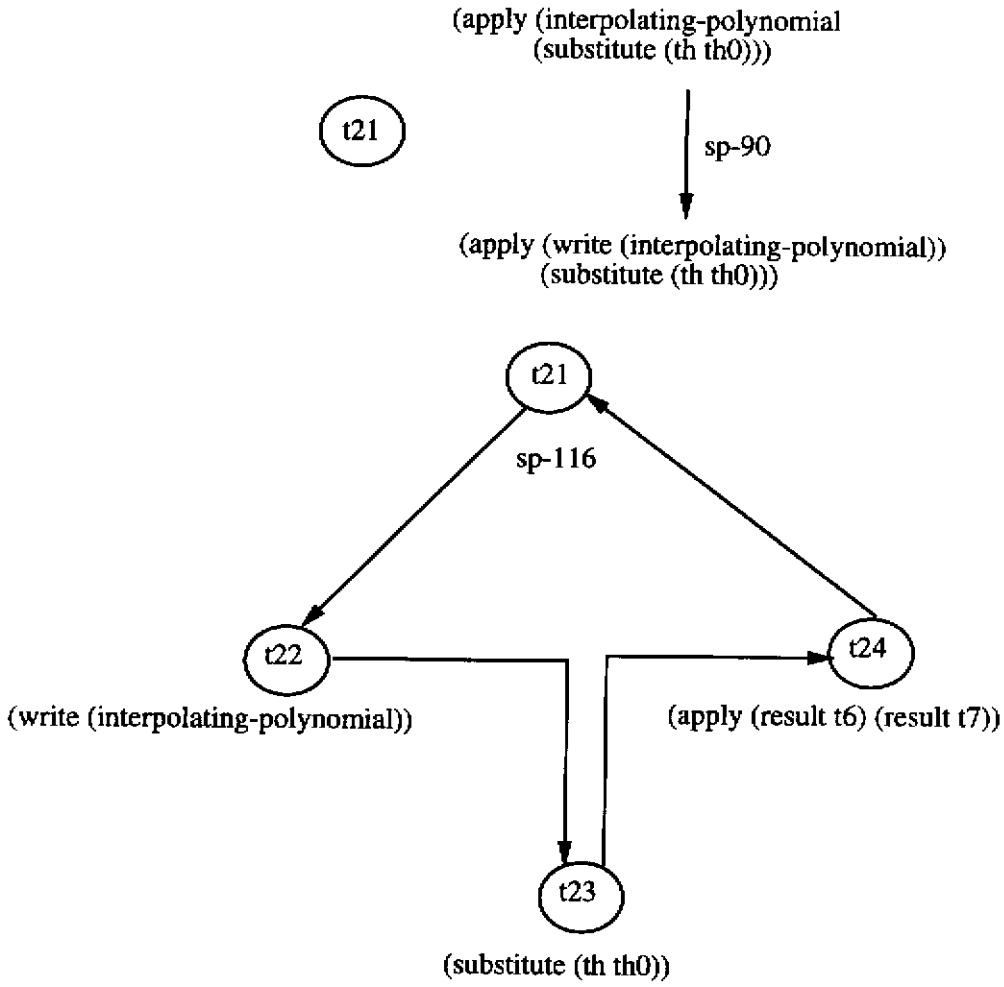


Figure 4.12: The use of the acquired strategy rule.

tablish since we are not engaged in proving theorems. The relatedness stems from the notion that a strategy is an abstract characterization of processing. This is true since a function application is an abstraction in that it is silent about what actual processing may be required to actually produce a result for it.

4.2 Acquisition of mapping knowledge by doing

The tentativeness in agent's software system knowledge can create a situation where multiple mappings are possible between an internal action and the software system function. A simple strategy for acquiring the missing discrimination knowledge is to try each of the several alternatives out and see which one is most appropriate. The general schema for the acquisition of mapping knowledge is shown in Figure 4.13. As shown in the figure, the first step in the general schema requires that the use of the software system function, *f1*, for an internal action, *t1*, leads to a failure. The next step is to reason about the properties, □, of the internal action that made *f1* the wrong choice. If □ can be found then a mapping rule is acquired that is contingent upon □ and whose action is to add to the internal action's description the suggestion not to use *f1* for it. In subsequent problem solving if the acquired mapping rule applies then the agent can choose to act upon the suggestion to not *f1* and eliminate as a possible alternative.

To illustrate the acquisition of a mapping rule let us consider the example illustrated in Figure 4.14. As shown in the figure, t7 is an internal action whose function application is to factor the integer, 21. There are two Mathematica functions, `Factor` and `FactorInteger`, that can be used to plan an external action specification for t7. However, sufficient mapping knowledge is not available to make an informed choice among the two alternatives presented. Therefore, the strategy to try each alternative in turn is adopted. In the first instance the external action specification is, Factor[21]. Its execution produces a result that is the same as the input of t7's function application. A general interpretation heuristic is to recognize a result as a failure if no change is produced as a result of a function execution. This heuristic applies here and r7 is assigned the description, fail. This completes the first step of

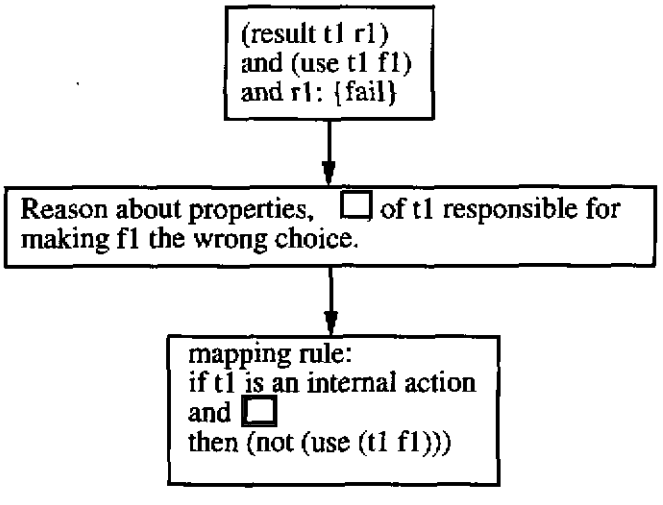


Figure 4.13: The general schema for the acquisition of mapping knowledge.

the general schema for acquiring mapping rules. The next step is to reason about the properties of t7 that made the use of Factor the wrong mapping. The properties of an internal action that could be responsible are the types of the arguments of the function application. Of course, there is always the option to use the name of the arguments as the responsible property. That is, to assume that if the internal action's function applies to 21 then Factor is a wrong choice. Clearly, this is a limited value property since it does not offer any generalization over the instance being considered. However, it must be pointed out that the particular property chosen to stand in for □ in Figure 4.13 is very much a function of the heuristic knowledge of the agent. In our example we assume that the identified property is the fact that the argument of t7's function is an integer. Therefore, the acquired mapping rule is,

If an internal action description is, (factor (O1)), and O1 is an integer, then do not use Factor for the external action specification for O1.

The general schema for acquiring mapping rules by doing is a domain-independent way of acquiring knowledge of mappings between internal actions and external functions. The schema requires that there be heuristic knowledge to identify the properties of an internal action that make a particular mapping to an external software system function wrong. The schema

stipulates no restriction of the nature of these properties. The only requirement is that such properties be identified. Once the properties of the internal action are identified a mapping rule is added to the long-term memory to suggest not using the external function for an internal action satisfying these properties.

4.3 Acquisition of function properties by doing

Sometimes the agent's expectation for an external software system function is violated by the execution results obtained for an external action specification. An expectation violation can lead to the inference of a new property of the external function. In this section we discuss how the agent can acquire new property rules for an external function. The general schema for property rule acquisition is shown in Figure 4.15. As shown in this figure the first step is to execute an external action specification. The second step is to recognize that the execution result obtained for the external action specification violates an expectation for it. The expectation violation is the trigger for reasoning about the observed behavior of the function that was executed. A general heuristic is to infer the proposition that if the inputs to the function have the properties that they have then the output produced by function execution will be the observed output. In the fourth step of the schema, a property rule is added to the long-term memory whose contingency is the use of the function that produced the expectation violation and whose action is to add the inferred property to the blackboard. In subsequent problem solving, this property of the function will be retrieved from memory as soon as the decision to use this function is taken. Therefore, the agent's subsequent performance can factor in this additional knowledge. The net effect of the property acquisition is to make available knowledge previously generated by the execution of a function prior to its subsequent execution.

To illustrate the acquisition of a property rule let us consider the example shown in Figure 4.16. In the figure t9 is an internal action whose function application, (compare (o1 o2)), is a comparison of two constant objects. The external action specification for t9 is f9 that uses the `Equal` function to make the comparison. The execution result for f9 is r9 whose description is the

o1: {integer, (name (21))}

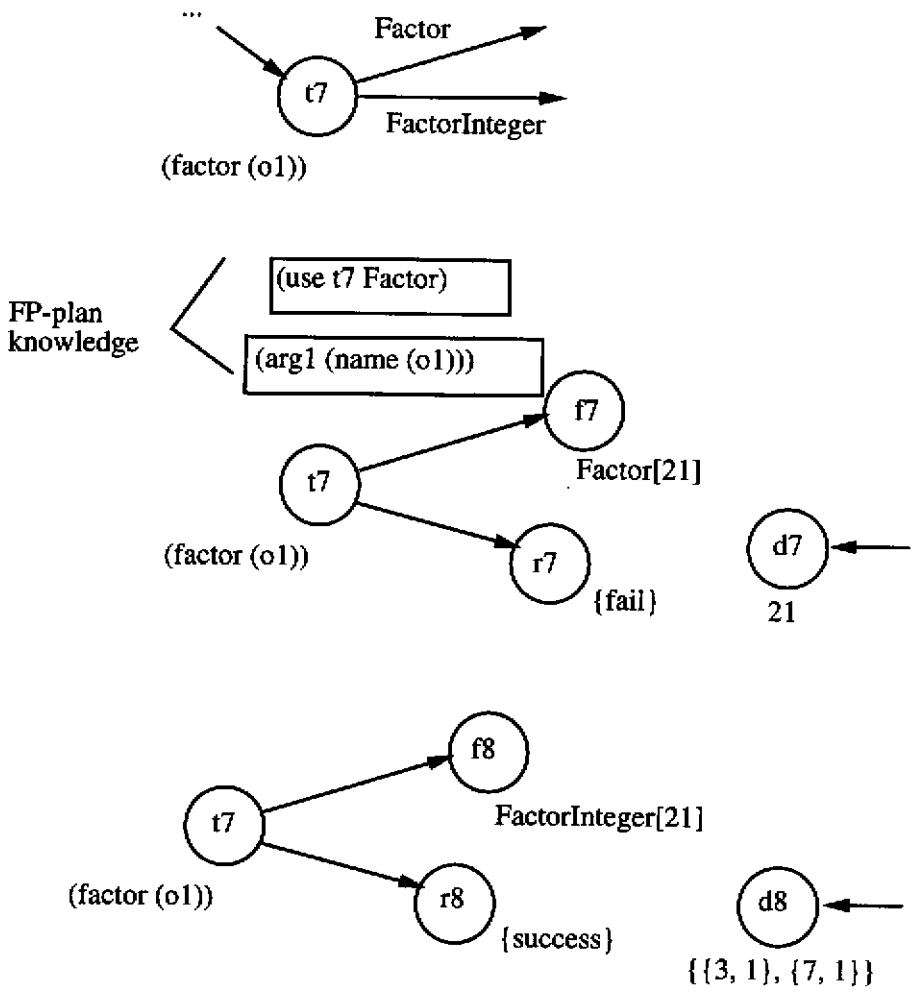


Figure 4.14: An illustration of a mapping rule acquisition.

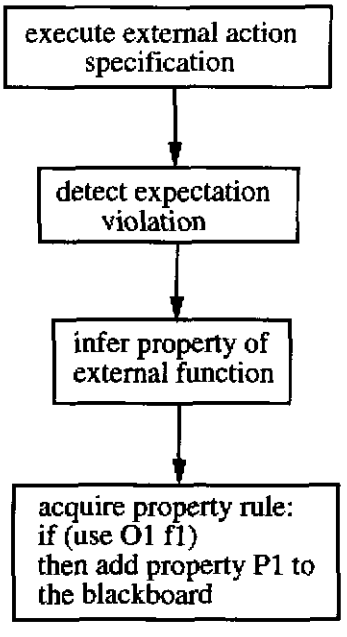


Figure 4.15: The general schema for property rule acquisition.

predicate, (equal a b). A further inference is made that the application of `Equal` to the two objects produces no change in the output. The expectation of t9 was that the result will be *false* since the inputs to the external function were known to be different. Since r9 violates the expectation the second step of property acquisition is completed. The inferred property of the function, as per the general heuristic mentioned above, is that if the inputs to the function are not equal then the result of function execution is to obtain as output the objects that were the input to the function. This condition is represented by the predicate, (output no-change). Given that a property of the function is now available, the following property rule is acquired,

If there is a decision to use the `Equal` *function then add to the blackboard the property, (if (and (argument1 X) (argument2 Y) (not (equal X Y))) then (output no-change)), of the function.*

It is useful to examine the heuristic that infers the property of the function that is acquired. The heuristic correlates the properties of the input to the function with the observed output produced by the execution of the function. So, in our example, the inputs to the function had the property that

they were not equal and the output of function execution had the property, (output no-change). But neither the inputs not the observed output had *only* these properties. They both had other properties as well. For example, the inputs could be described by their names, a and b, and the output could be described as an equation whose left side was a and whose right side was b. We have presented no *solution* to the identification of the *right* properties for the input and the output. The view put forth is that the agent's heuristic knowledge will enable inference of a reasonable property of the function upon recognition of an expectation mismatch. Additional heuristic knowledge can make the agent's inference process more refined. For example, a general heuristic can be to prefer a semantic property over a syntactic property such as the names of the arguments. Nothing in our general schema for property acquisition suggests that arbitrary inference may not be used for establishing the appropriate property to acquire for the function. In fact, the property that we showed as being acquired for our example, is not quite correct. It suggests that the only requirement for the output to not change is that the inputs be not equal. This is not true. The real requirement is that the inputs should not be equal and should be symbolic as well. But acquisition of incorrect knowledge is an inescapable consequence of using a heuristic theory to generate the knowledge that is acquired. Another noteworthy point is that the generalization of the acquired knowledge is controlled by the inference process that produces the property of the function. If this inference process were to identify properties that were the names of the inputs and the output then no generalization would be had. The property rule that was acquired in our example generalizes because it does not refer to the names of the inputs and outputs but instead to their properties.

4.4 Acquisition of control rules for strategy rules

In this section the acquisition of control rules for strategy rules is described. A strategy rule initiates some processing whose necessity may not be known a priori. Therefore, it is useful for the agent to have control knowledge suggesting when it is appropriate to select a given strategy. For example, the decision to simplify a result, obtained by the execution of a Mathematica

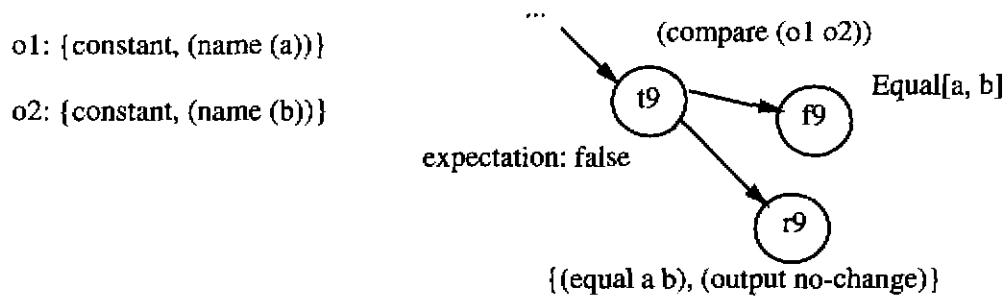


Figure 4.16: An illustration of property acquisition.

function, is a strategy rule. If the agent does select this strategy it may find that there was no need for simplifying the result. Such a conclusion could be based either upon an inability to actually simplify the result or the recognition of a property of the result that suggests the futility of the attempt to simplify it. The control knowledge for this strategy ought to point out forms of the result that would be useful to simplify and those that would not. One way of making such control available is to encode it manually. For example, for the simplify strategy, a control rule is to simplify a fraction. An alternative method of obtaining the control rules for a strategy is to have the agent acquire them automatically. The general schema for the acquisition of control rules for a strategy is shown in Figure 4.17. As shown in the figure the first step is to select an applicable strategy. Next, the strategy is applied. The third step is to determine whether or not the strategy application was useful. If the strategy is found useful then a control rule associating some properties, $\Box$, of the internal action to which the strategy was applied, with the suggestion to use the strategy is acquired. If the strategy application was not useful then a control rule associating some properties of the internal with the suggestion to not use the strategy is acquired.

Let us consider the acquisition of a control rule for the strategy rule to simplify the result of a Mathematica function execution. The operator, sp-14, is the strategy rule for simplification. It applies when the result of an internal action is available. The instantiation of the general schema for sp-14 is shown in Figure 4.18. An example of the application of the strategy is shown in Figure 4.19. In the figure t1 is an internal action to find the derivative of $K/(1+st+KK1)$ with respect to K. t1 is mapped onto the external action

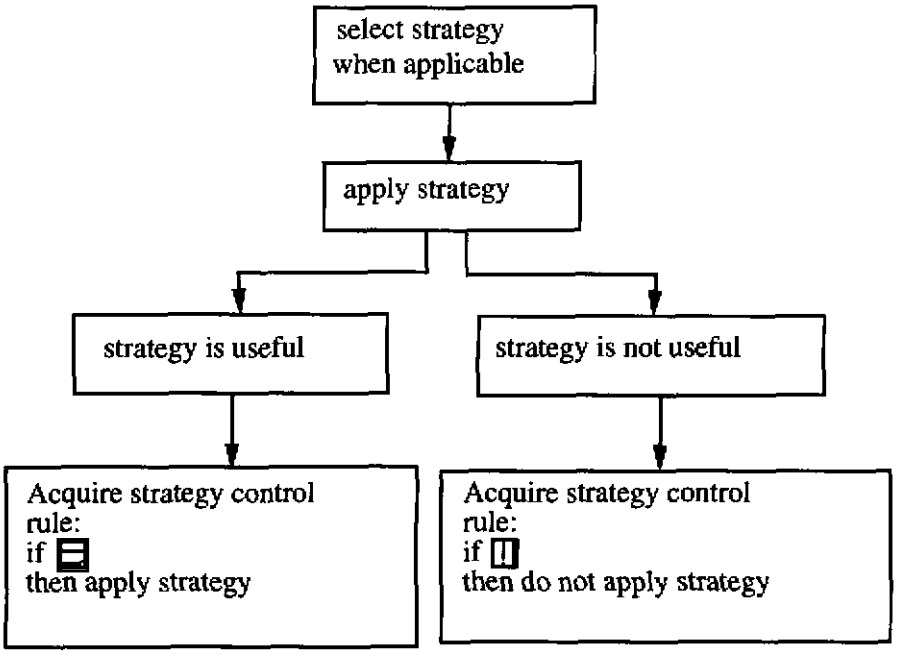


Figure 4.17: The schema for strategy control rule acquisition.

specification, f1, whose execution result is r1. The simplification strategy, sp-14, creates the internal action, t2. t2 is mapped onto the external action, f2, whose execution result is r2. A part of the simplification strategy is to compare the original result with the simplified result. The supervisory operator, sp-15, creates the internal action, t3, for the comparison. t3 is mapped onto the external action, f3, whose execution result is r3. Since r3 is false it is inferred that sp-14 is successful for t1. Now a control rule for the simplification can be acquired. To do this the conditions of the acquired must be determined. The conditions are to be a subset of the properties of t1. The properties of t1 include the properties of r1, the result object for t1. There are many alternative subsets of properties that can be selected to be the conditions of the acquired control rule. For example, the name of the result could be considered, or the fact that the result is a sum, or perhaps the fact that the result is a sum whose second term is a fraction. It is assumed that knowledge acquisition heuristics will determine a subset of properties to be made the conditions of the acquired control rule. For example, if the

third alternative is selected then the acquired control rule is,

If the result, O1, of an internal action is a sum, such that the second term of the sum is a fraction, then add (simplify O1) to the blackboard.

