A CASE-BASED PLANNER TO AUTOMATE REUSE OF ES SOFTWARE FOR ANALYSIS OF REMOTE SENSING DATA

D. Charlebois[§], J.-F. Deguise[†], D. Goodenough[*], S. Matwin[§], M. Robson[†]

[§]Department of Computer Science, University of Ottawa, Ottawa, Canada
[*]Canada Centre for Remote Sensing, Ottawa, Canada
[†]Intera-Kenting, Ottawa, Canada

ABSTRACT

The Canada Centre for Remote Sensing (CCRS) has created an expert system shell as well as an expert system whose task is to provide image analysis programs (Landsat Digital Image Analysis System - **LDIAS**) with the necessary knowledge to solve difficult image processing problems (e.g.: updating Geographic Information Systems). An **ILTI** (Interactive LDIAS Task Interface) provides an expert system with a **Prolog** module designed to answer queries from an **LDIAS** program by retrieving knowledge from an image analysis knowledge base, the Analyst Advisor (**AA**). Image analysis experts currently create **ILTIs**. They have found this to be a time consuming task. Our goal is to design an *incremental/adaptive planner* that will create a plan that emulates the **ILTI**'s behavior by analyzing image processing session dialogs between an image analysis expert and an **LDIAS** program.

Keywords:*Machine learning, Planning, Software reuse, Expert systems.*

1 INTRODUCTION

Expert system (ES) software for the analysis of Remote Sensing (RS) data often relies on existing program libraries for specific analysis tasks. The Analyst Advisor Expert System developed at CCRS [1] is an example of this approach. The knowledge encoded in the Analyst Advisor can be conceptually classified into three categories: 1) the *factual* knowledge: what imaging data are required to solve a specific RS problem (e.g. device on which to calculate image gradients), 2) the *control* or procedural knowledge: what lower-level experts are required to process the knowledge and in what sequence (e.g. perform maximum likelihood analysis for supervised classification after having selected training sites), and 3) the actual *domain* knowledge: what are the computational processes required to provide the specific domain expertise (e.g. the code to compute maximum likelihood probabilities on areas of pixels). The Analyst Advisor uses the RESHELL ES shell to represent the factual and control knowledge, and relies on the FORTRAN code contained in LDIAS (Landsat Digital Image Analysis System, [1]) for the domain knowledge. Consequently, there is a need to interface between Prolog- and frame-oriented structures used to represent the factual and control knowledge with the FORTRAN code that implements the domain experts. In the AA, this interface is provided in the form of ILTI (Interactive LDIAS Task Interface, [4]). Each time a new expert is developed, an ILTI has to be built to integrate the application software with the ES. Development of an ILTI is a non-trivial task requiring expertise in the intricacies of RESHELL. In order to ease the task of the users having to develop new experts, we have set out to build tools to automate the task of constructing an ILTI.

This paper introduces the use of planning to create **ILTIs** from examples of the dialogue between an image analyst and an **LDIAS** task. This approach allows an expert to **train** the planner by presenting it with a number of typical execution runs of an LDIAS task. The planner creates a plan which integrates all the steps that have occurred in the example runs [5] . This plan is subject to a knowledge-based generalization, so that the resulting generalized procedure will provide an interface for an entire class of applications of a given task. This class will reuse sections of plans given to the planner during training, and extend those plans by simple analogical reasoning.

Section 2 describes the current methodology of interface development, and presents the parameters of the problem. Section 3 presents the planning and the learning component of our system. Section 4 concludes by discussing the possible evaluation of the system, and outlining future work.

2 ILTI DESIGN

The first major consideration in designing the planner was to preserve as much as possible of the existing program behavior and thus reduce the amount of recoding of different modules. The existing system has two knowledge bases:

- a global knowledge base (*advisor-frames*),
- a local knowledge base dedicated to each LDIAS program (*task-frames*).

It is also composed of five different program modules:

- RESHELL-expert,
- CA-module,
- ILTI-module
- ILTI-engine,
- LDIAS-task.

The RESHELL-expert is an expert system that requests an image analysis task by firing a predicate supplied by the CA-module. The CA-module (condition-action module) prepares knowledge for a given task by getting information from the AA knowledge base (*advisor-frames*) and storing it in a local knowledge base (*task-frames*). When the CA-module has finished copying knowledge from the advisor-frames to the task frames, it calls the ILTI-module to initiate the communication with the LDIAS-task. The ILTI-module then starts the LDIAS-task by using procedures provided by the ILTI-engine and then sees to answering any questions from the LDIAS-task by querying the task-frames.

For each LDIAS program, there is an interface as described above; hence our requirement to function within the same

framework.