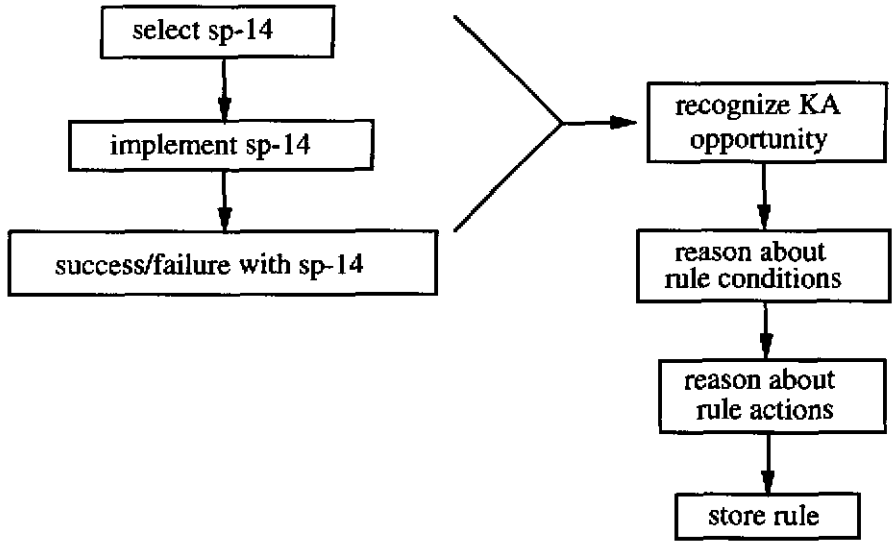
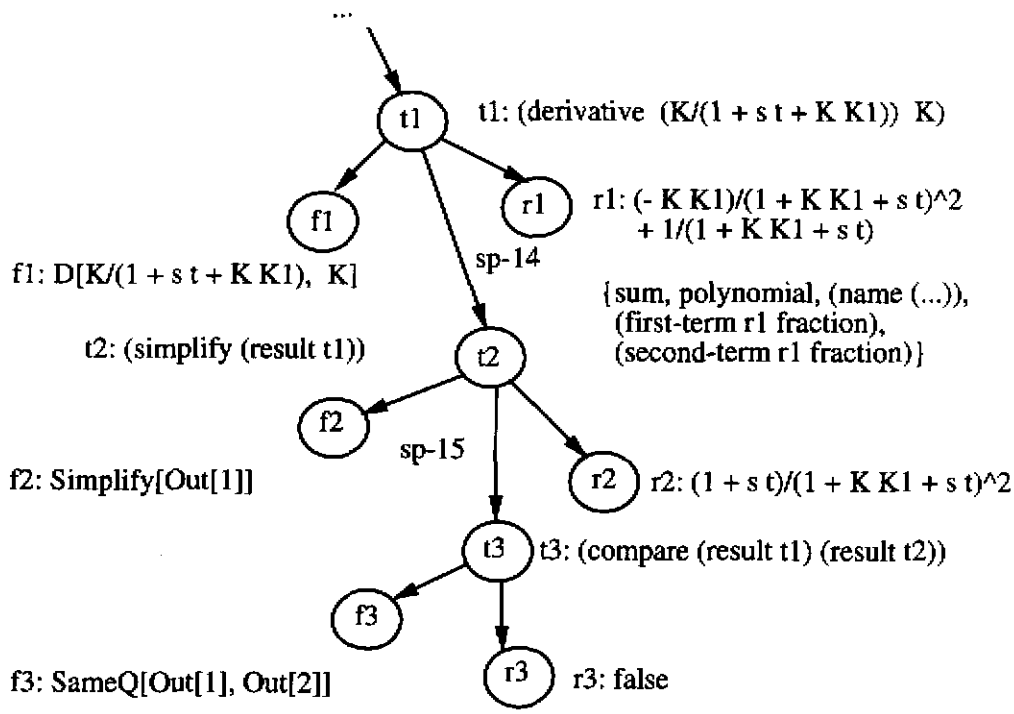


Figure 4.18: The instantiated schema for sp-14.

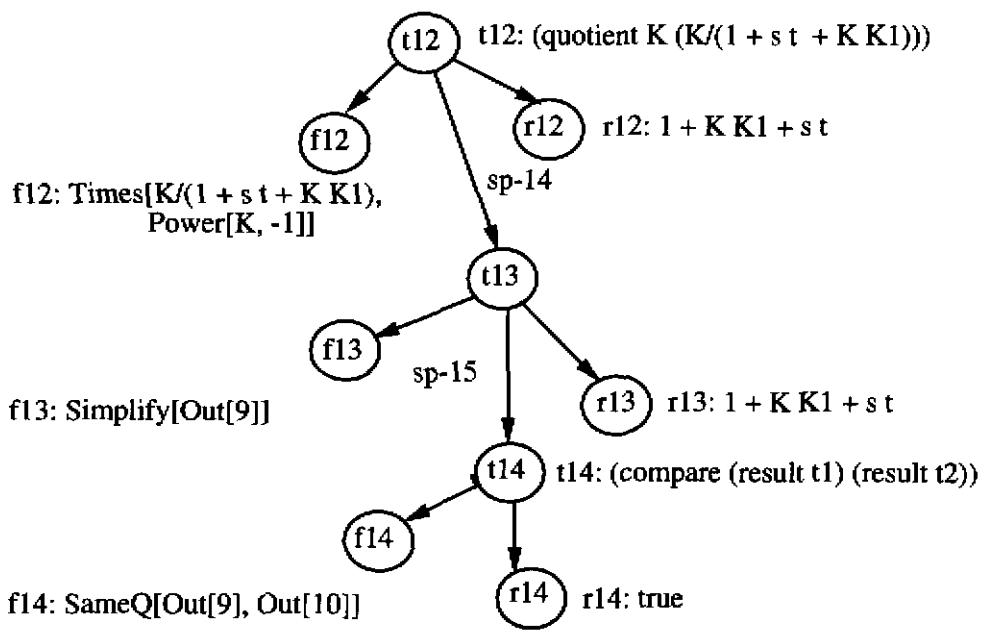
Next, let us consider the processing shown in Figure 4.20. In the figure t12 is an internal action to find a quotient. t12 is mapped onto the external action specification, f12, whose execution result is r12. The simplification strategy, sp-14, creates the internal action, t13. t13 is mapped onto the external action, f13, whose execution result is r13. sp-15 creates the internal action, t14, to compare the result of t12 and t13. t14 is mapped onto the external action, f14, whose execution result is r14. Since r14 is true it is inferred that sp-14 is a failure for t12. Now a control rule for the simplification can be acquired. There are many alternative subsets of properties that can be selected to be the conditions of the acquired control rule. For example, the name of the result could be considered, or the fact that the result is a sum, or perhaps the fact that the result is a sum and its first term is a constant and its second and third terms are products. It is assumed that knowledge acquisition heuristics will determine a subset of properties to be made the conditions of the acquired control rule. For example, if the third alternative is selected then the acquired control rule is,



fact(sp-14 t1 success)

Figure 4.19: A positive example of the application of the simplification strategy.

If the result, O1, of an internal action is a sum, such that the first term of the sum is a constant and its second and third terms are products, then add (not (simplify O1)) to the blackboard.



fact(sp-14 t1 failure)

Figure 4.20: A negative example of the application of the simplification strategy.

In subsequent problem solving, when a result object, say o1, is a sum whose first term is a constant, the acquired control rule will add the fact, (not (simplify o1)). Given this fact about o1, the agent can decide not to select the sp-14 strategy to simplify the result. If, however, o1 is a sum, whose second term is a fraction, then the acquired rule will add, (simplify o1), and in this case the agent can choose to select the sp-14 strategy. So, the acquisition of the control rules for a strategy rule enable the agent to be more informed about the selection of the strategy rule with experience. The two acquired control rules apply when the result object is a sum. If

we consider the class of all sums to be a single concept then the subclasses covered by the two rules are illustrated in Figure 4.21. The acquisition of control rules is guided by heuristics. These heuristics are independent of the purpose to which the properties of the result objects are to be put, namely, to determine the conditions for control rules for the simplification strategy. There is no guarantee that the acquired rules may not be either over-general or over-specific. For instance, the positive rule in our example applies to the specific subclass of sums whose second term is a fraction. However, this is still a considerable generalization from the particular instance that was used to acquire the rule. The negative rule is likely over-general since it will suggest that simplification not be tried on a result whose first term is a constant and whose second and third terms are products while also containing fractions.

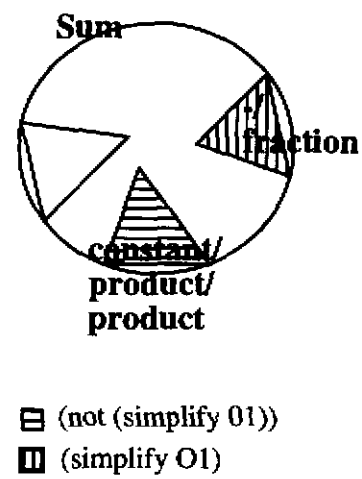


Figure 4.21: The coverage of the two acquired control rules.

A strategy, such as to simplify an intermediate result, may require arbitrary processing. Only the final outcome of strategy application is of interest. This differentiates a strategy from an operator. In our example we saw several process operators making up the application of simplification strategy. Consequently, the credit assignment problem appears in a different form. Instead of requiring a determination of which operator in a sequence of operators was useful and which was not we need to know if the strategy was useful or not. As the agent goes about applying the strategy it is also building chunks. However, the chunks represent an implementation of the

selected strategy. These chunks do not include a production that says when to use strategy. The chunks correspond to the reflective acquisition that is a side-effect of problem solving. This is the reason for using deliberate acquisition to obtain the control rules for strategy application. It is useful to compare the acquisition of control rules for strategy rules described with the learning in LEX [47]. LEX learns control rules for its operators. In LEX, a distinction was not made between strategy rules and operators. We have discussed the acquisition of control rules for strategy rules. As mentioned above this changes the credit assignment problem somewhat. In LEX, a version space [46] was used represent partially learned control rules for operators. We assume that heuristics control the generalization achieved in the acquired control rules. Each acquired control rules is a specification of when it is useful to either apply a strategy or not apply the strategy. The acquisition can be seen as a form of discrimination learning [39]. The agent begins with discrimination knowledge for a strategy rule. With experience the agent builds up discrimination rules for a strategy rules.

4.5 Reflective knowledge acquisition

In this section the automatic acquisition of control knowledge (of the type described in section 3.5) is described. It is a useful capability of the agent to acquire control knowledge automatically.

Case 1

Let us consider the following trace of (obtained for sa: task 2):

```
80      ==>g: g936 (operator tie) [o1073 o1070]

(operator o1073 ^intent i814 ^entry f2 ^input-object c795
                ^name formulation-process)

(operator o1070 ^cause f1063 ^entry w1 ^input-object c795
                ^name output-process)

81          p: p1079 (lookahead)
82          s: s1080
```

```
83          o: o1081 ((output-process) evaluate-object)
84          ==>g: g960 (operator no-change)
85             p: p12 (task)
86             s: d919 (blackboard)
87             o: c922 (output-process)
88             ==>g: g1089 (operator no-change)
89                p: p1090 (output-process)
90                s: s1091
91                o: o1093 (mma-syntax)
92                ==>g: g1094 (operator no-change)
93                   p: p1095 (mma-syntax)
94                   s: s1096
95                   o: o1097 (wait-output)

build:p943
build:p944
build:p945

evaluation of operator o1070 (output-process) is failure
96          o: o1082 ((formulation-process) evaluate-object)
97          ==>g: g1098 (operator no-change)
98             p: p12 (task)
99             s: d926 (blackboard)
100            o: c923 (formulation-process)
101            ==>g: g1104 (operator no-change)
102               p: p1105 (formulation-process)
103               s: s1106
104               o: o1108 (determine-mma-function-arguments)

build:p946
build:p947
build:p948

evaluation of operator o1073 (formulation-process)
is partial-success

build:p949

105     o: o1073 (formulation-process)
```

```
firing 106:475 p946
firing 106:475 p947
106     o: o1122 (resolve-references)
```

At the end of decision cycle 80 (d80) two operators are applicable, o1073 and o1070. One is a formulation operator and the other is an output process operator. Soar detects an impasse because it cannot decide (in the task space) which of the two operators to select. Consequently, Soar enters the lookahead space in d81 to apply each of the contentious operators. First, o1070 is applied. In d95 we find that Soar cannot implement this operator. Recognizing failure to make progress with o1070 Soar evaluates the choice of o1070 to be a failure. Next, o1073 is applied. This operator can be implemented and therefore Soar evaluates the choice as partial-success. With these evaluations available for the two operators, Soar returns to the task space and select o1073 as the next process operator to apply. The chunk p949 is the one that captures the control knowledge governing the choice between a formulation operator and an output operator. The chunk is shown in Figure 4.22.

Case 2

Suppose two operators are in contention. Soar detects an impasse and enters the lookahead space to resolve the impasse. In the lookahead space there can be available knowledge that adds a preference for one operator over another. This will resolve the impasse and cause Soar to create a control rule to order these operators in this manner subsequently without having to enter the lookahead space to find the ordering. Therefore, Soar makes explicit the control knowledge for these operators by acquiring the control rule. An example of such a control rule acquisition appears in the following trace (obtained from sa: task-2):

```
17      ==>g: g437 (operator tie) [o431 o434]
(goal g437 ^quiescence t ^impasse tie ^object g11 ^attribute operator
           ^choices multiple ^item o431 o434)
(operator o431 ^entry sh8 ^input-object o424 ^name supervisory-process)
(operator o434 ^entry sh6 ^input-object o424 ^name supervisory-process)
18         p: p438 (lookahead)
19         s: s439
build:p1042
```

```
(sp p949
   (goal <g1> ^problem-space <p1> ^state <s1>
                                  ^operator <o1> + <o2> +)
   (problem-space <p1> ^name task)
   (operator <o1> ^entry f2 ^name formulation-process
                            ^input-object <c1>)
   (operator <o2> ^entry w1 ^name output-process
                  ^cause <f1> ^input-object <c1>)
   (sexp <f1> ^value use ^next <l3>)
   (state <s1> ^fact <f1>)
   (object <c1> ^matched-description <d1>)
   (sexp <d1> ^value derivative ^next <l1>)
   (sexp <l3> ^value <c1> ^next <l2>)
   (sexp <l2> ^next nil ^value <s2>)
   (sexp <l1> ^value <a1>)
 -->
   (goal <g1> ^operator <o1> > <o2>))
```

Figure 4.22: An example of an automatically acquired control rule (Case 1).

```
20      o: o434 (supervisory-process)
```

Operators o431 and o434 are in contention. Both are supervisory process operators. Soar enters the lookahead space in d18. In the lookahead space a rule adds a preference for o431 over o434. This resolves the impasse and creates a control rule, p1042, shown in figure 4.23. That the preference rule ordering the two contentious operators applies in the lookahead space and not in the task space is noteworthy. The preference rule is an inference drawn on the subgoal to solve for the contention between the two operators. Arbitrary processing could occur to obtain an ordering between the two operators. But this processing is done when needed by Soar and within Soar. Once the control rule has been acquired, however, there is no need for such processing as Soar now directly orders the two contending operators in the task space. A second point to note is that the control rule is specific to operators, sp-8 and sp-6. While the processing in the lookahead space may have been general it is the result for the case at hand that is acquired for subsequent use. A third point to note is that such a control rule biases subsequent behavior.

A contention between sp-8 and sp-6 is going to be resolved in favor of sp-6 in the absence of additional preference information at the time of making the choice. That is, there is no deliberation about the choice the next time around. If this other occasion is exactly the same as the one producing the control rule all will be fine. However, if the situation were to change the rule will still apply although it may not produce appropriate ordering of the two operators. This is illustrative of the heuristic nature of acquired knowledge and its susceptibility to failure.

```
(sp p1042
   (goal <g1> ^operator <o2> + { <> <o2> <o3> } +)
   (operator <o2> ^name supervisory-process ^entry sp-8
                  ^input-object <o1>)
   (operator <o3> ^name supervisory-process ^entry sp-6
                  ^input-object <o1>)
 -->
   (goal <g1> ^operator <o3> > <o2>))
```

Figure 4.23: An example of an automatically acquired control rule (Case 2).

4.5.1 Instance acquisition

Every time Soar transits into a process space and modifies the blackboard new rules are created that serve to directly modify the blackboard the next time around without having to enter the process space. These rules are called instance rules because they capture the specific instance that was involved in the transition into the process space and will apply only when this specific instance manifests in subsequent problem solving. Their effect is in making the agent more efficient. These rules also correspond to the memory trace of problem solving retained for subsequent use.

```
(sp p916
   (goal <g1> ^state <s2> ^operator <o1>)
   (operator <o1> ^entry sp-10 ^name supervisory-process
                               ^input-object <c1>)
   (state <s2> ^fact <f1>)
   (sexp <f1> ^value computes ^next <n3>)
   (sexp <n3> ^value mma ^next <n2>)
   (sexp <n2> ^value derivative ^next <n1>)
   (sexp <n1> ^next nil ^value <s1>)
   (object <c1> ^description <d1>)
   (sexp <d1> ^value derivative)
 -->
   (sexp <f3> ^value formulate + formulate + ^next nil)
   (object <c1> ^status <f3> &, <f3> +)
   (state <s2> ^fact <f2> &, <f2> +)
   (sexp <f2> ^value sp-10 + ^next <n4> +)
   (sexp <n4> ^next nil + ^value <c1> +))

(sp p927
   (goal <g1> ^operator <o1> ^state <s2>)
   (operator <o1> ^name supervisory-process ^entry sp-10
                                            ^input-object <c1>)
   (state <s2> ^fact <f1>)
   (sexp <f1> ^value computes ^next <n3>)
   (sexp <n3> ^value mma ^next <n2>)
   (sexp <n2> ^value derivative ^next <n1>)
   (sexp <n1> ^next nil ^value <s1>)
   (object <c1> ^description <d1>)
   (sexp <d1> ^value derivative)
 -->
   (goal <g1> ^operator <o1> @))
```

Figure 4.24: Two instance rules that are automatically acquired.

Chapter 5

Experimentation with the agent

In this chapter several experiments to study the automatic use of Mathematica by the agent are described. Additionally, an experiment each to study the use of SignalProcessing [25] and a FrameKit [59]-based database system by the agent is described. The goals of the experimentation were to assess the adequacy of the agent's control structure, process organization, and representations for symbolic calculations using Mathematica. Another goal was to characterize the domain-independent and domain-dependent processing required in the agent. By experimentation, to a smaller extent, with two other software systems, we hoped to obtain further insights into the generality of the agent design. After describing each of the experiments, we conclude the chapter with an evaluation of the agent implementation.

5.1 Experiments with Mathematica

5.1.1 Sensitivity analysis

Sensitivity analysis in linear systems is the study of the dependence of system behavior on system parameters [28]. Typical parameters are initial conditions, natural frequencies, and dead times; system behavior can be the time response, the transfer function, or any other quantity characterizing the system dynamics. In this subsection an instance of the sensitivity anal-

ysis problem is discussed. The input task is to calculate system sensitivity with respect to the open-loop gain parameter, $S_K^T = \frac{\partial T}{\partial K}\frac{K}{T}$, for a system whose closed-loop transfer function is $T(s) = \frac{K}{ts+(1+KK1)}$. [1] The task is specified to the agent as the object network illustrated in Figure 5.1.

```
object:
    id: o1
    type: goal
    purpose: (S T K)
    data: d1 d2 d3 d4 d5

object:
      id: d1
      type: data
      description: (transfer-function)
                   (name (K/(1+K K1 + s t)))

object:
      id: d2
      type: data
      description: (parameter) (name t)

object:
      id: d3
      type: data
      description: (parameter) (name s)

object:
      id: d4
      type: data
      description: (parameter) (name K1)

object:
      id: d5
      type: data
      description: (open-loop-gain) (name K)
```

Figure 5.1: The input specification for the sensitivity analysis example.

The object, *o1*, is the goal whose purpose, *(S T K)*, denotes that the sensitivity is to be computed with respect to the open loop gain parameter

[1] The transfer function, T(s), is defined as the ratio of the Laplace transform of the output variable to the Laplace transform of the input variable under the assumption that all initial conditions are zero [49]. K, K1, and t are system parameters.

of the system. The goal object is linked to the data objects, *d1*, *d2*, *d3*, *d4*, and *d5* that specify the various system parameters. This input task specification is added to the task space blackboard by the user. The input to Mathematica and the output received from Mathematica for this task are shown in Figure 5.2.

I1: D[$\frac{K}{1+KK1+st}$, K]

O1: $\frac{-KK1}{(1+KK1+st)^2} + \frac{1}{1+KK1+st}$

I2: $\frac{K}{\frac{K}{1+KK1+st}}$

O2: $1 + KK1 + st$

I3: Times[Out[1],Out[2]]

O3: $(1 + KK1 + st)(\frac{-KK1}{(1+KK1+st)^2} + \frac{1}{1+KK1+st})$

Figure 5.2: The input and output produced for the example.

The agent processing on this instance is illustrated in Figure 5.3. The solution trace produced by the agent for this instance is shown in Figure 5.4. The decision cycles are numbered along the left margin [2] and the symbols G, P, S, and O stand for goal, problem space, state, and operator, respectively. An impasse is indicated by an arrow and indentation. Elided processing prior to a decision cycle is indicated by an ellipsis in the left margin. In figure 5.4 we have shown only a few of the operators of the supervisory, formulation, interpretation and external memory processes to demonstrate a working implementation of the agent. Agent processing begins in a problem-space called top-ps. In d3, a task operator is applied in top-ps to invoke the task problem space. In d7, there is a tie impasse as more than one operators are applicable in the task space. These are two supervisory process operators, *sp1* and *sp2*, that can be applied to the input task specification. The first operator is applicable because a definition for the input task exists and the second operator is a general heuristic that suggests an assessment of whether the computation can be performed by Mathematica. To resolve this impasse the agent

[2]In subsequent discussion we use the notation d<number> to refer to a particular decision cycle.

invokes the selection space that has a search control heuristic that resolves the contention in favor of *sp1* as it is is more specific than *sp2*. Therefore, in d10, Soar selects *sp1* whose application produces an impasse and leads to the invocation of the supervisory problem space. In this problem space the impasse is resolved by the application of the *use-definition* operator which is a rewrite rule for the input calculation that instantiates the root node, t0, of the calculation graph with the internal action, *(product (derivative (transfer-function open-loop-gain)) (quotient (open-loop-gain transfer-function)))*.

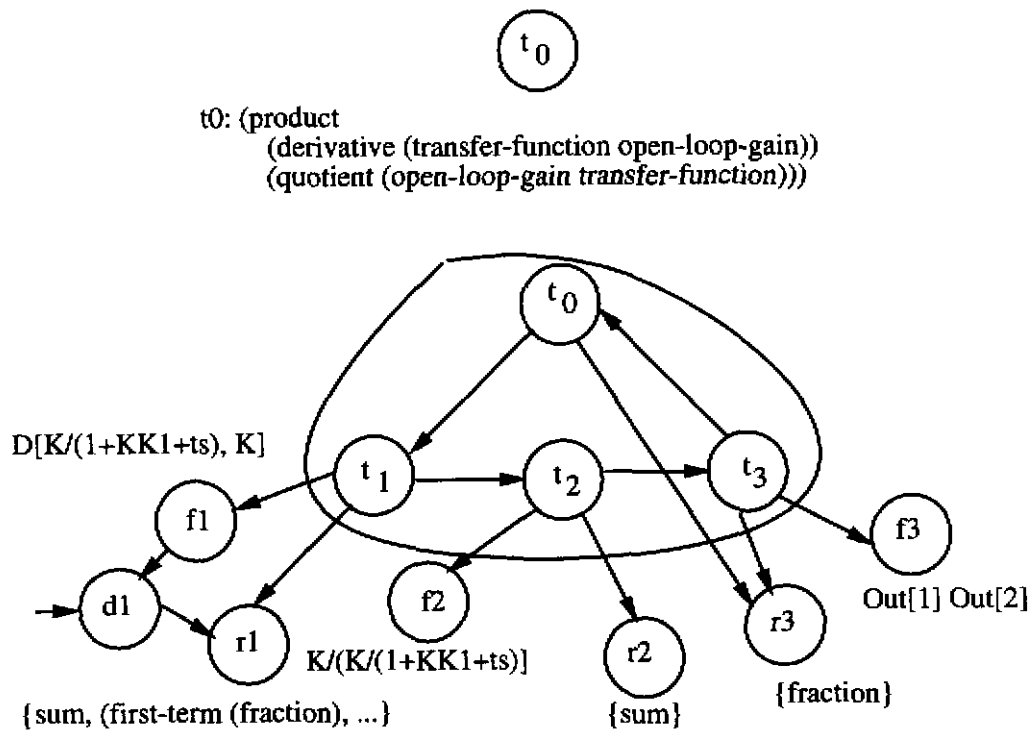


Figure 5.3: The agent's processing on a sensitivity analysis task.

Since the calculation associated with t0 consists of a function whose argu-

ments are themselves functions another supervisory process operator, *sp3*, is selected in d20, whose application results in the splitting of t0 into the ordered sequence of nodes, t1, t2, and t3, with the associated calculations, *(derivative (transfer-function open-loop-gain))*, *(quotient (open-loop-gain transfer-function))*, and *(product (result t1) (result t2))*. At this point, an algorithm has been synthesized by the supervisory process consisting of the sequence, t1, t2, and t3. These calculations are contained in the frontier of the calculation graph. A supervisory operator, sp4, selects t1 to be the focus of attention and removes t1 from the frontier. Next, a formulation process operator, *f1*, recognizes that Mathematica can perform the calculation associated with t1. The implementation of *f1* is completed in d42 with the determination of a Mathematica function for t1. Further formulation process activity serves to establish the arguments for the Mathematica function and the instantiated Mathematica function is an executable plan for t1. As planning is incremental, the moment the agent has completed an executable plan it sends it off to Mathematica for execution. In d181, the formulation operator, *f3*, is selected to generate the actual syntax for the instantiated Mathematica plan for t1. The implementation of *f3* is completed in d346 and the input for Mathematica is transmitted by an output process operator. I1 is the transmitted output to Mathematica and O1 is the input received from Mathematica in response.

In d351, the agent applies an input operator that records the receipt of data from Mathematica. In d356, an interpretation operator is applied and the resulting impasse is resolved by the application of the build-meaning operator in the interpretation space that maps the received data onto a meaning representation. O1 is interpreted as a sum of two rational polynomials. As a default O1 is accepted by the agent as being the proper result for I1. The focus of attention is shifted from t1 to the next calculation in the sequence. Processing similar to above occurs between d361 and d513 to produce the second interaction with Mathematica with the output I2 being sent out and the result O2 being received in response from Mathematica. After completion of t2, attention is shifted to t3. The formulation operator *f1* determines an appropriate Mathematica function for t3. Now, the arguments of t3 refer to results of previously computed calculations. Since the external memory process maintains cognizance of previously computed results the agent recognizes that these results are available in the external environment. In d565, an external memory operator is selected. In the external-memory space there is an

```
0    g: g1
1    p: p2 (top-ps)
2    s: s4 (top-state)
3    o: o10 (task)
4    ==>g: g11 (operator no-change)
5       p: p12 (task)
6       s: s13
7       ==>g: g148 (operator tie: sp1 sp2)
8          p: p149 (selection)
9          s: s150
10      o: o142 (supervisory-process: sp1)
11      ==>g: g160 (operator no-change)
12         p: p161 (supervisory-process)
13         s: s162
...
16         o: o167 (use-definition)
...
20      o: o188 (supervisory-process sp3)
21      ==>g: g203 (operator no-change)
22         p: p204 (supervisory-process)
23         s: s205
...
25         o: o210 (split-calculation)
...
38      o: o270 (formulation-process f1)
39      ==>g: g273 (operator no-change)
40         p: p274 (formulation-process)
41         s: s275
42         o: o277 (determine-mma-function)
...
181     o: o286 (formulation-process f3)
182     ==>g: g825 (operator no-change)
183        p: p826 (formulation)
184        s: s827
185        o: o829 (mma-syntax)
...
```

Figure 5.4: The problem solving trace of the agent for the sensitivity analysis example.

```
346     o: o1271 (output-process)
347     ==>g: g1274 (operator no-change)
348        p: p1275 (output)
349        s: s1276
350        o: o1278 (mma-transmit)
I1
01
351     o: o1287 (input-process)
352     ==>g: g1338 (operator no-change)
353        p: p1339 (input)
354        s: s1340
355        o: o1342 (result-received)
356     o: o1355 (interpretation-process)
357     ==>g: g1359 (operator no-change)
358        p: p1360 (interpretation)
359        s: s1361
360        o: o1363 (build-meaning)
...
512     o: o1965 (formulation-process)
513     o: o1979 (output-process)
I2
02
514     o: o1987 (input-process)
515     ==>g: g2003 (operator no-change)
516        p: p2004 (input)
517        s: s2005
518        o: o2007 (result-received)
519     o: o2020 (interpretation-process)
520     o: o2031 (supervisory-process)
521     o: o2037 (interpretation-process)
522     o: o2045 (supervisory-process)
...
```

Figure 5.4 continued.

operator, *find-environment-index*, that obtains the indices into the external environment corresponding to these previously computed results. Processing between d570 and d687 produces the third interaction with Mathematica and the output, I3, is sent out and the input, O3, is received from Mathematica in response. Upon receipt of O3 the agent has completed the calculations for t0. The result of t3 is propagated to t0 and then communicated to the user.

```
565     o: o2204 (external-memory-process)
566     ==>g: g2248 (operator no-change)
567        p: p2249 (external-memory-process)
568        s: s2250
569        o: o2252 (find-environment-index)
...
686     o: o2748 (formulation-process)
687     o: o2762 (output-process)
I3
O3
688     o: o2770 (input-process)
689     ==>g: g2833 (operator no-change)
690        p: p2834 (input)
691        s: s2835
692        o: o2837 (result-received)
693     o: o2850 (interpretation-process)
```

Figure 5.4 continued.

We next consider another instance of a sensitivity analysis calculation. The change in the input task representation from the previous case is that the open-loop gain parameter is now a function of K1. That is, the input task is to calculate system sensitivity with respect to the open-loop gain parameter, $S_K^T = \frac{\partial T}{\partial K}\frac{K}{T}$, for a system whose closed-loop transfer function is, $T(s) = \frac{K}{ts+(1+KK1)}$, given that K1 is a function of K. The agent's processing on this example is shown in Figure 5.5. As shown in the figure the initial processing is identical to the previous instance. First, the root node of the calculation graph for the task is created using the definition of the sensitivity analysis for linear systems. Next, a supervisory operator splits the compound function application into simpler function applications to create internal actions, t1,

t2, and t3. However, t3, cannot be mapped into an external action since there is the dependency of K on K1 to be accounted for. Two supervisory operators, sp-63 and sp-64, are applicable. sp-63 creates the internal action, t4, that sets up a derivative of the transfer function with respect to the independent variable, K, keeping constant K1. sp-64 creates the internal actions, t6, t7, and t8, to apply the chain rule to find the derivative of the transfer function with respect to K1. Once the results for t4 and t8 have been obtained by the execution of the external actions, f4, f6, f7, and f8, another supervisory operator, sp-65, applies to create the internal action, t9. This operator represents the knowledge that if a decomposition of an internal action has been attempted then the result of the internal action is obtained by a composition of the results of the constituent internal actions. In this case, the composition is the sum of the results of t4 and t8. When a result for t9 has been obtained by the execution of the external action, f9, the processing can once again resume the course taken in the previous instance of this calculation.

One of the major sources of variability in the processing requirements of input tasks is simply the variability in the problem data. Comparing the two instances of the same calculation we find that a small change in the input data, namely, a dependence of K1 on K, leads to several additional problem solving steps. The agent responds to input variability by accessing appropriate mathematical knowledge. This was the knowledge of partial differentiation in our example. Clearly, some additional task knowledge was used in the second instance. However, the basic processing operators used in the first instance remained unchanged. This is a characteristic of the agent design that we will find true for each of the examples of agent processing to be presented in this chapter.

5.1.2 Trajectory planning

The trajectory planning problem, a subproblem in robot manipulation systems design, requires the derivation of a path in space and time, satisfying various constraints, for an end-effector [14, 19]. One method for trajectory planning is to fit a polynomial to a given set of path constraints. An instance of this is considered in this subsection. The task is to calculate an interpolating polynomial, $th(t) = d + ct + bt^2 + at^3$, given that, $th(t0) = th0$, $th(t1) = th1$, $\partial th/\partial t(t0) = 0$, $\partial th/\partial t(t1) = 0$. The input task representation

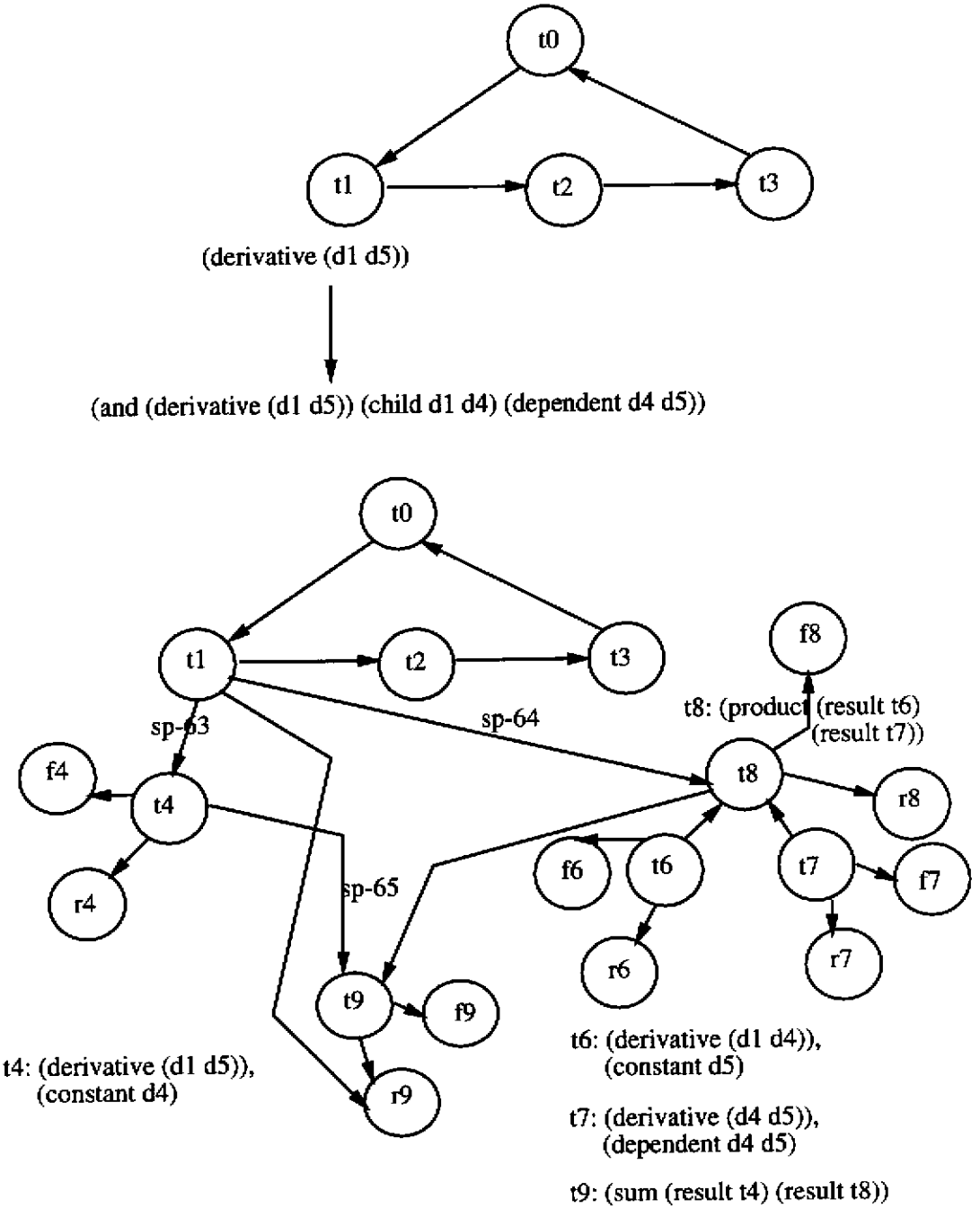


Figure 5.5: The agent's processing on a slightly different instance of sensitivity analysis.

consists of the following objects and facts:

- object(id: o1 type: goal purpose: (interpolation (o2 o7)) data: o2 o3 o4 o5 o6 o7)
- object(id: o2 type: data description: (interpolating-polynomial) (equation) (independent-variable o5) (dependent-variable o6) (name ($th = d + ct + bt^2 + at^3$)) child: o3 o4)
- object(id: o3 type: data parent: o2 description: (lhs) (name (th)))
- object(id: o4 type: data parent: o2 description: (rhs) (name ($d + ct + bt^2 + at^3$)))
- object(id: o5 type: data description: (independent-variable) (name (t)))
- object(id: o6 type: data description: (dependent-variable) (name (th)))
- object(id: o7 type: data description: (function-values)
  (if (value independent-variable t0) (value dependent-variable th0))
  (if (value independent-variable t1) (value dependent-variable th1))
  (if (value independent-variable t0) (value (derivative (dependent-variable independent-variable)) 0))
  (if (value independent-variable t1) (value (derivative (dependent-variable independent-variable)) 0)))
- f1(unknown a), f2(unknown b), f3(unknown c), f4(unknown d).

The following supervisory operators are required for the task:

- sp-51:

```
(If (and
        object(id: O1 type: calculation-graph-node
               description: (interpolation (O2 O3)))
        object(id: O3 type: data description:
        (if (value independent-variable X)
           (value dependent-variable Y))))
   (add
        object(id: O4 type: calculation-graph-node
               description:
```

(apply ((apply (O2 (substitute
(independent-variable X))))
(substitute (dependent-variable Y)))))))

- sp-52:
  (If (and
  object(id: O1 type: calculation-graph-node
  description: (interpolation (O2 O3)))
  object(id: O3 type: data description:
  (if (value independent-variable X) (value
  (derivative (dependent-variable independent-variable)) Y))))
  (add
  object(id: O4 type: calculation-graph-node
  description:
  (apply ((apply ((derivative (O2
  independent-variable)) (substitute
  ((derivative (dependent-variable
  independent-variable)) Y)))) (substitute
  (independent-variable X)))))))
- sp-53:
  (If (and
  object(id: O1 internal-action: C1 type: result
  description: (equation))
  object(id: O2 internal-action: C2 type: result
  description: (equation))
  object(id: O3 internal-action: C3 type: result
  description: (equation))
  object(id: O4 internal-action: C4 type: result
  description: (equation))
  F1(unknown X1)
  F2(unknown X2)
  F3(unknown X3)
  F4(unknown X4))
  (add
  object(id: O5 type: calculation-graph-node
  description: (solve-equations (result C1) (result C2)
  (result C3) (result C4)))))

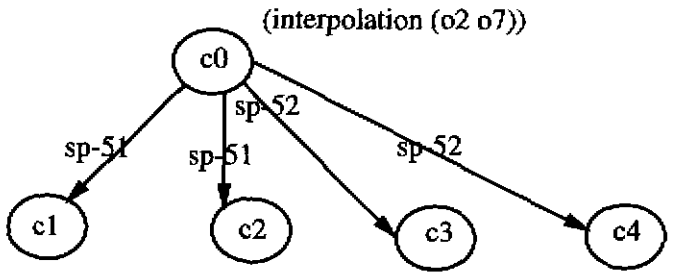
The processing is illustrated in Figures 5.6, 5.7, 5.8, and 5.9. As shown in Figure 5.6 c0 is the root node of the calculation graph for the task. It sets up the computation of the interpolation function for the task data. Two supervisory operators, sp-51 and sp-52, apply to c0. These operators essentially substitute the function values into the interpolating polynomial. The application of these operators creates four internal actions, c1, c2, c3, and c4. Figure 5.7 shows the processing for c1. The processing for c2 is similar to that for c1 and has not been illustrated. Figure 5.8 shows the processing for c3. The processing for c4 is similar to that for c2 and has not been shown. Once the four constraints have been substituted into the interpolating polynomial and four equations obtained, the supervisory operator, sp-53, applies. This operator represents the knowledge that if four equations are available for four unknowns then the values of the unknowns can be obtained by solving the equations simultaneously. The application of sp-53 creates the internal action c31. It is mapped onto the external action specification, f31. The execution of f31 produces the result r31 that provides the values for the unknowns in the interpolating polynomial as required by the input task.