Figure 1 introduces a new design where the components were defined to facilitate the use of a planner. The RESHELL-expert must still start the LDIAS program, but most of the knowledge that could be found in the CA-module predicates is now stored in expert system rules. However, the CA-module now supplies the expert system with predicates that handle tasks which are difficult for expert system rules.

The ILTI-engine provides exactly the same functionality as it did previously as do the LDIAS-task, the advisor-frames and the task-frames.

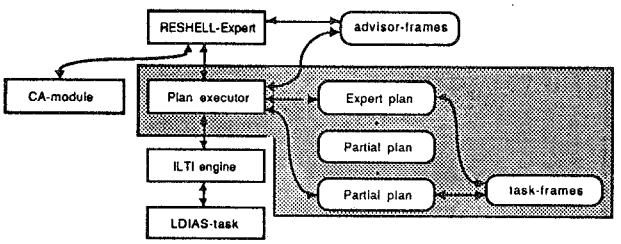

figure 1 - new expert system architecture

The *plan executor*, the *plans* and *partial plans* [2] now replace the ILTI-module. The Plan-executor supervises plan execution to insure proper behavior. A plan is a set of clauses that supply the appropriate information to an LDIAS program given a particular request. Partial-plans are pieces of plans destined for reuse by the same plan as well as other plans.

3 THE PLANNER

Two major design considerations must be addressed before any details of the general design can be laid out.

First, contrary to most planning environments, this planner does not have prior knowledge of possible states for an LDIAS task, nor does it know all the legal transitions from a given state. In this sense, the planner must be adaptive and let the trainer guide it through a new sequence of state transitions or a thread of execution (an important assumption here is that a new thread provided by the user and the LDIAS task is always legal).

Second, the planner must work in an incremental fashion. One execution of the task does not provide the planner with all possible execution threads through the program. In fact, it supplies an unambiguous, single thread. For most LDIAS software, several threads can exist and the planner must be able to add new threads to the existing one(s) incrementally.

Given an existing ILTI, the planner should supervise a user dialog with the LDIAS task and react when a prompt or an answer to a prompt does not match what has previously been recorded for the ILTI. The result of this reaction would be to add, incrementally, the appropriate predicates to handle the new thread of execution.

3.1 SYSTEM COMPONENTS

The planner has three essential components (see figure 2):

- the planner module,
- the planner knowledge base:
  - plans,
  - partial plans,
  - micro-plans,
- the RESHELL knowledge base:
  - advisor frames,
  - planner taxonomy.

The *planner module* is an *adaptive/incremental* planner that reacts to new interactions between a *domain expert* (for AA, an image analysis expert/researcher) and an LDIAS task. A new interaction is a dialog between the domain expert and an LDIAS task that the planner has not seen previously and, thus has not provided for in a plan. In these circumstances, the knowledge contained in the plan developed so far by the planner must be extended (new alternatives must be added to the plan).

The LDIAS task execution through the use of a plan can best be expressed by:

```
set up LDIAS task parameters
start LDIAS task
repeat
        get prompt from LDIAS task
        process prompt and return answer
until normal disconnect request or LDIAS task abortion
```

Each iteration through this loop should place the plan in a new state. The state transitions are fired when an answer is provided by the *advisor-frames* or by the domain expert.

Given this algorithm, two situations involve unrecognizable prompts:

- when working from scratch (i.e.: a plan did not exist for the LDIAS task at hand),
- when no answer can be found in the *advisor-frames* or when the domain expert replies with an unknown answer (one for which a state transition does not exist).

The planner must recognize the occurrences of these events and handle them in the appropriate manner.

Detecting when the system is working from scratch is trivial and can be done in different ways, the simplest being to check for the presence of a plan dedicated to the LDIAS task at hand.

However, detecting if the domain expert replies with an unknown answer will require more extensive processing. The planner must:

- process the prompt and identify the object(s) the LDIAS task needs,
- process the objects, with help from the domain expert, to determine if the AA has knowledge of the objects at this stage of processing, or if they should be added to it,
- create a new state identifier and state transition predicate.

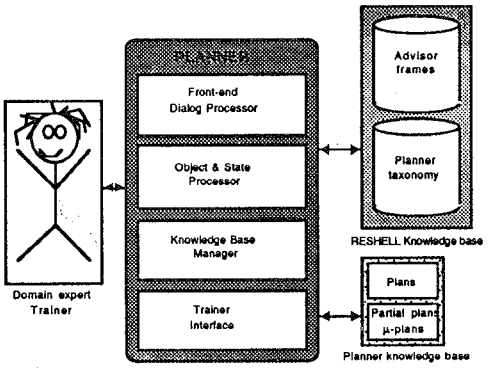

figure 2 - planner design

The task of processing the prompts and identifying the objects being manipulated is assigned to the *Front-end Dialog Processor* (**FDP**) (see figure 3). By using a language defined with a Definite Clause Grammar (**DCG**), the FDP can analyze each prompt and identify the actions and the objects used by the LDIAS task and place them in a symbol table. This will allow the planner to recognize the different contexts where objects are used.

Once the objects have been identified, the *Object & State Processor* (**OSP**) has essential tasks to accomplish. First the OSP must process the objects. If the objects in the LDIAS task prompt have not been recorded in the *advisor-frames* or the planner taxonomy, the planner must determine:

- if an object is created and used only during the execution of the current LDIAS task,
- if an object should normally be output from another LDIAS task and used by the current task,
- if an object is created by the current LDIAS task and if it should be output and made available to other LDIAS tasks.

The second task the OSP must perform is to trace the state transitions the plan executor fires. As known answers are

given to prompts, the ILTI changes states. When an answer for which there is no legal transition exists, the OSP must create a new legal transition from the current state given the new input. This will trigger a dialog between the trainer and the planner. The planner will have to determine if the new transition leads to an existing state or if it leads to a new state.

If a new transition leads to an existing state, the planner simply has to add a new legal transition to the FSM (finite state machine) since the initial state and final state for the new transition are known. Also, if the initial and final states of different transitions are the same, *Machine Learning* (ML) processing may be necessary to generalize and possibly merge the transitions.

If a new transition leads to a new state, the state transition must be added to the FSM and the process repeated.

Whenever the planner cannot determine the origin of an object, its use by other LDIAS tasks or its place in the planner taxonomy or the advisor-frames, the trainer interface must present the problem to the domain expert in order to classify the object. The dialog between the domain expert and the planner will depend on the amount of knowledge the planner has gathered about the object.

Partial-plans and micro-plans are the main building blocks used by the planner. Partial-plans are sequences of state transitions that are designed to answer the needs of frequent occurrences of similar behavior. More specifically, LDIAS programs usually process objects that have the same general structure. Since these objects have the same attributes, gathering their particular values can be accomplished using the same subsequences of plans leading to the reuse of partial-plans. The most prominent example is that most, if not all, LDIAS programs require image files as input or output. In these cases, a partial-plan that can build a file specification can be reused. Micro-plans are very simple, generalized state transitions that can be used to answer prompts that occur often.

3.2 PLANNER OUTPUT

Since the goal of the planner is to create and/or extend plans, its design must include a detailed description of its output. As an example, we have used the SEGGRA LDIAS program. This program is used to create a gradient image file for the purpose of segmentation. By using a log produced by SEGGRA we can step through the plan creation process.

Two problems must be addressed:

- how to select state names,
- how to determine the current state and fire a transition.

Although as programmers, we have complete latitude over state name selection, the plans must have state names that are easy to generate automatically and that are understandable to a domain expert. Also, since the planner must produce a plan that functions within the existing RESHELL/AA framework, the state-checking and transition-firing mechanisms must be added without altering the behavior of the ILTI-engine or RESHELL-experts.

When an LDIAS task executes, it follows a thread of execution according to the answers that a user has provided. From a different perspective it can be seen that:

- each prompt indicates what state the LDIAS task is in,
- each answer is the input that fires a state transition.

These two statements motivate:

- using actual prompts as state names,
- using actual user input to create a transition table.

3.2.1 AN LDIAS EXAMPLE (SEGGRA)

Although the new ILTI for SEGGRA ,expressed as a FSM, does not handle as much as its original version (the handcoded ILTI), the new ILTI does provide further understanding of the work involved in its automatic generation.

By carefully studying a SEGGRA execution log, state names and state transitions can easily be identified and subsequently encoded in the ILTI. The first prompt the domain expert has to answer is:

```
Select device on which to perform gradient operation
Enter one of VDP UNIDSK {UNIDSK} >
```

Hence the choice for the initial state name is:

```
device_to_perform_gradient_operation
```

The answer given by the domain expert is:

```
unidsk
```

Hence the creation of a state transition from the initial state (device_to_perform_gradient_operation) to the next state (whose name remains to be determined) upon the input *unidsk*.

Before memorizing the transition, the next state must be determined. This can be accomplished by using the next prompt:

```
Enter input UNIDSK filename specification
```

We can now create a state transition expressed as the Prolog clause:

```
transition(
  device_to_perform_gradient_operation,
  unidsk,
  enter_input_unidsk_filename_specification
).
```

The clause that will fire a state transition through knowledge base access is:

```
process_state :-
  state( device_to_perform_gradient_operation ),
  frame_get( seggra, input_device, ANSWER ),
  transition( device_to_perform_gradient_operation, ANSWER, STATE ),
  set_state( STATE ).
```

In section 3.1.1 it was stated that there were three different types of objects:

- objects created and used only during the execution of the current LDIAS task,
- objects created by the execution of previous LDIAS tasks and used by the current task,
- objects created during the execution of the current LDIAS task and made available to other tasks.