5.1.3 Geometric reasoning

By geometric reasoning I mean simple analytic geometry [44, 48, 62, 29] problems. For example, given a set of points in the plane, distances among some of the points, and the coordinates of some of the points, find the coordinates of designated points in terms of known distances and coordinates. An instance of this problem is discussed in this subsection. The task is to calculate the coordinates of a point, P_4, given that (see Figure 5.10):

- $P1(x_1, y_1)$.
- $\mid P_2P_3 \mid = a$.
- $\mid P_1P_3 \mid = b$.
- $\mid P_1P_2 \mid = c$.
- $P_2P_4 \perp P_1P_3$.

The input task representation consists of the following objects and facts:



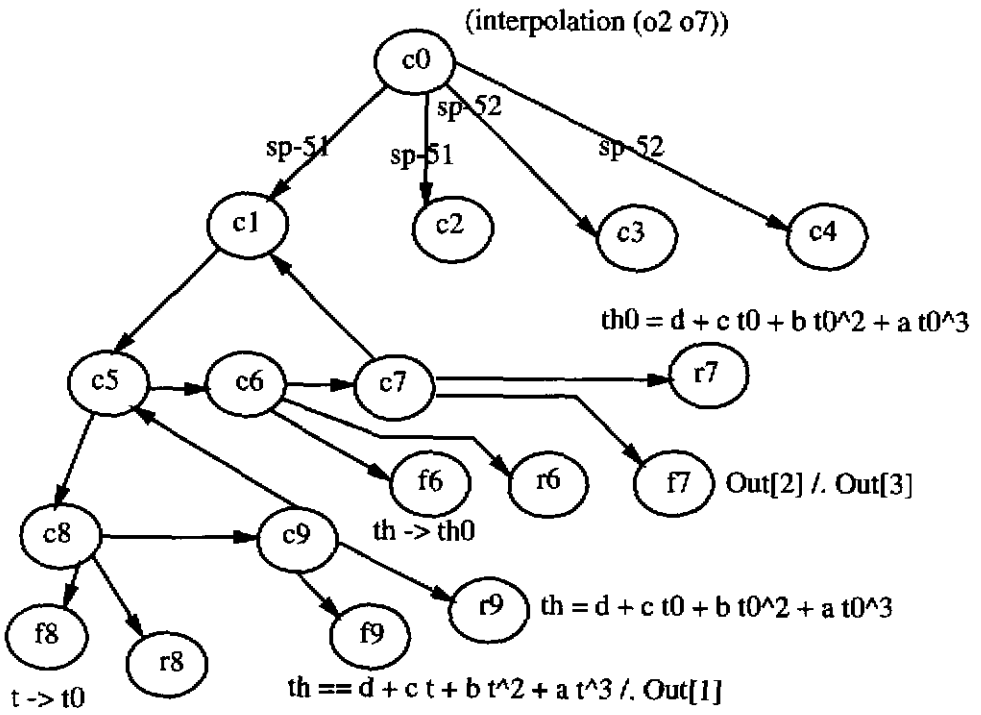
c1: (apply ((apply (o2 (substitute (independent-variable t0))))
(substitute (dependent-variable th0))))

c2: (apply ((apply (o2 (substitute (independent-variable t1))))
(substitute (dependent-variable th1))))

c3: (apply ((apply ((derivative (o2 independent-variable))
(substitute ((derivative (dependent-variable
independent-variable))
0))))
(substitute (independent-variable t0))))

c4: ((apply ((derivative (o2 independent-variable))
(substitute ((derivative (dependent-variable
independent-variable))
0))))
(substitute (independent-variable t1))))

Figure 5.6: The agent's processing for the trajectory planning instance.



c5: (apply (o2 (substitute (independent-variable t0)))

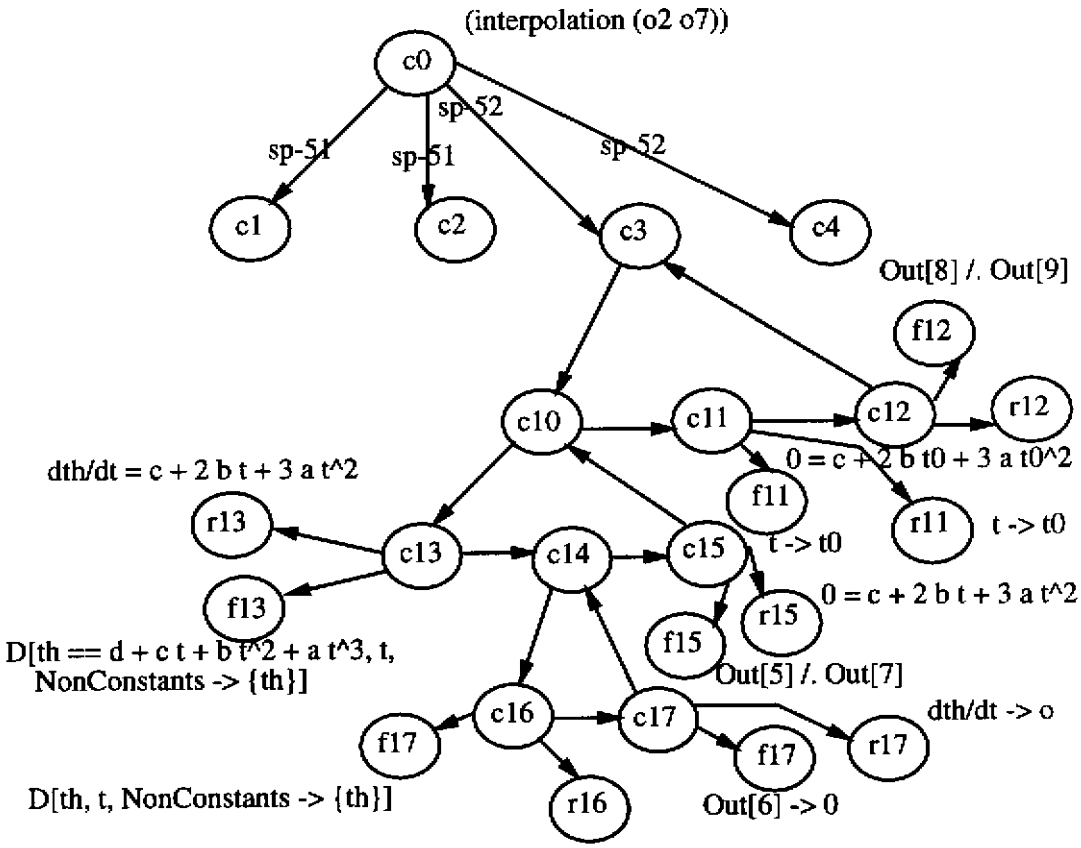
c6: (substitute (dependent-variable th0))

c7: (apply (result c5) (result c6))

c8: (substitute (independent-variable t0))

c9: (apply (o2 (result c8)))

Figure 5.7: The agent's processing for the trajectory planning instance.



c10: (apply ((derivative (o2 independent-variable))
(substitute ((derivative (dependent-variable
independent-variable))
0))))

c11: (substitute (independent-variable t0))

c12: (apply (result c10) (result c11))

c13: (derivative (o2 independent-variable))

c14: (substitute ((derivative (dependent-variable
independent-variable))
0))

c15: (apply (result c13) (result c14))

c16: (derivative (dependent-variable independent-variable))

c17: (substitute ((result c16) 0))

Figure 5.8: The agent's processing for the trajectory planning instance.

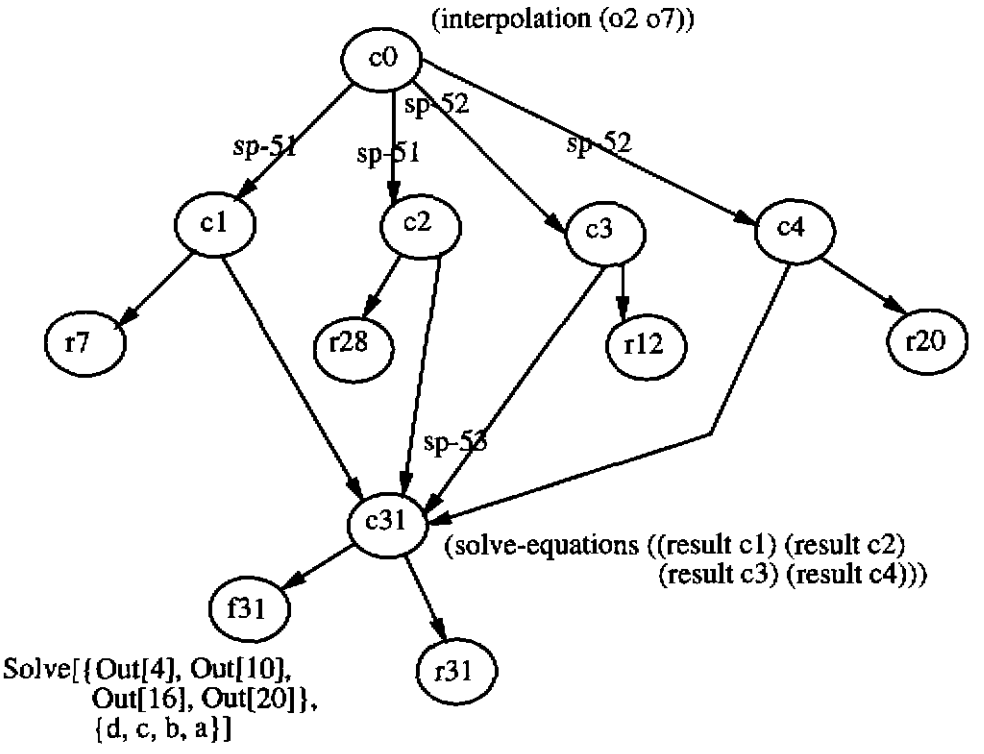


Figure 5.9: The agent's processing for the trajectory planning instance.

- object(id: o1 type: data purpose: (coordinates o5) data: o2 o3 o4 o5)
- object(id: o2 type: data description: (point) (x x1) (y y1) (name (p1)) (known x1) (known y1) (symbolic x1) (symbolic y1))
- object(id: o3 type: data description: (point) (x x2) (y y2) (name (p2)) (known x2) (known y2))
- object(id: o4 type: data description: (point) (name (p3)) (x x3) (y y3) (unknown x3) (unknown y3))
- object(id: o5 type: data description: (point) (name (p4)) (x x4) (y y4) (unknown x4) (unknown y4))
- f1(distance o2 o3 c)

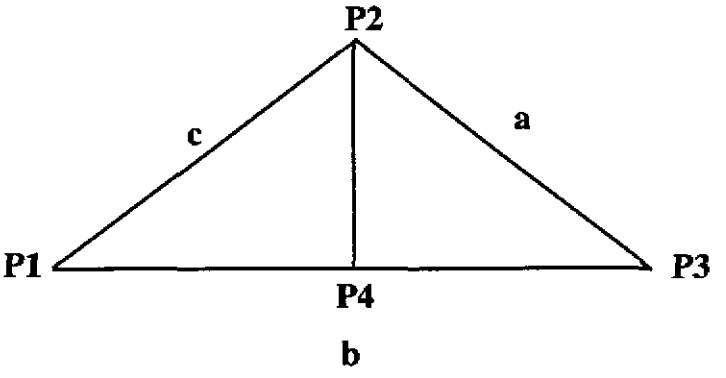


Figure 5.10: An instance of geometric reasoning.

- f2(known c)
- f3(distance o2 o4 b)
- f4(known b)
- f5(distance o3 o4 a)
- f6(known a)
- f7(perpendicular o3 o5 o2)
- f8(perpendicular o3 o5 o4)
- f9(collinear o2 o5 o4)

The following supervisory operators are required for the task:

- sp41: *If the coordinates of a point are known and the coordinates of another point are unknown and the distance between the points is known then write the distance equation for the two points.*

  (If (and
  object(id: O1 type: calculation-graph-node
  description: (coordinates (P1)))
  object(id: P1 description: {(point), (x X1), (y Y1)})
  object(id: P2 description: {(point), (x X2), (y, Y2)})
  fact(distance P1 P2 D1)

fact(known D1))
(add
object(id: O2 type: calculation-graph-node
description: (equal (R1 R2)))
object(id: R1 description: (name $((X1 - X2)^2 + (Y1 - Y2)^2)$))
object(id: R2 description: (name $(D1^2)$))
object(id: O1 continuation: O2)))

- sp42: *If there is a right-angle at the point whose coordinates are required and the sides of the containing triangle are known then apply the cosine rule.*

(If (and
fact(distance P1 P2 D3)
fact(distance P1 P3 D2)
fact(distance P2 P3 D1)
fact(perpendicular P1 P4 P2)
object(id: P1 description: {(name (N1)), (point), (x X1), (y Y1),
(known X1) (known Y1) })
object(id: P2 description: {(point), (name (N2))})
object(id: P3 description: {(point), (name (N3))})
object(id: P4 description: {(point) (x X4) (y Y4)
(unknown X4) (unknown Y4)})
object(id: O1 description: (coordinates (P4))))
(add
object(id: O2 description: (equal (R1 R2)))
object(id: R1 description: (name $(D1^2)$))
object(id: R2 description:
{(name $(D2^2 + D3^2 - 2D2D3$ cos[N1, N2, N3])),
(unknown cos[N1, N2, N3])})))

- sp43: *If there is an unknown element in an equation then solve for that element.*

(If (and
object(id: O1 description: (equal (R1 R2)))
object(id: R2 description: (unknown R3)))
(add
object(id: O2 description: (solve ((result O1) R3)))))

- sp44: *If P1P2 $\perp$ P3P2 and cos(P3, P1, P2) is known and | P1P2 | is known and the coordinates of P1 are known and the coordinates of P2 are unknown then the distance between P1 and P2 is equal to | P1P3 | cos(P3, P1, P2).*

  (If (and
  fact(perpendicular P1 P4 P2)
  fact(value cos[N1, N2, N3] V1)
  fact(distance P1 P2 D1)
  object(id: P1 description: {(point), (name (N1)), (x X1), (y Y1)})
  object(id: P2 description: {(point), (name (N2))})
  object(id: P3 description: {(point), (name (N3))}))
  object(id: P4 description: {(point), (name (N3)), (x X4), (y Y4)})
  (add
  object(id: O1 description: (equal (R1 R2)))
  object(id: R1 description: (name $((X1 - X4)^2 + (Y1 - Y4)^2)$))
  object(id: R2 description: (name $(D1V1)$))
  fact(distance P1 P4 (second (result O1)))
  fact(known (second (result O1)))))

- sp45: *If two sides of a right-angled triangle are known then compute the third if the third point is unknown.*

  (If (and
  (perpendicular P1 P2 P3)
  (distance P1 P2 D1)
  (distance P1 P3 D2)
  object(id: P1 description: {(point), (x X1), (y Y1),
  (known X1), (known Y1)})
  object(id: P3 description: {(point), (x X3), (y Y3),
  (known X3), (known Y3)})
  object(id: P2 description: {(point), (x X2), (y Y2),
  (unknown X2), (unknown Y2)}))
  (add
  object(id: T1 description: (equal (R1 R2)))
  object(id: R1 description: (name $((X3 - X2)^2 + (Y3 - Y2)^2)$))
  object(id: R2 description: (name $(D2^2 - D1^2)$)))))

- sp46: *If an unknown quadratic variable appears in two equations with two different variables and these two variables also appear in a third*

equation then isolate the unknown variable by subtracting the third equation from the first equation and then subtracting the difference from the second equation.

(If (and
object(id: E1 description: {(equation), (quadratic X1), (quadratic X2)} calculation: O1)
object(id: E2 description: {(equation), (quadratic X1), (quadratic X4)} calculation: O2)
object(id: E4 description: {(equation), (quadratic X2), (quadratic X4)} calculation: O3))
fact(unknown X4)
(add
object(id: O4
description: (difference ((result O2)
(difference ((result O1)
(result O3)))))))

- sp47: *If a line is perpendicular to another then compute the product of the slopes and set it equal to -1.*

(If (and
fact(perpendicular P1 P2 P3)
object(id: P1 description: {(point), (x X1), (y, Y1), (known X1), (known Y1)})
object(id: P3 description: {(point), (x X3), (y, Y3), (known X3), (known Y3)})
object(id: P2 description: {(point), (x X2), (y, Y2), (unknown X2), (unknown Y2)}))
(add
object(id: T1 description: (equal (R1 R2)))
object(id: R1 description:
(name $(((Y3 - Y2)/(X3 - X2))((Y1 - Y2)/(X1 - X2)))$)
object(id: R2 description: (name (-1)))))

- sp48: *If there is an equation containing two unknown quadratic variables and a linear equation among the same variables then solve the two equations simultaneously.*

(If (and
object(id: E1 description: {(equation), (linear X1), (linear X2)})

object(id: P1 description: {(unknown X1), (unknown X2)})
object(id: E2 description: {(equation), (quadratic X1),
(quadratic X2)}))

(add
object(id: T1 description: (solve ((E1 E2) (X1 X2))))))

The agent processing for this instance is illustrated in Figure 5.11. *sp-1* creates the root node of the calculation graph, t0: (coordinates o5). *sp41* adds t1: (equal $((x1-x2)^2+(y1-y2)^2)$ c^2), t2: (equal $((x1-x3)^2+(y1-y3)^2)$ (b^2)), and t3: (equal $((x2-x3)^2+(y2-y3)^2)$ (a^2)) to the calculation graph. t1, t2, and t3 are implemented by f1: $Equal[(x1-x2)^2+(y1-y2)^2, c^2]$, f2: $Equal[(x1-x3)^2+(y1-y3)^2, b^2]$, and f3: $Equal[(x2-x3)^2+(y2-y3)^2, a^2]$. The results are r1, r2, and r3. *sp42* adds t4: (equal (a^2) $(b^2+c^2-2bccos[p1,p2,p3])$) to the calculation graph. It is implemented by f4: $Equal[a^2, b^2+c^2-2bccos[p1,p2,p3]]$ and the result is r4. *sp43* adds t5: (solve (result t4) cos[p1, p2, p3]) to the calculation graph. It is implemented by f5: $Solve[Out[4], cos[p1,p2,p3]]$ and the result is r5. *sp44* adds t6: (equal $((x1-x4)^2+(y1-y4)^2)$ $(c((-a^2-b^2-c^2)/(2bc))^2)$) to the calculation graph. It is implemented by f6: $Equal[(x1-x4)^2+(y1-y4)^2, c((-a^2-b^2-c^2)/(2bc))^2]$ and the result is r6. *sp45* adds t7: (equal $((x2-x4)^2+(y2-y4)^2)$ (c^2 - (second (result t6)))) to the calculation graph. *sp46* adds t8: (difference ((result t6) (difference ((result t1) (result t2))))) to the calculation graph. *sp47* adds t9: (equal $(((y4-y1)/(x4-x1))((y2-y4)/(x2-x4)))$ -1) to the calculation graph. *sp48* adds t10: (solve-equations ((result t9) (result t1))). The result, r10, for t10 is recognized as the solution for t0 since the values for x4 and y4 determine the coordinates of P4. An interesting point to note is that sp46 represents an alternative strategy that does not prove useful in achieving the input goal. By acting on plausible internal actions the agent is able to synthesize the successful strategy represented by sp48.