The predicates designed above serve as a template for the generation of predicates that handle objects created by the execution of previous LDIAS tasks and used by the current task.

Objects created and used only during the execution of the current LDIAS task can be handled in two different ways:

- ask the LDIAS user to supply the data,
- hardcode a default reply for the LDIAS task question.

In the SEGGRA log, one such object would be the reply to the question:

```
Display Histogram of gradient? (Yes/No)
Enter one of YES NO {YES} >
```

Given our knowledge of the task at hand, it is preferable to have the ILTI ask the user, hence the predicate:

```
process_state :-
    state( display_histogram_of_gradient ),
    write( 'Display Histogram of gradient?' ),
    write( 'Enter one of YES NO {YES}' ),
    read( ANSWER ),
    transition( display_histogram_of_gradient, ANSWER, STATE ),
    set_state( STATE ).
```

Had our choice been to hardcode the default value *no*, the result would be:

```
process_state :-
    state( display_histogram_of_gradient ),
    transition( display_histogram_of_gradient, no, NEWSTATE ),
    set_state( NEWSTATE ).
```

The final case is for objects created during the execution of the current LDIAS task and made available to other tasks. In the SEGGRA log, the output gradient file is such an object since it will be subsequently used by the segmentation program (this is known because of dialog with the domain expert).

Given that RESHELL provides daemons to handle querying of the *advisor-frames*, the predicate that can get the output file name is:

```
process_state :-
    state( enter_output_filename_specification ),
    frame_get( seggra, output_filename, ANSWER ),
    transition(
            enter_output_filename_specification,
            ANSWER,
            STATE
    ),
    set_state( STATE ).
```

We need only create a daemon that will ask the LDIAS task user for a filename and add an *if_needed* facet for the *output_filename* slot of the *seggra* frame:

```
frame( seggra, output_filename, if_needed, get_output_filename ).
```

We can interpret this clause as a specification of the *if_needed* facet described above, and its meaning is:

if a *frame_get* is attempted for the *seggra* frame slot *output_filename*
and
no value exists
then
fire the daemon *get_output_filename*

The *get_output_filename* daemon for the *seggra* frame slot *output_filename* would be:

```
daemon( seggra, output_filename, get_output_filename, ANSWER ) :-
  write( 'Enter output UNIDSK filename specification' ),
  read( ANSWER ),
  frame_put( seggra, output_filename, ANSWER ).
```

By using this approach, only three *process_state* clause templates are necessary. The first is for objects provided by the LDIAS task user and used only locally. The second provides default values. The third is for objects that must be stored in the *advisor-frames*.

4 Conclusion

The paper shows a design of a case-based planner that applies machine learning techniques to the problem of automated re-use of specialized interfaces between the factual knowledge and the domain knowledge in a RS expert system. The planner is currently being developed at CCRS. Early experience indicates that even after a successful implementation, several important research issues remain to be addressed. The problem of evaluation of the planner is the first such issue. How to measure success or failure of the proposed approach, compared to the existing manual procedure? A simple way to evaluate the utility of the planner would be to assess the effort that will go into creation of several interfaces using the planner, as opposed to the normal, manual procedure of interface development. Another question is how best to train the planner. This is the problem of selection of training runs that will be processed by the proposed system. Should they all be similar? Or should they rather cover a broad spectrum of possible uses of the same task expert, at the expense of intensive user dialogue with the planner? One can expect that different trainers may develop individual styles of training the planner, just as there are individual styles of teaching.

References

[1] Goodenough, D.G., Goldberg, M., Plunket, G., Zelek, J., 1987, *An Expert System for Remote Sensing*, IEEE Transactions on Geoscience and Remote Sensing, Vol. GE-25, no. 3, pp 349-359.

[2] Hendler, J., Tate, A., Drummond, M., 1990, *AI planning: systems and techniques*, AI magazine, vol. 11 no. 2, summer 1990, p82.

[3] Riesbeck, C.K., Schank, R.C., 1989, *Inside Case-based Reasoning*, Lawrence Erlbaum Associates, Publishers.

[4] Robson, M., Goodenough, D.G., Deguise, JC., 1990, *Automated Program Execution in a Hierarchical Expert System: RESHELL*, Proceedings of the 23rd DECUS Canada Symposium.

[5] Waterman, D., Faught, W, Klahr, P., Rosenschein, S., Wesson, R., 1986, *Exemplary programming: applications and design considerations*, In *Expert systems: techniques, tools and applications*, edited by Klahr, P. and Waterman D., Addison-Wesley, pp 273-309.