5.2 An experiment with SignalProcessing

SignalProcessing [25] is a Mathematica package for signal processing applications. Although it is built atop Mathematica there is no reason not to consider it an independent software system. In this section agent processing

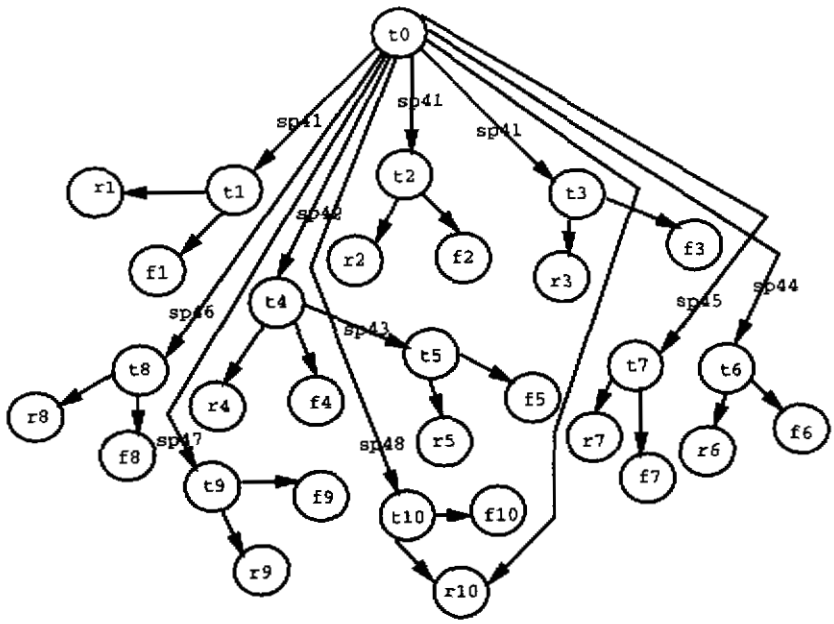


Figure 5.11: The processing for example of Figure 5.10

for a signal processing task using SignalProcessing is described.[3] The task is to compute the impulse response of the cascade of filters shown in Figure 5.12. The input task representation consists of the following objects and facts:

- object(id: o1 type: goal purpose: (impulse-response o2))
- object(id: o2 type: data description: linear-system child: o3 o4)
- object(id: o3 type: data parent: o2 description: (block) (system o5) (in o6) (out o7))
- object(id: o4 type: data parent: o2 description: (block) (system o8) (in o9) (out o10))
- object(id: o5 type: data description: (transfer-function) (unit-step))

[3]The processing described in this section has not been implemented. However, the description provided is sufficiently indepth to make an implementation straightforward.

- object(id: o8 type: data description: (transfer-function) (unit-step))
- object(id: o6 type: data description: system-input)
- object(id: o7 type: data description: (to o9))
- object(id: o9 type: data description: (from o7))
- object(id: o10 type: data description: (system-output))
- f1(cascade o1 (o2 o3))

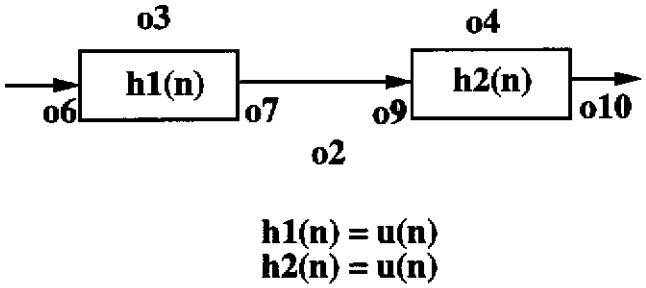


Figure 5.12: A signal processing system configuration.

The supervisory operators required for the task are:

- sp-91:
  (if (and
  object(id: O1 description: (impulse-response (O2)))
  fact(cascase O2 (O3 O4)))
  (add
  object(id: O5 description: (convolution (O3 O4)))))
- sp-92:
  (if (and
  object(id: O1 description: (convolution (O2 O3))))
  (add
  object(id: O1
  description: (product ((z-transform (O2))

(z-transform (O3))))
post-condition: (inverse-z-transform ((result O1))))))

- sp-93
  (if(and
  object(id: O1 description: (impulse-response *)))
  (add
  object(id: O2 description: (confirm ((result O1))))))
- sp-94
  (if (and
  object(id: O1 description: (product ((z-transform (*))
  (z-transform (*))))))
  (and
  object(id: O2 description: (region-of-convergence
  ((result O1))))))
- sp-95
  (if (and
  object(id: O1 description: (confirm (result O2)))
  object(id: O2 result: (result O3))
  object(id: O3 description: (F (result O4)))
  fact(inverse F G))
  (add
  object(id: O5 description: (G (result O2)))
  object(id: O6 description: (compare (result O5)
  (result O2)))))

The processing on the signal processing example is illustrated in Figure 5.13. As shown in the figure sp-91 is the strategy used to process the input task. The convolution is calculated by t2, t3, and t4. sp-94 applies to create the internal action t7 to find the region of convergence for t1. Now, the post-condition calculation for t1 is instantiated as the internal action t5. The result for t5 is the result for t1. The result for t1 is also the result for t0. At this point sp-93 applies to create the internal action t6 to confirm the result of t0. sp-95 is a strategy that tries to confirm a result by applying the inverse of the function that produced the result. It creates the internal actions t8 and t9. A consequence of executing t8 by f8 is that a result for t7 is obtained as well. The reason is that an inverse z-transform calculation generates the region of convergence information as well. Thus, the heuristic

of acting upon plausible internal actions can lead to the synthesis of new strategies.

5.3 An experiment with FrameKit

In this section an experiment with a FrameKit [59]-based database system is described. A simple object-oriented database of Carnegie Mellon faculty members in Computer Science was created in FrameKit. The agent's task is to find professors interested in software engineering. The input task representation consists of the following objects:

1. object(id: o10 type: goal purpose: (find (o1))).

2. object(id: o1 type: data description: (rank o1 professor) (faculty cmu-cs o1) (interest o1 se)).

The agent processing on the task is shown in Figure 5.14. t0 is the calculation graph root node. Its function application is, (find (o1)), where o1 is a data object whose description consists of the three properties that objects to be retrieved must have. A retrieval strategy is to retrieve the values of that attribute of an object in the database that is a predicate on the object sought. For example, a predicate on o1 is, (faculty cmu-cs o1). Knowing that cmu-cs is an object in the database, a strategy is to retrieve values of the attribute, faculty, of cmu-cs. sp-21 is the supervisory operator that represents this retrieval strategy. sp-21 applies to add to t0's description the function application, (retrieve o1 faculty). Using the knowledge that values of a database object's attribute can be retrieved using the FrameKit function, `get-values`, the external action specification, f0, is obtained. The execution results produced for f0, r0, include a set of objects that represent faculty members. However, result objects do not satisfy the other properties that the objects required by t0 were supposed to have. For example, the result objects do not have the property giving the faculty member's rank as part of their description. Consequently, the result r0 is considered to have failed. Next, another attempt is made to map t0 into an external action specification. The reason for this second attempt is that the agent's supervisory process tries to have the formulation process plan an external action specification

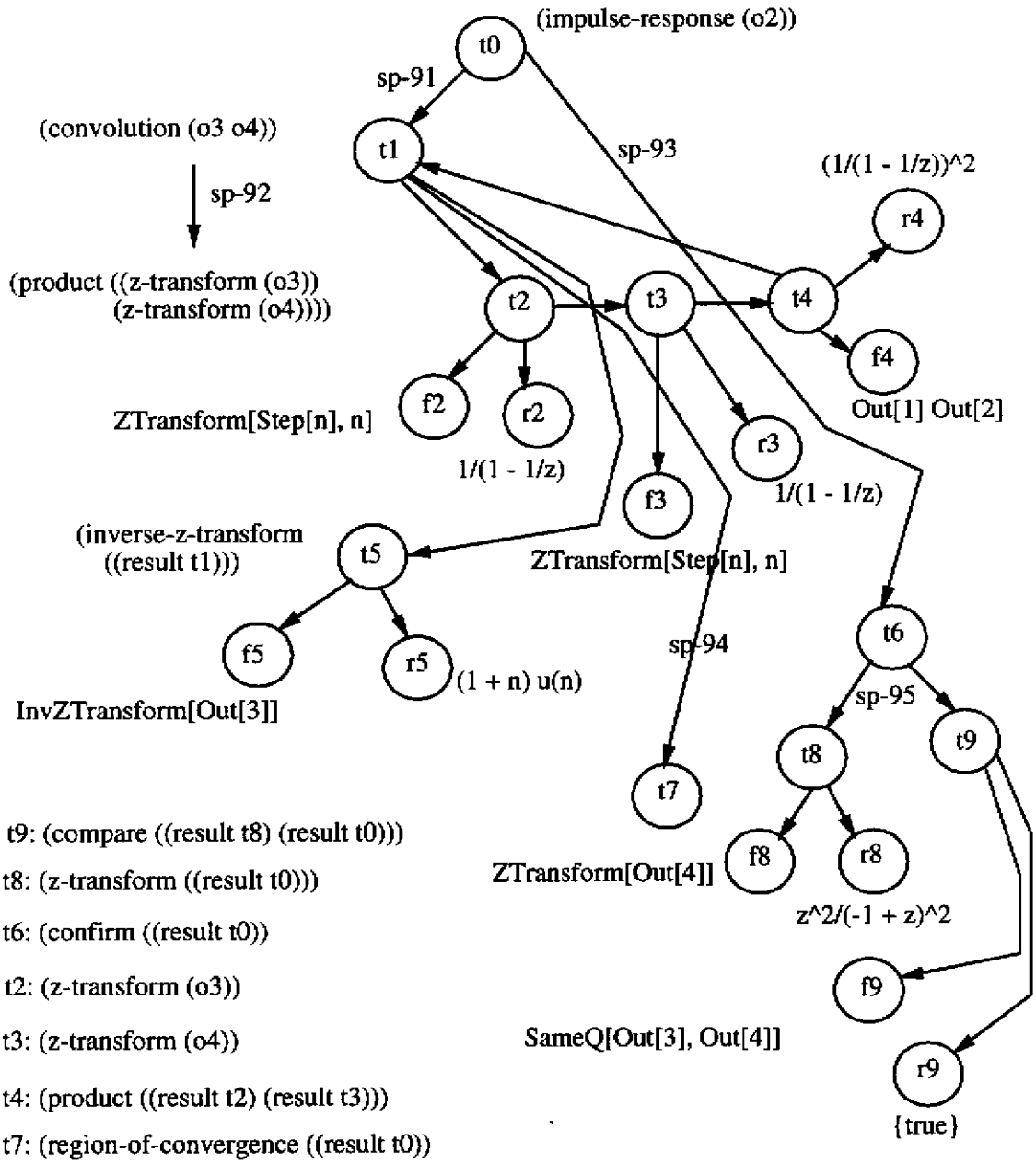


Figure 5.13: The agent's processing for the signal processing example.

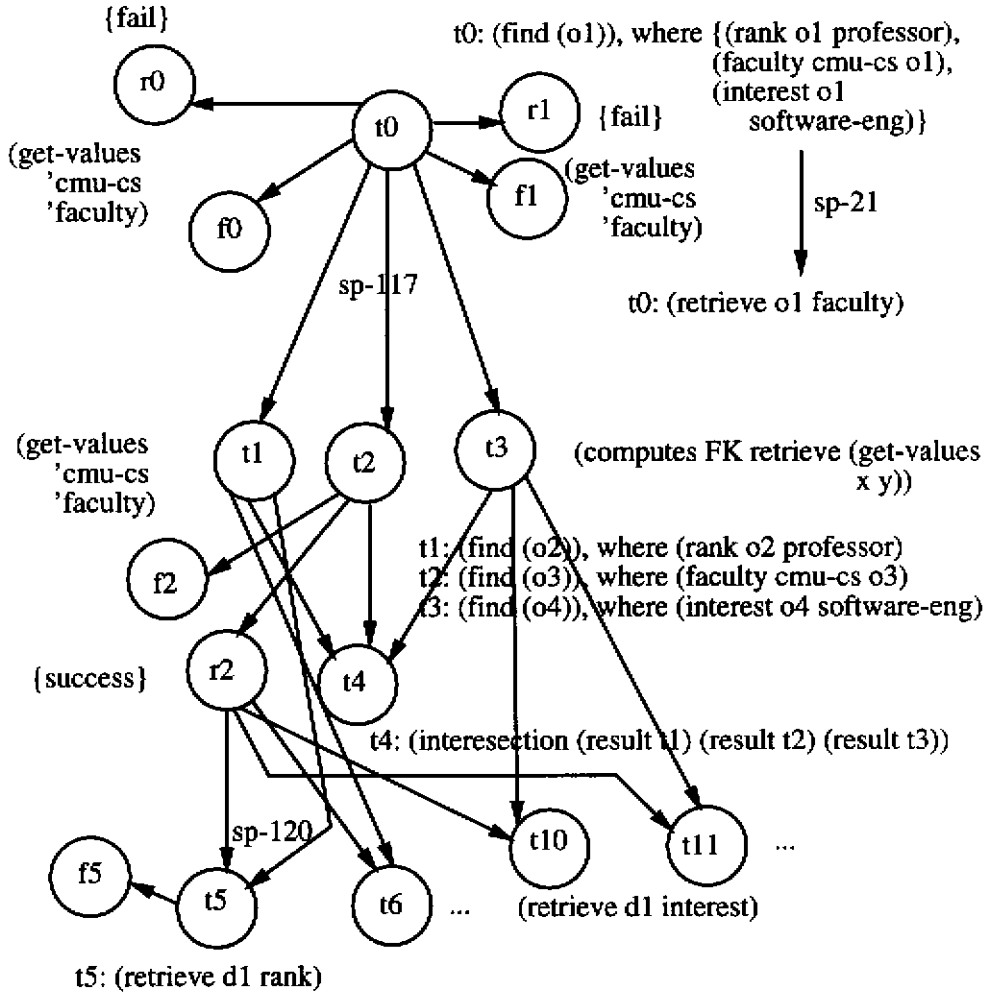


Figure 5.14: The agent's processing on the FrameKit task.

even if no direct knowledge of the feasibility of such planning is available. Once again there is a failure in the execution result.

At this point, an alternative strategy is applied. The strategy is to create separate internal actions to find objects that separately satisfy the properties that the objects required in t0 must have. The results of these internal actions are then intersected in order to obtain objects that jointly satisfy all the properties. This strategy is represented by the supervisory operator, sp-117. Its application creates the internal actions, t1, t2, t3, and t4. t2 is now mapped onto the external action, f2. The execution result, r2, for f2 is considered a success because the retrieved objects satisfy the property that the objects required by t2 must have. Another retrieval strategy is to retrieve the value of that attribute of an object that is a predicate on an object to be found. This strategy is simply a variant on the strategy of sp-117. For example, the predicate on o2 is, (rank o2 professor), and the data objects belonging to the result, r2, are known to have the attribute, *rank*. Therefore, an internal action can be created to retrieve the value of the *rank* attribute of a data object of r2. This strategy is represented by the supervisory operator, sp-120. It applies to create internal actions, such as t5 and t6, for each of the data objects of r2. The results for some of these internal actions are successful since the rank matches the sought after rank. These objects are included as the result for t1. In an exactly similar manner, the application of sp-120, on the data objects of r2 for t3's function produces internal actions such as t10 and t11. The results for some of these internal actions are successful since the interest matches the sought after interest. These objects are included as the result for t3. With the results for t1, t2, and t3, available, a supervisory operator adds to the result for t4 those objects that are common to the result of t1, t2, and t3.

5.4 Evaluation of agent design

The basic design of the agent has been validated by the implementation of several tasks that require the use of Mathematica. The processing on these tasks was described in Section 5.1 of this chapter. To assess the generality of the agent design we applied it a task each for SignalProcessing and FrameKit. The processing on these tasks was described in Sections 5.2 and 5.3. In this section, an evaluation of the agent design is presented.

The adequacy of the agent design for using Mathematica is discussed first. The agent consists of a domain-independent organization and a set of task-specific rules that enable performance of symbolic calculations using Mathematica. A blackboard-style control structure, implemented within a forward-chaining production systems framework, is used to organize processing activity in the agent. There are six problem solving processes consisting of a set of domain-independent operators as detailed in Chapter 3. These operators determine which internal action to focus on, when to remove an internal action from the focus, how to refine abstract internal actions into simpler internal action, whether to plan an external action specification for an internal action, propagation of results to abstract internal actions, etc. A calculation graph structure is maintained on the blackboard, a subset of whose nodes represent the task plan and another subset represents the software system plan sequence for the input task. The domain-independent operators and the basic control structure have remained invariant as we have gone from the implementation of one task instance to the next. This suggests some degree of task-independence in the design of the agent. The current implementation is a reasonable prototype of an agent for using Mathematica. Another bit of evidence to sustain this assertion is that the increments to the agent structure have been confined more and more to the domain-specific supervisory operators as we have implemented a succession of tasks.

Approximately 50 Mathematica functions have been represented. To represent a single Mathematica function takes on the order of one hour. Let us consider the representation of the Mathematica function, `SameQ`. Two types of knowledge about this function can be represented. First, there is knowledge that associates an internal action description with the function. For example, (computes mma equal `SameQ`), associates the internal function, *equal*, with `SameQ`. A sense of what associations to represent for a given Mathematica function is obtained by analyzing tasks that are likely to occur and that could be performed using the function. The second representation for a function is independent of the tasks for which the function will be used. For `SameQ` it took about 10 minutes to specify the following representation: `SameQ` [75, page 228]:

object:
    id: o5
    type: mma-fact

action:
(if (equal x y) (output *true*))
(if (not (equal x y)) (output false))
input-type: algebraic-expression
output-type: boolean
name: ($\lambda x \lambda y$ SameQ x y)

The effort required in this case was minimal because the function is quite similar to `Equal` for which a declarative representation had been devised previously. For a completely new function it will take longer to specify the representation. However, the upper bound on the effort required is less than one hour. General guidelines for the declarative representation of a software system functions are obtained from deduction-based program synthesis work [42].

The Mathematica functions that are represented currently are capable of supporting substantial engineering problem solving. It is estimated that the number of additional Mathematica functions that might be useful to represent for the three domains of sensitivity analysis, trajectory planning, and geometric reasoning is limited to less than twenty. Of course, for extending the agent for other tasks, additional Mathematica functions might be found useful to represent. But, in any event, such scaling up seems quite practical. There is one additional consideration involved in representing knowledge of Mathematica functions. The (internal) functions and predicates used in the representation of a Mathematica function (for example, *equal* and *output*, for `SameQ`) must be ones that the agent's other knowledge can interpret. So, if we used *equality* instead of *equal*, the interpretation machinery for *equal* will not apply. A non-trivial effort may be required to ensure a proper fit of newly represented knowledge with already available knowledge. Other than the generic principle of exercising caution while representing knowledge we do not have a way to lessen the possibility of incurring such overhead.

The task instances discussed in Section 5.1 are reasonably prototypical of engineering calculations for which Mathematica is a natural candidate software system. Our experience with these tasks suggests that the representation of task knowledge used in the agent design is adequate. The represented task knowledge was obtained from textbooks. Much of the task knowledge is represented procedurally by supervisory operators. The declarative task knowledge consists mainly of definitions. To represent a useful body of task

knowledge is a bounded endeavor. For example, in the case of the geometric reasoning task, discussed in Subsection 5.1.3, we find that less than 10 supervisory operators are needed for that problem class. Of course, this is still a fairly simple problem class and it may be that the moment we consider a more challenging problem class the task knowledge requirements will explode. From our experimentation we can only assert that for the tasks considered the task knowledge requirements have not emerged as a bottleneck. It is an interesting aspect of engineering calculations that the background knowledge is that of mathematics. The mathematical knowledge that underlies various tasks need not be duplicated. For example, solving equations in both geometric reasoning and trajectory planning is done by the same body of knowledge.

It may be noted that engineering calculations are not only a matter of mathematical (formal) manipulations but also require substantial non-formal knowledge [45, 7]. To make the agent really useful to engineers some of this knowledge may have to be represented as well. It is an open question to what extent this will prove doable. As it stands, the prototype agent is most promising for removing the tedium out of symbolic manipulations that are part of engineering design by automatically accessing the functionality of Mathematica.

The extent to which the agent can itself acquire new knowledge the better its chances for practical realization become. As detailed in Chapter 4, several kinds of deliberate and reflective knowledge acquisition have been demonstrated for the agent. Automatic knowledge acquisition follows from successful problem solving by the agent and this further suggests that the basic processing organization of the agent is adequate.

5.4.1 Generality of agent design

The experience of applying the agent, designed for Mathematica, to other software systems, SignalProcessing and FrameKit, suggests that the basic structure of the agent remains invariant attesting to its generality. The nature of the task knowledge changes sharply in going from Mathematica to FrameKit. The biggest change is the formality of mathematical knowledge in contrast to the commonsense [21, 34] knowledge used in database retrieval. However, examining the processing for a task with a given software system, say FrameKit, we continue to find that there are domain-specific supervisory

operators acting with the domain-independent process operators, identified in the agent for Mathematica. The agent's representation of each software system continues to consist of a set of functions executable by the software system. Given an input task representation the agent creates internal actions appropriate for achieving the task by applying relevant task knowledge. A subset of these internal actions are then mapped onto external action specifications to be executed by the software system. This is true for each of the software systems considered.

5.4.2 Using Soar for agent implementation

As a generic problem solving system Soar comes with far more structure than say Lisp or Prolog. It is this structure, to the extent that it is useful, that argues for the use of an architecture rather than a programming language to instantiate an agent design. Another motivation to use Soar is provided by the proposal to augment a general problem solver's capability by having it automatically use existing software systems as tools [54]. However, the agent design presented in this dissertation must not be confounded with Soar. The two are separate. Soar has been used as the substrate for the implementation of the agent. The difference between Soar and the agent can be seen by noting that Soar computations may not be a part of the agent design, for example, a no-change impasse for a problem space slot of a goal context. Similarly, the restriction to a flat hierarchy of problem spaces is not necessarily a part of the Soar design.

Many of the basic mechanisms of the Soar architecture have been found useful for the agent implementation. The multiple problem space organization of Soar is useful for system modularity. The various problem solving processes that were considered sufficient for the agent design were directly mapped onto problem spaces in Soar. The development of each process could proceed independently of others by the addition of operators to the corresponding problem space. The multiple problem space organization also enabled straight-forward implementation of the dual-space planning function considered necessary for the agent design. Soar's decision procedure enables flexible control strategies to be easily implemented. For example, consider the supervisory operators for calculation graph initialization. There are three different operators for the initialization. The first operator applies if there is a definition for the input task. The second operator applies if the input

task is known to be computable function, such as factor. The third operator applies if there is knowledge of Mathematica's ability to compute the input task. In some instances all three realizations will be applicable. By using appropriate preference rules it is possible to set up various control strategies such as selecting to use the first operator over the second but selecting the third over the other two. Additionally, the separation of preference processing from operator application simplifies system development since either can be changed independent of the other. Finally, the provision of lookahead search and chunking enable automatic acquisition of control knowledge and this also simplifies system development as less control knowledge may be supplied manually. Knowledge representation in Soar is in terms of production rules. This was found a particularly convenient knowledge representation technique since direct guidance from protocols of human users of Mathematica was being used to guide system design. The knowledge evidently used in the protocols could be mapped onto sets of rules and represented by operators in Soar.

Some problems with using Soar as the implementation medium were also identified. From a purely software technology point of view a fixed architecture can imply the overhead of fashioning solutions to conform to the provided mechanisms. For example, processing list representations in Soar proved to be arduous. In many instances where a simple Lisp function could suffice, complex sets of productions were required. Of course, one reason for the fixedness of an architecture's mechanisms is to enable all mechanisms to coexist. Arbitrary calls to Lisp functions can well disturb the integrity of chunking.

Chapter 6

The agent abstraction

In this chapter a description of the agent is provided by abstracting away from the specifics of the task and the software system used by the agent to perform the task. First, let us review the general parameters of an agent for automatically using a software system. Such an agent is illustrated in Figure 6.1. A user provides a task specification to the agent. The agent performs the task and returns the results of the task to the user. Internally, the agent makes use of a software system to assist in the performance of the input task. To use the software system the agent maps the input task onto the software system functions. In this dissertation, we are primarily concerned with the automatic use of a software system by an agent. Substantive considerations relating to the interaction between such an agent and a user have not been addressed in this dissertation. The agent design is determined by the type of agent/software-system interaction considered. We have considered the design of an agent for incremental task performance by interactive use of a software system. The two defining properties of such an interaction are:

1. *Significant computation* by individual software system functions.

2. *Modest internal processing* for an internal action.

The first property ensures exclusion of the case where the software system is a programming language. Examples of software systems that satisfy the first property include CASs, databases, and statistical packages. The second property ensures exclusion of the case where an internal action requires substantial processing by the agent. For example, consider a database retrieval

task where a query returns 100 items each of which must be checked for a given property. Assuming there is no function provided by the database interface for this filtering the agent must process each item individually. Already the internal action seems to require significant internal processing by the agent. However, if the number of items retrieved is 1000, then the agent processing required may well be too inefficient. If the database interface is programmable then such an internal action is the point where a new function must be assembled from more primitive functions. We have not considered the efficiency-driven synthesis of new software system functions by the agent.

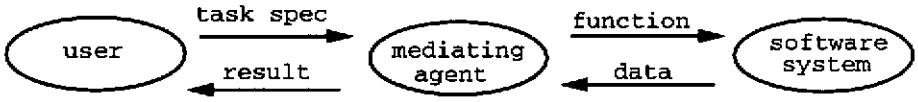


Figure 6.1: A mediating agent.

6.1 Agent specification

In this section a specification of the agent is provided.

- $Agent = \bigcup_j s_j, S, F, C.$
- s_j: the j^{th} software system used by the agent.
- $s_j = \{f_{j1}, ..., f_{ji}, ...\}$.
  f_{ji} : agent representation of i^{th} executable function of s_j.
- *task_spec* is object(id: O1 type: goal purpose: ($function < arguments >$) data: D1 ...). D_i are data objects for the task specification. A data object has a description. A description is a conjunction of sentences.
- *task_graph*: a node is either a $task_spec$ or a $task_mp$. Links represent temporal and causal ordering among the nodes.
- $task\_spec \equiv \{(g < arguments >), ..., (p < arguments >)\}$. g is a function name. p is a predicate name. Arguments can be objects, descriptions of objects, or symbols denoting themselves.

- $task\_imp \equiv (f_{ji}.name < arguments >)$. $f_{ji}.name$ is the name of f_{ji}.
- $task\_rep \equiv task\_graph, state$
- $state = \{D_1, ..., R_1, ..., K1, ...\}$. R_i are the result objects. A result object corresponds to the data obtained from the software system by a function execution. K_i are facts. A fact is a sentence.
- S: see Subsection 6.1.1.
- $F : (task\_spec, task\_rep) \rightarrow task\_imp$.

  F is one to many. (ie. for the same implementation there may be a number of implementations.)

  F is partial. Assume heuristic sufficiency for F. (ie. F applies to a sufficient number of *task_specs*.)
- C: see Subsection 6.1.2.

6.1.1 The S function

The notation used is from [65]:

- $input\_spec \rightarrow$ S.task_graph_root.O(type: task_graph_node description: input_spec.purpose).
- $g_i \rightarrow g_j$. g_i is rewritten into g_j.
- $g_1, A(g_1(.)) \rightarrow g_1; h_1(g_1)(h_2(g_1(.)))$.
- $h_1(g_1, ..., g_i) = g_j, 1 \leq j \leq n$.
- $h_2(g_1(.), ..., g_i(.)) = perm(g_1(.), ..., g_i(.)) \mid subsequence(g_1(.), ..., g_i(.))$.
- $g_1 \| ... \| g_i, A(g_1(.), ..., g_i(.)) \rightarrow g_1 \| ... \| g_i; h_1(g_1, ..., g_i)(h_2(g_1(.), ..., g_i(.)))$.
- $g_1, A(g_1(.)) \rightarrow g_1; h_1(g_1)(h_2(g_1(.))); ...; h_1(g_1)(h_2(g_1(.)))$.
- $g_1; ...; g_i, A(g_1(.), ..., g_i) \rightarrow g_1; h_1(g_1, ..., g_i)(h_2(g_1(.), ..., g_i(.)))$.
- $g_{11}; ...; g_{1i} \| g_{21} ... \| g_{2j}, A(g_1(.), ..., g_i) \rightarrow$
  $g_1; h_1(g_{11}, ..., g_{1i}, g_{21}, ..., g_{2j})(h_2(g_{11}(.), ..., g_{1i}(.), g_{21}(.), ..., g_{2j}(.)))$.

A: A propositional formula.

6.1.2 The C function

- $(task\_rep, S, F, IO) \rightarrow (g_j, SorForIO)$.
- $(g_i, S_1, ..., S_j) \rightarrow S_k, 1 \leq k \leq j$.
- $(g_i, F_1, ..., F_j) \rightarrow F_k, 1 \leq k \leq j$.
- $IO : task\_imp \rightarrow transmit(task\_imp, s_j) \cup evaluation(task\_imp, s_j) \rightarrow task\_rep$.

6.2 S_algorithms

An S_algorithm is a task graph whose nodes are *task_specs*. An S_algorithm is generated by S. Since S is partial, S_algorithms can be generated for only a subset of the input tasks given to the agent. An S_algorithm is like a flowchart program [76]. S_algorithms are generated by piecing together the following units:

1. Sequential (Figure 6.2).
2. Conditional (Figure 6.3):
   If (true t_i.result.(P ...)) then t_j else t_k
3. Iteration (Figure 6.4):
   (and (false t_i.result.(P_i ...)) ... (false t_{k-1}.result.(P_{k-1})) (true t_k.result.(P_k)))
4. Partial ordering (Figures 6.5 and 6.6).
5. Refinement (Figure 6.7).
6. Decomposition (Figure 6.8).

The syntax of S_algorithms is given by the following grammar:

- $< sequence >::=< spec > | < sequence >; < spec >$
- $< spec >::= F \mid < sequence >; < par >$
- $< par >::=< sequence > \parallel < sequence > | < sequence > \parallel < par >$

- $< spec >::= (F; P \rightarrow (< sequence >, < sequence >))^i [ifinite]$
- $< par >::=< par >; < sequence >$
- $F ::= (< function >< data >)$
- $< data >::= (O_1.name, ..., O_i.name)$ [i finite]
- $< spec >::=< refinement > | < decomposition >$
- $< refinement >::=< sequence >$
- $< decomposition >::=< par >$
- $< S\_algorithm >::=< sequence >$

Figure 6.2: The task_graph of a sequential computation.

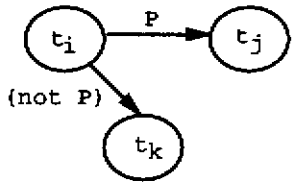


Figure 6.3: The task_graph of a conditional computation.

An S_algorithm computes a multivariate, multi-valued function from the data objects contained in the input task specification to the result objects produced as a solution for the input task. The executable S_algorithms are a subset of the specifiable S_algorithms. An executable S_algorithm is one whose task_specs can be either internally or externally evaluated. A program for an S_algorithm is the subgraph of the task_graph whose nodes are task_imps that were evaluated externally using the software system. Termination condition is a predicate on the input data objects and the results produced by the evaluation of task_imps. Refinement and decomposition rules allow for the specification of abstract algorithms.

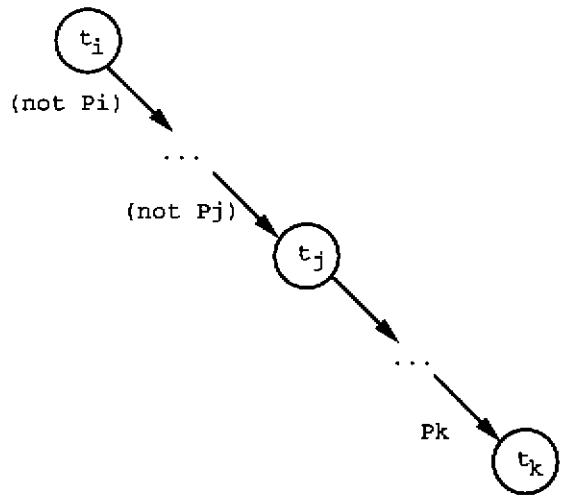


Figure 6.4: The task_graph of an iterative computation.

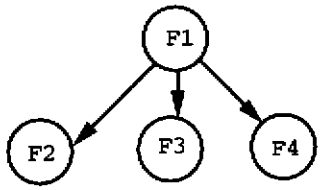


Figure 6.5: The task_graph of a partially ordered computation.

6.3 Discussion

- Algorithm synthesis: $(input\_spec, S, C) \rightarrow S\_Algorithm$.
- Program synthesis: $S\_algorithm \rightarrow ((f_{j1}.name < arg_1 >), ..., (f_{jn}.name < arg_n >))$.
- The class of algorithms specifiable: S_algorithms. The specifiable S_algorithms are those that could conceivably by generated by some S function.
- The class of algorithms synthesizable: A subset of S_algorithms. This is the class of S_algorithms that can actually be synthesized by the actual S function of the agent.

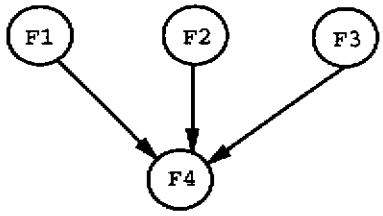


Figure 6.6: The task_graph of a partially ordered computation.

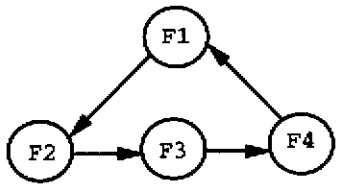


Figure 6.7: The task_graph of a refinement.

- The class of functions specifiable: Functions computed by the specifiable S_algorithms.
- The class of functions synthesizable: Functions computed by the synthesizable S_algorithms.
- The class of functions computable: Functions computed by programs.
- The class of programs implementable: S_algorithms that have corresponding programs.
- The software system is used to evaluate a function that the agent could not evaluate internally. The agent synthesizes an S_algorithm for the

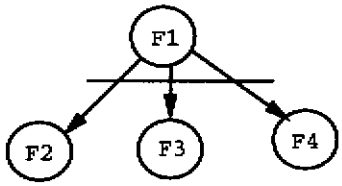


Figure 6.8: The task_graph of a decomposition.

input task and a program to implement the S_algorithm.

- Inability to perform a given task may be attributable either to S or F or C.
- The issue is not only *how to use a software system* but also *how to perform a task using a software system.* The agent must bring together together task knowledge and software system knowledge to perform the input task.
- The labor of automatic software use by an agent is the determination of a way to perform the task using accessible software system functionality.

Chapter 7

Related work

In this chapter some connections with research in intelligent interfaces, software reuse, automated deduction, and automatic programming are discussed.

7.1 Intelligent interfaces

Intelligent user interfaces [70, 22] provide an abstraction for the underlying application software. In contrast, our approach is to build a generic agent capable of using software systems and then customize it for specific software systems by adding knowledge of their functionality. Such an approach is capable of a more flexible access to complex software systems. The knowledge of the tasks and a capacity to perform these tasks is independent from the particular software system needed to be used. Instead of trying to simplify the use of the underlying software system we advocate complete hiding of the software system complexity by having the agent do all of the problem solving needed for the use of the software system. Intelligent agents that mediate in the use of complex information systems have been proposed by [71]. We have focussed specifically on the automatic use of a single software system. Many of the issues raised in [71] will be part of our future agenda.

The UNIX consultant (UC) [73] provides intelligent mediation between a user and UNIX. UC offers a treatment of both the primary functions making up a mediating agent that were described in Chapter 6 with major emphasis placed on the user/agent interaction while our work on agent design is primarily concerned with the agent/software-system interaction. Therefore, the

language analysis component that "computes a representation of the content of an utterance" [74, pages 4], the goal analysis component to "hypothesize the plans and goals under which the user is operating" [74, pages 5], the user modeling component that assesses "the knowledge state of the user" [74, pages 6], and the components that give determine the natural language output for the user have no counterparts in our agent design. These components will be found useful in any future extension of the agent design.

UC has two planners, the agent and the domain planner, and our agent design has two planners as well, the supervisory and the formulation processes. However, there are substantive differences in the planning functions in the two cases. In the UC, the agent "reacts to users' requests by forming goals and acting on them" [74, page 6] while the the supervisory process is concerned with planning required by the input task as the user goals are assumed directly available. Given a goal by the UC agent the UC domain planner "tries to determine how to accomplish this task, using knowledge about UNIX and knowledge about the user's likely goals" [74, page 7] and is an amalgamation of the planning performed by the supervisory and formulation processes. The UC does not have an execution model associated with it. As we have seen our agent design tightly integrates the planning and execution required in using a software system. UC learning mechanism is concerned with the extraction of UNIX facts and knowledge of English vocabulary from user input which contrasts with the reliance on agent activity alone for new knowledge acquisition in our agent design.

7.2 Automated deduction

Many features of natural deduction theorem-proving systems [10, 11, 12] were found useful in the agent design for Mathematica. Rewrite rules provide a convenient representation for task definitions. Compound calculations are refined into simpler calculations using splitting rules. Object-oriented knowledge representation was useful in the encapsulation of related facts for efficient access. Task-specific inference rules were used to represent task knowledge. Finally, a forward-chaining control strategy was found useful in organizing the agent's processing of symbolic calculation tasks.

7.3 Software reuse

Software reuse is defined as [9, page xv] the reapplication of code that was produced for some application to a current application. One form of software reuse occurs when an agent automatically uses the existing functionality of a software system. There are many software systems that can be considered to be a collection of functions. In this dissertation we have focused on one such software system, namely, a CAS. Other examples include databases, word-processors, and image processing libraries. When user interactively uses such a system to perform a task there is reuse of the software system functions. Our agent provides an automatic means for the reuse of software system functions for task instances provided by the user. The proposal to build reusable software components does not usually come with the proposal to automatically use these components. The provision for an automatic use of reusable software components represents a point of departure for the agent described in this dissertation. This result must be viewed in the limited context of an automatic reuse of code libraries. Since there are now in existence software systems that can be viewed as reusable code libraries, our agent design is one way of allowing for the automatic reuse of such code libraries.

7.4 Automatic programming

In knowledge-based transformation systems [4, 2, 17, 3, 60, 6, 8, 5, 36] for automatic programming, a high-level language program is incrementally converted into a target language program by applying a sequence of transformation rules that encode programming knowledge including how to implement algorithms and data structures and optimize code. Each rule takes a part of a program and replaces it with a new (transformed) part [63]. Rules are typically correctness-preserving. Since the starting point is a program, a specification of the essential computation has already been provided by the user to begin with. The initial program is an abstract computation because it is not directly implementable in the target computational system. So, the problem is to refine the initial program into computations that are less abstract (vertical transformations). Refinements do not change the basic computation to be performed but specify how to perform a computation in terms of other less abstract computations. A refinement also specifies the control flow among

the new computations that it introduces. A refinement is successful if it eventually leads to computations that are directly implementable in the target computational system. Other rules are meant to modify the control flow and the data structure design (lateral transformations) in partially transformed programs. What changes here is not the computation per se but rather the manner in which the computation is performed in the target computational system. A transformation system explores the space of partially transformed programs.

However, the real starting point for any programming activity is a task specification. The task specification sets up a requirement for what ought to be computed. The programmer then engages in two distinct but interrelated activities called algorithm design and program synthesis [37]. Our approach to automatic programming is patterned after such a state of affairs. An agent accepts an input task specification from the user and proceeds to determine computations required by the input task and performing these computations using the functions of a given software system in order to furnish a result for the input task to the user. The agent's processing concerned with the determination of computations required for the task corresponds to algorithm design while the processing concerned with using the software system to perform these computations corresponds to program synthesis. The agent comes with no completeness guarantees and may fail to produce a result for certain inputs with failure attributable to lack of task knowledge, lack of control knowledge, or lack of externally available functionality to provide an execution environment for the computations required by the task.

In a transformation system, the transformation rules are determined by the programmer. These rules define a search space of transformed programs. In the agent-based paradigm, the agent processes knowledge of the task and background knowledge to determine the transformation rules that specify useful computation. The problem solving that was performed by the programmer to determine the rules of a transformation system is the problem solving that the agent performs itself. By determining the transformation rules itself the agent establishes its own search space. The typical representation of software system knowledge in automatic programming systems [6, page 292] pre-determines the association between internal actions and external functions. To overcome this limitation, we have proposed a separation of knowledge of the software system functionality from the uses to which it can be put. This allows the agent to itself synthesize the connections between

internal actions and software system functions.

Burstall and Darlington [16, 20] have described a transformation system whose input is a formal specification of a program written in a high-level equational language and that has a collection of formal rules that could be applied to the abstract program to obtain a more efficient version. The user is responsible for providing a complete input program and is assisted in the process of rendering the program efficient. The system exploits the fact that if the input is of a certain form then the rules guarantee certain optimizations. As noted before, our problem definition eschews a complete program as input because we are interested in providing support for the very act of mapping a task specification into a program specification. Additionally, we are also interested in providing automatic support for the mapping of the task onto some means of execution using the functions of a software system.

We have not considered the efficiency of software system use by the agent, a concern in program synthesis work [35], because the agent's goal is defined as obtaining a result for the input task and not the production of a program to compute the result. The agent implicitly must devise an algorithm and a program to compute a result for the input task. But it is only the result for the input task that is of interest to the user and not the agent's internal processing. Therefore, at one level it does not matter whether the agent makes an efficient use of the software system or not. But what of an efficiency requirement internal to the agent ? Sometimes, an inefficient use of the software system by the agent will prevent it from computing a result for the input task. For example, consider the internal action to solve a set of simultaneous equations. In some instances, using Mathematica's *Solve* function may not yield a solution. Some alternatives to try include computing a numerical solution using *NSolve* or replacing constant terms with a single symbol to reduce the amount of symbolic processing required. Reasoning with such alternatives characterizes the efficiency concerns internal to the agent. To enable the agent to reason thus will require it to have a notion of time that it currently lacks.

Chapter 8

Conclusion

In this chapter the contributions of this dissertation are described. Also, some directions for future work are identified.

8.1 Agent design principles

My thesis is that general problem solving and learning principles underly the design of an agent meant to automatically use a software system. To identify some of these principles I designed and implemented an agent for interactively using Mathematica. The implemented agent provides principles for designing agents for the automatic interactive use of software systems whose individual functions provide significant computation.

To use a software system interactively two distinct planning activities are required. A plan for the software system is a sequence of individual software system functions applied to data objects. This plan is executed by the execution of the individual functions in the plan sequence. A plan for the task is a network of internal actions where each internal action is a function applied to data objects. A subset of the internal actions in the task plan are connected to the software system plan sequence as shown in Figure 8.1. That is, some of the internal actions are mapped onto software system functions that make up the software system plan. The result obtained by the execution of a function of the software system plan implicitly constitutes an execution result for the internal action that was mapped onto the software system function. So, there are two planning activities and one plan execution activity

that serves to execute the first plan implicitly.

We can imagine several ways of ordering these planning and execution activities relative to each other. For example, the task plan can be built first followed by the building of the software system plan. Or, the building of the two plans can be interleaved. However, a third alternative captures the essence of interactive programming. Here, not only are the two planning activities interleaved but once an internal action is mapped onto a software system function, the latter is executed. There are several reasons for integrating planning and execution in this manner. First, noting that an internal action is a function application, it may help to execute the internal action through the execution of the software system function to which it is connected. For it is only by executing the internal action that it can be known whether a useful result was obtained or not. Without the execution results for an internal action it can only be assumed that the internal action is useful for solving the input task. Secondly, the result of an internal action can provide guidelines for how to further the task planning. Thirdly, the knowledge of the software system that is used to produce the mapping between an internal action and a software system function may be incomplete or incorrect. Therefore, a planned mapping may not actually be the right mapping. To ascertain whether or not the planned mapping is accurate it is useful to execute the software system function after it has been specified. This way if a change in the mapping is found necessary then either an alternative mapping can be tried or the internal action modified. So, the heuristic is to interleave the building of the task plan and the software system plan, and execute the software system function after it has been specified instead of waiting for the entire software system plan to be available prior to execution. Since the execution of the software system plan implicitly executes the task plan as well, we realize the heuristic of integrating planning and execution activities required for interactively using a software system.

The design principles that we discovered for building agents for automatic interactive use of a software system include a blackboard-style control structure realized within a forward-chaining production system architecture, an object-oriented knowledge representation augmented with a first-order language, a dual-space planning function, an integration of planning and execution, and mechanisms for automatic knowledge acquisition through environment interaction. Each of these principles are described in the following sections.

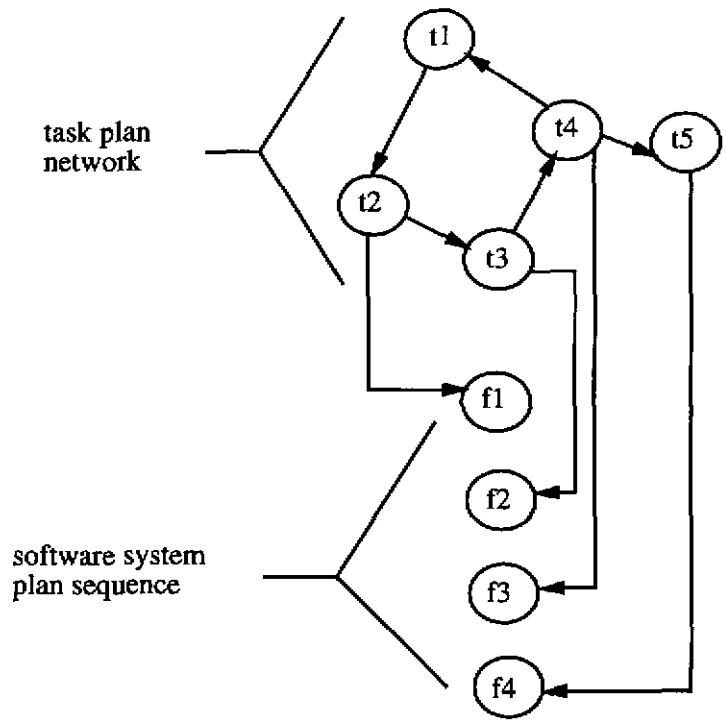


Figure 8.1: The relationship between a task plan and the software system plan sequence.

8.1.1 Control structure

The control structure of the agent is organized as a set of processes acting on a global state called the blackboard. The blackboard contains a task plan, a software system plan, the execution results, and the task representation. Each process is a set of operators that react to the blackboard representations. The task plan is a network of internal actions and the software system plan is a sequence of individual software system functions applied to data objects. Two different focusing decisions are made. First, an internal action is selected as the focus of attention on the blackboard. The blackboard focus of attention serves also as an inference control mechanism as only inferences related to the in-focus internal action are allowed. An inference is nothing but a supervisory operator. For example, a supervisory operator that adds the function application, (equal (o1 o2)), to an internal action description,

that originally had the function application, (compare (o1 o2)), is an example of an inference rule.

Secondly, a selection is made from the applicable process operators. If more than one process operator is applicable a selection is made based either upon the process that is evoked by the operator or the intent of the operator. The process evoked by an operator can be one of supervisory, formulation, interpretation, input, output, or external memory. The intent of an operator is some representation of the state change that the application of the operator will introduce on the blackboard. For example, the intent of a supervisory operator can be to create an internal action to write a polynomial equation to Mathematica. As another example, the intent of a formulation operator can be to use a specified function of Mathematica for the external action specification for an internal action. In the absence of control knowledge to make a selection from among the competing alternatives, a lookahead search is performed. Each of the operators is applied to a copy of the blackboard state. If an operator cannot be applied, that is, produces no state change, then it is assigned the lowest possible priority. If an operator can be applied then it is assigned a high priority. If multiple operators can be applied then they are made indifferent with respect to each other. After all the operators haven been assigned a priority value an arbitrary selection is made from amongst those that were considered indifferent to each other. Observe that only a 1-level deep lookahead search is carried out since the only control decision to resolve is the contention between multiple process operators. The objective of the lookahead is not to find out whether or not a competing operator will lead to a complete solution to the input task. Instead, the objective of the lookahead is to determine the better local candidate at a given point in the problem solving.

The application of operators add representations on the blackboard in a forward-chaining style. This is an appropriate heuristic for automatic interactive use of a software system because of the tentativeness in both the task knowledge and the knowledge of the software system functionality. Given this tentativeness the best thing to do is to apply whatever operators are applicable factoring in any available prioritization knowledge. An operator application in our case produces only additive change to the blackboard. The possible changes to the blackboard state are increments to the task plan and software system plan, and the accretion of results of execution of software system functions. Therefore, a change to the blackboard does not cause

previously applicable operators to become inapplicable.

8.1.2 Knowledge representation

Qualitatively different knowledge is represented as operators of distinct problem spaces. By partitioning the knowledge into multiple problem spaces we realize the benefits of modularity in system development. Two distinct types of knowledge are required: knowledge of software system functionality to plan external actions and task knowledge to plan internal actions. The agent capability is a function of both types of knowledge. All of the representations are part of the global state called the blackboard. The blackboard elements are objects and facts. An object clusters related knowledge and enables efficient access. Facts enable representation of relations among objects. These relations can either be among data objects or among problem solving objects (internal actions, external actions, results). Objects are described by a conjunction of S-expressions. These S-expressions are interpreted either as relations or function applications. The use of both objects and a predicate-calculus-like language for descriptions of objects is similar to Cycl [40]. Using logical formulas helps in limiting introduction of arbitrary attribute names in the representation as pointed out in [72].

8.1.3 Dual-space planning function

We have termed the two planning spaces for using a software system, supervisory and formulation. Problem solving is a consequence of the tentativeness of moves in each of these two spaces. As the agent moves in the supervisory space, its knowledge of what computations the software system can perform allows it to decide to interact with the software system by moving in the formulation space. These spaces are concerned with two essentially distinct functions. The supervisory operators are responsible for the specification of internal actions and the management of the execution of the external actions. The formulation operators are responsible for the specification of external actions. The distinctiveness of function provides a pragmatic reason for the separateness of the supervisory and formulation spaces. However, there are two other reasons that argue for the independent status of these spaces.

A separation of the planning into two separate planning functions enables meta-level reasoning. A problem space is an encapsulation of operators rel-

evant to that space. First let us consider no such encapsulation. That is, all operators belong to an undifferentiated space. Now there is no convenient way to encode knowledge that reasons about those operators that perform the formulation function. With the formulation and supervisory operators differentiated by their membership to the corresponding problem space it is straightforward to reason about whether a move using task knowledge is to be preferred over a move using software system knowledge. For example, for a given internal action it may be possible to refine the action using task knowledge as well as to map the action into an external action to be performed using the software system. A control rule that prefers supervisory operator to a formulation operator applies with a generality that would have to be replaced with rules that check the contingencies of individual operators if a single problem space perspective is adopted. The supervisory process is at the meta-level vis a vis the formulation process because it acts on the results produced by the execution of external actions created by the formulation process operators.

The planning spaces provide independent threads of control. Progress on the task using supervisory operators is possible independent of the processing by the formulation operators. By separating the operators into two problem spaces we allow for such independent progress. For example, while the agent is formulating an external action and is struck, it could begin processing an applicable supervisory operator because the two spaces are logically independent. While the existence of two threads of control is suggestive of parallel processing, the agent is a sequential information processor and progress on multiple threads happens only through interleaving. Another example of independent processing occurs when a formulation process operator is awaiting the execution results from the software system and supervisory operators can be applied during this period.

8.1.4 Integration of planning and execution

The heuristic to integrate planning and execution is a hedge against the tentativeness in both the task knowledge and the software system knowledge. Agent relies on external execution for achieving internal actions but assumes that its knowledge of externally available functionality is incomplete. For example, consider the internal action, (factor (21)). Available knowledge may suggest using the external function, `Factor`. The agent immediately acts to

execute the external action, `Factor[21]`. The result obtained is 21. The agent now realizes that its choice of the external function was inappropriate. If no alternative can be found then the agent may well fail to achieve a result for (factor (21)). Thus, immediate execution of planned actions is a way to avoid building unimplementable plans. As demonstrated in Subsection 4.1.2, the heuristic enables the creation of new strategies. The new strategy creation there required the application of sp-82 as a critical step. However, sp-82 would not apply unless t6 was executed. So, if the task plan had been created fully we would have had t6, t7, and t8 as possible internal actions. But, the relevance of t6 to the achievement of t5 could not be recognized. Therefore, the agent might well have considered t6 an extraneous internal action. By abandoning t6, the opportunity for the creation of a new strategy would have been lost. The interspersing of planning and execution can trigger strategies that are contingent upon execution results. For example, consider the internal action, (derivative $((ax^2+bx+x)(x+cx)$ x)). The result obtained is, $\frac{1+b+2ax}{x+cx} - \frac{(1+c)(x+bx+ax^2)}{(x+cx)^2}$. The form of the result suggests the internal action, (simplify (result o1)), where o1 is denotes the previous internal action. The result now obtained is, $\frac{a}{1+c}$. Thus, the result of the planned action, o1, determines the requirement for the second action. Similar advantages of merging planning and execution have been discussed in [1].

8.1.5 Automatic mapping of subtask onto software system function

The agent must be capable of automatic formulation defined here as the establishment of an association between an internal action and a software system function. For example, consider the calculation, (compare (o1 o2)). There is no explicit association between the calculation and any Mathematica function. In order to formulate the calculation an association is inferred between the calculation and the `Equal` function using the properties of the calculation and the properties of the function. The agent confirms the validity of this inference by trying out the formulated external action, `Equal[o1, o2]`. As another example, consider the case where two functions, `SameQ` and `Equal` are both candidates for the calculation. Here, the agent must infer the more suitable option by reasoning with the knowledge of the properties of the

two functions and the properties of the calculation. These are instances of inferential linkage between an internal action and a software system function. Another strategy is to try out various plausible alternatives. For example, to select among, `Factor` and `FactorInteger`, the agent can try out each and determine which is suited for the internal action. A third strategy is to modify an internal action if the external action specification produces an anomalous result. This case was described in Subsection 4.1.1. A fourth strategy is to try out a mapping and see if it was useful. For example, `Simplify`, may be applied to execution results without a priori knowledge of whether the mapping is useful. In the event that the mapping proves useful, that is, the execution result is indeed simplified, the simplified result can be used instead of the original result in subsequent problem solving. If the mapping proves ineffective, then further processing can be continued ignoring the ineffectual mapping.

8.1.6 Automatic knowledge acquisition

The provision for automatic knowledge acquisition is necessary because the agent is not construed as limited to some fixed set of interactions with the software systems. While processing a task using the software system new knowledge is liable to be generated. In order for the agent to improve its subsequent performance it must acquire the new knowledge. Since the knowledge is generated as part of the performance on the task to encode it manually requires the programmer to pin down the set of interactions - the constraint that we wish not to impose. The various types of knowledge acquired through deliberate acquisition include know-how rules, control knowledge for external functions, strategy knowledge, control rules for strategy rules, and semantics of external functions. The acquisition of control knowledge through reflective processing enables us to not manually represent some of the control knowledge for the agent. This simplifies the design of the agent.

8.2 Automation of task-level symbolic computations

CASs provide automation of symbolic calculations in engineering applications [18, 31, 61]. However, given a task to be solved, an intelligent agent

(human or computer) is required to mediate in the use of the CAS for the task. By task-level automation we mean an automatic mapping of a task specification onto the functions provided by a CAS. Two approaches are possible for achieving task-level automation of symbolic calculations. One approach is to write programs in a CAS for specific tasks [25, 18]. Here, the programmer is the intelligent agent and manually performs the mapping of pre-determined task specifications into CAS programs. The second approach is to build an agent that accepts task specifications from a user and automatically uses a CAS to perform these tasks as illustrated in figure 3.1. This approach can potentially support open-ended symbolic computations with the mathematics of the input task specifications governing what computations are actually performed. Our prototype agent for Mathematica, is an exploration of the second approach. The agent synthesizes the symbolic calculations required by an input task and performs these calculations using a computer algebra system by using appropriate task knowledge, mathematical knowledge, and computer algebra system knowledge. The source of generality lies in the agent's ability to apply relevant knowledge to input tasks to synthesize solutions for them.

A user specifies a task to be performed by the agent. The user is not concerned with whether or not the agent uses a CAS to assist in the performance of the specified task or how the agent uses a CAS if indeed such a use were to be made by the agent. The task is specified in a task specification language. We have not spent much time with fashioning a superior task specification language and have assumed that tasks are specified in the way people verbalize them as function statements and sets of objects representing task data. The agent goes to work on a task specification and begins synthesizing a plan using available task knowledge. Using its knowledge of Mathematica functionality the agent engages in a second planning activity to synthesize a plan for using Mathematica. An agent capable of automatically using a CAS can potentially lessen the learning curve associated with CASs [13]. It can also constitute a basis for investigating automation of the mathematical problem solving process [33].

Let us consider the sensitivity analysis problem class. There is some body of knowledge that constitutes this problem class and enables solution of its instances. Our approach is to build an agent with access to this knowledge. Upon demand by the user the agent applies its knowledge of sensitivity analysis to provide a solution to a given problem instance. Knowing about

Mathematica, as it does, the agent may use Mathematica to perform the symbolic computations needed for the task. Contrast this approach to one that would have us devise a program, perhaps in Mathematica, to solve sensitivity analysis problems. Let us compare the two approaches. The first approach produces an agent capable of applying a body of knowledge to an input task. The second approach provides a program that can act upon some specified input task. Such a program is produced by a programmer who must determine what is needed for this task input. Clearly, the programmer will have to anticipate the requirements of the task input in order to write an appropriate program. The agent also synthesizes a program except that it delays writing the program, as it were, till the time of receipt of the task input. It then interprets the requirements of the task input and proceeds to obtain a solution for the task. So, the agent takes over the role of the programmer in the second approach.

8.3 Limitations

The agent design described in this dissertation is shown to require task knowledge to build task plans in the context of which a use of the software system can be made. For symbolic calculations with Mathematica the task knowledge is well-defined engineering and mathematical knowledge. For other tasks and software systems this knowledge may not readily available. For example, to use the agent for a database system means that we represent knowledge relevant for retrieval tasks. There is no well-defined notion of what constitutes such knowledge. However, the methodology of analyzing protocols of human users of databases can prove helpful in identifying useful knowledge for a given class of tasks to be performed using a given software system.

We have not addressed the issue of efficiency of software system use. For example, suppose that a set of simultaneous equations are to be solved and the `Solve` function of Mathematica is used for the purpose. It is quite possible that the computation time may be unacceptable. To reason with this situation the agent must have a notion of the amount of time that has elapsed since a function execution was initiated on the software system. By itself, this does not seem like a major problem to solve. A simple internal counter could suffice as a clock. Once, the inefficient computation has been recognized, an alternative strategy for the performing the subtask may be

devised. If successful, the whole experience should yield some new knowledge of how to use the software system.

To use the agent for a word-processing system points to another interesting limitation of the current agent design. Currently, the agent accepts parsed output of the software system that it uses. For a word-processor the information in the output of the software system is really visual rather than linguistic. A possible solution to visual processing is to have a set of visual processing operators that determine attributes of interest for the output produced by the word-processor. In fact, a similar technique could enable the agent to perform graphing tasks using Mathematica.

It is useful to separate the design principles identified in this dissertation from the code that is a concrete embodiment of these principles. Clearly, my emphasis has been on discovering some of these design principles. The implementation that I have currently is at best a prototype. A non-trivial amount of work is needed to carry the prototype agent all the way into a product of industrial value. Some of the computations that I have described in this document are not implemented in their fullest generality. A simple illustration can be provided by examining the implementation of an interpretation operator whose purpose is to recognize that the output produced by a function application is the same as the input to the function. Currently, the operator recognizes the input-output sameness for only a limited class of data. Technically, there is no problem in extending the implementation to make this recognition general. After all, what is needed is the ability to compare two S-expressions since we are assuming only linguistic data. But, such a comparison turns out to be convoluted in a production system framework. Observe that I am not suggesting it cannot be done. In fact, I have implemented some generalized list-processing routines as part of my agent implementation. So, the point is only that beyond the identification of principles there lies the need to give expression to these principles in code and this simply takes time no matter how fast you are able to generate code.

In its fullest generality to match the flexibility of a human user of a software system, say Mathematica, sets up the requirement that the agent have access to the same vastness of knowledge that a human can draw upon. From a cognitive perspective this is really what makes the research challenging. But from an engineering perspective it implies that a useful product may be many years away. I believe that a possible way out is to carefully delimit the scope of the agent, say, to the class of geometric reasoning tasks.

8.4 Future Work

In this section I describe some directions for future work.

8.4.1 A proposal for further development of the agent for Mathematica

The current implementation of the Soar agent for Mathematica consists of a limited amount of task-specific knowledge required for the experimentation described in Chapter 5. We can augment the agent's knowledge of sensitivity analysis, trajectory planning, and geometric reasoning to make it more useful for engineering design practice. However, before the agent can be field-tested some non-trivial work is required at the interface between the user and agent. The recent developments in automatic speech recognition technology raise the prospects for providing a voice-based interface between the user and the agent. Some other desirable attributes of the agent in this regard include a capability of the agent to estimate the processing time for a given task. In many cases, the input task might be quite involved and the user will be helped a lot if given an estimate of the turn-around time. Secondly, the agent should be able to walk the user through its processing on the task both to assure the user of the accuracy of the processing and also to instruct the user in how to perform the task.

8.4.2 Transfer to other software systems

Pragmatically, it makes most sense to develop intelligent mediating agents for software systems that perform high value-added tasks. Engineering design software fits this bill most closely. We can try to transfer the agent technology to CAD tools and simulators.

Appendix A

Trajectory planning protocol

Written deliberations

This is a protocol recorded on Jan. 29, 1992:

There are four constraints: two constraints on start and end time; two constraints on start and end positions.

Interaction with Mathematica

```
In[1]:= D[th == d + c t + b t^2 + a t^3, t]

                                  2
Out[1]= 0 == c + 2 b t + 3 a t

In[2]:= D[th == d + c t + b t^2 + a t^3, t, NonConstants -> {th}]
{th}]
            ^ <retype line>

In[2]:= D[th == d + c t + b t^2 + a t^3, t, NonConstants -> {th}]

                                                              2
Out[2]= D[th, t, NonConstants -> {th}] == c + 2 b t + 3 a t

In[3]:= Solve[{th0 == d  + c t0 + b t0^2 + a t0^3, th1 == d + b t1 + c
t1^2 + a t1^3, 0 == c + 2 b t0 + 3 a t0^2, 0 == c + 2 b t1 + 3 a
t1^2}, {a, b, c, d}]
```

```
Out[3]= {{a -> -(((-2 t0 + 2 t1) th0) /

                  4       2          3          2   2       3          3
>           (-t0  + 3 t0  t1 + 4 t0  t1 - 3 t0  t1  - 3 t1  - 2 t0 t1  -

                    2   3       4          4
>             6 t0  t1  + 2 t1  + 6 t0 t1 )) -

>         ((2 t0 - 2 t1) th1) /

              4       2          3          2   2       3          3
>         (-t0  + 3 t0  t1 + 4 t0  t1 - 3 t0  t1  - 3 t1  - 2 t0 t1  -

                2   3       4          4
>           6 t0  t1  + 2 t1  + 6 t0 t1 ),

                    2       2
>       b -> -(((3 t0  - 3 t1 ) th0) /

                  4       2          3          2   2       3          3
>           (-t0  + 3 t0  t1 + 4 t0  t1 - 3 t0  t1  - 3 t1  - 2 t0 t1  -

                    2   3       4          4
>             6 t0  t1  + 2 t1  + 6 t0 t1 )) -

                  2       2
>         ((-3 t0  + 3 t1 ) th1) /

              4       2          3          2   2       3          3
>         (-t0  + 3 t0  t1 + 4 t0  t1 - 3 t0  t1  - 3 t1  - 2 t0 t1  -

                2   3       4          4
>           6 t0  t1  + 2 t1  + 6 t0 t1 ),

                      2             2
>       c -> -(((-6 t0  t1 + 6 t0 t1 ) th0) /

                4       2          3          2   2       3          3
```

```
>          (-t0  + 3 t0  t1 + 4 t0  t1 - 3 t0  t1  - 3 t1  - 2 t0 t1  -

                  2   3       4          4
>            6 t0  t1  + 2 t1  + 6 t0 t1 )) -

               2             2
>       ((6 t0  t1 - 6 t0 t1 ) th1) /

              4       2          3          2   2       3          3
>        (-t0  + 3 t0  t1 + 4 t0  t1 - 3 t0  t1  - 3 t1  - 2 t0 t1  -

                 2   3       4          4
>           6 t0  t1  + 2 t1  + 6 t0 t1 ),

                     2          3          3       2   3       4          4
>     d -> -(((-3 t0  t1 + 3 t1  + 2 t0 t1  + 6 t0  t1  - 2 t1  - 6 t0 t1 )

>             th0) /

               4       2          3          2   2       3          3
>          (-t0  + 3 t0  t1 + 4 t0  t1 - 3 t0  t1  - 3 t1  - 2 t0 t1  -

                  2   3       4          4
>            6 t0  t1  + 2 t1  + 6 t0 t1 )) -

             4       3          2   2
>       ((t0  - 4 t0  t1 + 3 t0  t1 ) th1) /

              4       2          3          2   2       3          3
>        (-t0  + 3 t0  t1 + 4 t0  t1 - 3 t0  t1  - 3 t1  - 2 t0 t1  -

                 2   3       4          4
>           6 t0  t1  + 2 t1  + 6 t0 t1 )}}

In[4]:= Simplify[%]

]
        ^ <retype line>
```

```
In[4]:= Simplify[%]

Out[4]= {{a -> (-2 (th0 - th1)) /

            3                  2          2      3          3
>         (t0  - 3 t0 t1 - 3 t0  t1 - 3 t1  + 2 t1  + 6 t0 t1 ),

>       b -> (3 (t0 + t1) (th0 - th1)) /

            3                  2          2      3          3
>         (t0  - 3 t0 t1 - 3 t0  t1 - 3 t1  + 2 t1  + 6 t0 t1 ),

>       c -> (-6 t0 t1 (th0 - th1)) /

            3                  2          2      3          3
>         (t0  - 3 t0 t1 - 3 t0  t1 - 3 t1  + 2 t1  + 6 t0 t1 ),

                                  2           3              3         3
>       d -> (-3 t0 t1 th0 - 3 t1  th0 + 2 t1  th0 + 6 t0 t1  th0 + t0  th1 -

               2
>           3 t0  t1 th1) /

            3                  2          2      3          3
>         (t0  - 3 t0 t1 - 3 t0  t1 - 3 t1  + 2 t1  + 6 t0 t1 )}}

In[5]:=
```

Appendix B

Storing deliberate knowledge

The following procedure is used to add knowledge generated by problem solving to the long-term memory:

1. Create reject preferences for all applicable operators in the task space. Since all applicable operators have been rejected, Soar recognizes an impasse and enters an SNC subgoal. A problem space called the snc-space is created for the subgoal. An (empty) initial state is selected for the snc-space.
2. Various operators are proposed in the snc-space. Each operator of the snc-space corresponds to a blackboard configuration. An operator selection rule selects that snc-space operator that corresponds to the subset of the blackboard configuration that is desired to be the conditions of the acquired chunk.
3. The application rule for the selected snc-space operator tests the blackboard configuration to which that snc-space operator corresponds and adds to the task space a preference for a knowledge operator. Also tested is the absence of a blackboard fact, (processed X), where X stands for one or more of the elements of the blackboard configuration that triggered the acquisition episode.
4. With a preference for an operator in the task space now available, Soar resolves the SNC impasse and builds a chunk whose conditions are the conditions of the application rule of the applied snc-space operator and whose action is to add a preference for a knowledge operator in the task space. An additional condition of the chunk is the absence of the blackboard fact, (processed X).

5. Soar selects the knowledge operator next in the task space. This operator cannot be directly implemented in the task space and therefore Soar detects an ONC impasse and enters a subgoal to resolve the impasse. A problem space called the onc-space is created for the subgoal. An (empty) initial state is selected for the onc-space.
6. Various operators are proposed in the onc-space that are in one to one correspondence with the snc-space operators. An operator selection rule selects that onc-space operator that corresponds to the subset of the blackboard configuration that is desired to be the actions of the acquired chunk.
7. The application rule for the selected onc-space operator tests the blackboard configuration to which that snc-space operator corresponds and adds the blackboard elements that are the actions of the acquired chunk. Also the blackboard fact, (processed X), is added.
8. The addition of (processed X) causes the preferences for the knowledge operator in the task space to be withdrawn. Consequently, Soar considers the application of the knowledge operator in the task space completed and builds a chunk whose conditions are the conditions of the application rule of the applied onc-space operator and whose action is to add some elements to the blackboard including the fact, (processed X).
9. The two chunks constitute the storage of the acquired knowledge in long-term memory.

Bibliography

[1] P. E. Agre and D. Chapman. What are plans for ? In P. Maes, editor, *New Architectures for Autonomous Agents: Task-level Decomposition and Emergent Functionality.* MIT Press, Cambridge, Massachusetts, 1990.

[2] R. Balzer. Transformational implementation: An example. *IEEE Transactions on Software Engineering*, 7(1), 1981.

[3] R. Balzer. A 15-year perspective on automatic programming. *IEEE Transactions on Software Engineering*, 11(11):1257–1267, 1985.

[4] R. Balzer, N. Goldman, and D. Wile. On the transformational implementation approach to programming. In *Proceedings of the 2nd International Conference on Software Engineering.* IEEE Computer Society, 1976.

[5] D. Barstow. Automatic programming for device-control software. In M. R. Lowry and R. D. McCartney, editors, *Automating software design*, chapter 6. AAAI Press, 1991.

[6] D. R. Barstow. *Knowledge-Based Program Construction.* Elsevier North-Holland, 1979.

[7] D. R. Barstow. Domain-specific automatic programming. *IEEE Transactions on Software Engineering*, pages 1321–1336, 1985.

[8] S. Bhansali. *Domain-Based Program Synthesis Using Planning and Derivational Analogy.* PhD thesis, Department of Computer Science, University of Illinois at Urbana-Champaign, 1991.

[9] T. J. Biggerstaff and A. J. Perlis, editors. *Software Reusability: Concepts and Models*, volume 1. ACM Press, 1989.

[10] W. W. Bledsoe. Splitting and reduction heuristics in automatic theorem proving. *Artificial Intelligence*, 2(1):55–77, 1971.

[11] W. W. Bledsoe. Non-resolution theorem proving. *Artificial Intelligence*, 9:1–35, 1977.

[12] W. W. Bledsoe. Some automatic proofs in analysis. In W. W. Bledsoe and D. W. Loveland, editors, *Contemporary Mathematics: Automated Theorem Proving: After 25 Years*, volume 29, pages 89–118. American Mathematical Society, 1985.

[13] J. Borwein and P. Borwein. Some observations on computer aided analysis. *Notices of the American Mathematical Society*, 39(2), October 1992.

[14] M. Brady, J. M. Hollerbach, T. L. Johnson, T. Lozano-Perez, and M. T. Mason. *Robot Motion: Planning and Control.* MIT Press, 1982.

[15] A. Bundy. A science of reasoning. In J. L. Lassez and G. Plotkin, editors, *Computational Logic: essays in honor of Alan Robinson.* MIT Press, 1991.

[16] R. M. Burstall and J. Darlington. A transformation system for developing recursive programs. *J. ACM*, 24(1):44–67, January 1977.

[17] T. E. Cheatham. Reusability through program transformation. *IEEE Transactions on Software Engineering*, 19(5):589–595, September 1984.

[18] T. Cline, H. Abelson, and W. Harris. Symbolic computing in engineering design. *AI EDAM*, 3(3):195–206, 1989.

[19] J. J. Craig. *Robotics.* Addison-Wesley, 1989.

[20] J. Darlington. An experimental program transformation and synthesis system. *Artificial Intelligence*, 16:1–46, 1981.

[21] E. Davis. *Representations of Commonsense Reasoning.* Morgan Kaufman, 1990.

[22] R. P. dos Santos and W. L. Roque. On the design of an expert help system for computer algebra systems. *SIGSAM Bulletin*, 24(4):22–5, October 1990.

[23] A. K. Ericsson and H. A. Simon. *Protocol Analysis.* MIT Press, 1984.

[24] L. D. Erman, F. Hayes-Roth, V. R. Lesser, and D. R. Reddy. The hearsay-ii speech-understanding system: Integrating knowledge to resolve uncertainty. *ACM Computing Surveys*, 12(2), 1980.

[25] B. L. Evans and J. H. McClellan. Symbolic analysis of signals and systems. In A. V. Oppenheim and S. H. Nawab, editors, *Symbolic and Knowledge-Based Signal Processing.* Prentice Hall, Englewood Cliffs, New Jersey, 1992.

[26] R. E. Fikes, P. E. Hart, and N. J. Nilsson. Learning and executing generalized robot plans. *Artificial Intelligence*, 3(4):252–288, 1972.

[27] C. L. Forgy. Ops5 user's manual. Technical report, Computer Science Department, Carnegie Mellon University, Pittsburgh, Penn., July 1981.

[28] P. M. Frank. *Introduction to System Sensitivity Theory.* Academic Press, 1978.

[29] P. C. Gasson. *Geometry of Spatial Forms.* Ellis Horwood, 1983.

[30] Y. Gil. *Acquiring domain knowledge for planning by experimentation.* PhD thesis, School of Computer Science, Carnegie Mellon University, 1992.

[31] R. Grossman. *Symbolic Computation: Applications to Scientific Computing.* SIAM, 1989.

[32] J. R. Hayes and L. S. Flowers. Identifying the organization of writing processes. In L. W. Gregg and E. R. Steinberg, editors, *Cognitive Processes in Writing.* Lawrence Erlbaum Associates, 1980.

[33] A. C. Hearn. Algebraic computation: The quiet revolution. In V. Lifschitz, editor, *Artificial Intelligence and Mathematical Theory of Computation*, pages 177–186. Academic Press, 1991.

[34] J. R. Hobbs and R. C. Moore. *Formal Theories of the Commonsense World.* Ablex Publishing Corporation, 1985.

[35] E. Kant. On the efficient synthesis of efficient programs. *Artificial Intelligence*, 20:253–305, 1983.

[36] E. Kant and D. Barstow. The refinement paradigm: The interaction of coding and efficiency knowledge in program synthesis. *IEEE Transactions on Software Engineering*, 7(5):458–471, 1981.

[37] E. Kant and A. Newell. Problem solving techniques for the design of algorithms. *Information Processing and Management*, 20(1-2):97–118, 1984.

[38] J. Laird, C. B. Congdon, E. Altmann, and K. Swedlow. Soar user's manual: Version 5.2. Technical Report CMU-CS-90-179, School of Computer Science, Carnegie Mellon University, 1990.

[39] P. Langley. A general theory of discrimination learning. In D. Klahr, P. Langley, and R. Neches, editors, *Production System Models of Learning and Development.* MIT Press, 1987.

[40] D. B. Lenat and R. V. Guha. *Building Large Knowledge-based systems.* Addison-Wesley, 1990.

[41] V. R. Lesser, R. D. Fennell, L. D. Erman, and D. R. Reddy. Organization of the hearsay-ii speech understanding system. *IEEE Transactions on Acoustics, Speech and Signal Processing*, 23:11–23, 1975.

[42] Z. Manna and R. Waldinger. A deductive approach to program synthesis. *ACM Transactions on Programming Languages and Systems*, 2(1):90–121, January 1980.

[43] D. Marr. *Vision.* W. H. Freeman, New York, 1982.

[44] T. E. Mason and C. T. Hazard. *Brief Analytical Geometry.* Ginn and Company, 1947.

[45] W. H. Middendorf. *Engineering Design.* Allyn and Bacon, Inc., 1969.

[46] T. Mitchell. *Version spaces: an approach to concept learning.* PhD thesis, Stanford University, 1978.

[47] T. Mitchell, P. Utgoff, and R. Banerji. Learning by experimentation: Acquiring and modifying problem-solving heuristics. In R. S. Michalski, J. G. Carbonell, and T. M. Mitchell, editors, *Machine Learning: An Artificial Intelligence Approach*, chapter 6. Morgan Kaufmann, 1983.

[48] E. E. Moise and F. L. Downs. *Geometry.* Addison-Wesley, 1967.

[49] I. J. Nagrath and M. Gopal. *Control Systems Engineering.* Wiley Eastern, 1982.

[50] A. Newell. The knowledge level. *Artificial Intelligence*, 18:87–127, 1982.

[51] A. Newell. Putting it all together. In D. Klahr and K. Kotovsky, editors, *Complex Information Processing: The Impact of Herbert A. Simon*, chapter 15. Lawrence Erlbaum Associates, Hillsdale, New Jersey, 1989.

[52] A. Newell. *Unified Theories of Cognition.* Harvard University Press, 1990.

[53] A. Newell and H. A. Simon. *Human Problem Solving.* Prentice Hall, 1972.

[54] A. Newell and D. Steier. Intelligent control of external software systems. Technical Report 05-55-91, Engineering Design Research Center, Carnegie Mellon University, 1991.

[55] A. Newell, G. Yost, J. E. Laird, P. S. Rosenbloom, and E. Altmann. Formulating the problem-space computational model. In R. F. Rashid, editor, *CMU Computer Science: A 25th Anniversary Commemorative.* Addison-Wesley, New York, New York, 1989.

[56] H. P. Nii. The blackboard model of problem solving. *AI Magazine*, 7(3):38–53, 1986.

[57] H. P. Nii. Blackboard systems part two: Blackboard application systems. *AI Magazine*, 7(3):82–106, 1986.

[58] D. A. Norman and D. E. Rumelhart. *Explorations in cognition.* W. H. Freeman, San Francisco, 1975.

[59] E. H. Nyberg. *The FRAMEKIT User's Guide: Version 2.0.* Center for Machine Translation, Carnegie Mellon University, May 1988.

[60] H. Partsch and T. Steinbruggen. Program transformation systems. *ACM Computing Surveys*, 15(3):199–236, September 1983.

[61] R. Pavelle. *Applications of Computer Algebra.* Kluwer Academic Publishers, 1985.

[62] E. J. Purcell. *Analytic Geometry.* Appleton-Century-Crofts Inc., 1958.

[63] C. Rich and R. C. Waters. Automatic programming: Myths and prospects. *IEEE Computer*, pages 40–51, August 1988.

[64] P. S. Rosenbloom, J. E. Laird, A. Newell, and R. McCarl. A preliminary analysis of the soar architecture as a basis for general intelligence. *Artificial Intelligence*, 47(1-3):289–325, 1991.

[65] D. S. Scott. The lattice of flow diagrams. In E. Engeler, editor, *Symposium on semantics of algorithmic languages.* Springer-Verlag, 1971.

[66] P. G. Selfridge. How to print a file: An expert system approach to software knowledge representation. In *Proceedings of the eight national conference on artificial intelligence*, 1988.

[67] W. M. Shen. *Learning from the Environment Based on Percepts and Actions.* PhD thesis, Carnegie Mellon University, 1989.

[68] M. Singer. *Psychology of Language: an introduction to sentence and discourse processes.* L. Erlbaum Associates, Hillsdale, N. J., 1990.

[69] N. K. Sinha and B. Kuszta. *Modeling and Identification of Dynamic Systems.* Van Nostrand Reinhold Co., New York, 1983.

[70] J. W. Sullivan and S. W. Tyler. *Intelligent User Interfaces.* Addison-Wesley, Reading, Mass., 1991.

[71] G. Wiederhold. Mediators in the architecture of future information systems. *IEEE Computer*, 25(3):38–48, March 1992.

[72] R. Wilensky. Some problems and proposals for knowledge representation. Technical report, Computer Science Division, University of California, Berkeley, 1987.

[73] R. Wilensky, D. N. Chin, M. Luria, J. Martin, J. Mayfield, and D. Wu. The berkeley unix consultant project. *Computational Linguistics*, 14(4), December 1988.

[74] R. Wilensky, D. N. Chin, M. Luria, J. Martin, J. Mayfield, and D. Wu. The berkeley unix consultant project. Technical Report UCB/CSD 89/520, Computer Science Division, University of California, Berkeley, 1989.

[75] S. Wolfram. *Mathematica.* Addison-Wesley, second edition, 1991.

[76] W. A. Wulf, M. Shaw, P. N. Hilfinger, and L. Flon. *Fundamental Structures of Computer Science.* Addison-Wesley, 1981.

Algorithms for Automatic Sensor Placement to Acquire Complete and Accurate Information

Sergio William Sedas-Gersey

Submitted in partial fulfillment of the requirements for the degree of
Doctor of Philosophy in Robotics and Computational Design

The Department of Architecture and
The Robotics Institute
Carnegie Mellon University

May 1993

This work was supported by NASA under contracts NAGW-2998, US Air Force under contract F08635-92-C-0019, and DARPA under ctonract DAAE07-90-C-R059. The views and conclusions contained in this document are those of the author and should not be interpreted as representing the official policies, either expressed or implied, of the Robotics Institute or the United States Government.

Agent design for automatic use of a software system: A case study with a Soar agent for Mathematica

Dhiraj K. Pathak

Advisors: Dr. Allen Newell and Dr. David Steier

Robotics
School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy in Robotics at Carnegie Mellon University.

This research has been supported by the Engineering Design Research Center, a National Science Foundation Engineering Research Center.