Connecting ARC/INFO*and SNACTor - Project Report †

Stuart C. Shapiro‡, Hans Chalupsky‡
and Hsueh-cheng Chou§
National Center for Geographic Information and Analysis

June 8, 1992

Abstract

This report describes an interface between ARC/INFO, a geographic information management system, and SNACTor, the SNePS acting component. It also shows a small example interaction that demonstrates how ARC/INFO and SNACTor can interact using the new interface, and how more sophisticated future applications can make use of its functionality. The interface was designed and implemented during a two-month research project carried out in Summer, 1990 at the State University of New York at Buffalo.

1 The Problem

The problem tackled in this project was to interface ARC/INFO[2, 3], a commercially available geographic information management system that is widely used among geographers all

*ARC/INFO is a registered trademark of Environmental Systems Research Institute (ESRI), Inc., Redlands, CA

†This work was supported in part by the National Science Foundation Grant No. SES-88-10917 to the NCGIA, and in part by a grant to the authors from the Environmental Systems Research Institute.

‡Principal affiliation: Department of Computer Science
§Principal affiliation: Department of Geography

over the world, and SNACTor[4, 7, 8], the acting component of SNePS[5, 6]. SNePS is the Semantic Network Processing System, a knowledge representation and reasoning system developed by Stuart C. Shapiro et al. at the State University of New York at Buffalo. The main motivation for interfacing these two systems is to study how a knowledge representation system such as SNePS with its powerful representation and inference mechanisms can be used to develop more sophisticated and (human) user friendly interfaces to database systems such as ARC/INFO.

The main technical problem that had to be solved in this project was to interface two very different systems that are similar in one respect: They are not designed to be called from other programs. ARC/INFO is a collection of programs written in FORTRAN with two main components, (1) ARC, the module that handles all geography specific computations, and that gives a frame from which a variety of other programs can be called to perform various tasks such as editing, plotting, etc., and (2) INFO, a relational database management system that is used to store and access geographical databases in a variety of ways. ARC/INFO is a standalone system that was not designed to communicate with other programs, but rather give the user all functionality that s/he needs from within the ARC/INFO environment on a variety of different hardware and operating systems platforms.

SNePS and SNACTor on the other hand, are systems written in Common Lisp. They are similar to ARC/INFO in the respect, that it is not possible to "call" them from within another program, because the startup of the Common Lisp environment takes to long and because of the state information that accumulates and that is necessary for proper operation. What had to be done was to define a communication medium and a protocol with which these two different systems could communicate.

2 Scope of the Project

The scope of the project was to define and implement an interface between the two systems, and to develop a small demonstration that shows how the two systems can interact using the interface. This demonstration shows some features of how an interaction with a geographic information management system could be done in a small and simple example. It was not the intention to develop a full fledged natural language interface to ARC/INFO.

3 Description of the Interface

3.1 General Interaction Between ARC/INFO and SNACTor

The basic idea underlying the design of the interface is to run ARC/INFO and SNACTor in two separate processes and have them communicate on some medium via some simple protocol, instead of calling ARC/INFO or subroutines of it from SNACTor directly. The motivation for not doing the latter was that during a typical ARC/INFO session a lot of internal state information is acquired and automatically taken care of. If SNACTor would call ARC/INFO subroutines directly, it would have to manage all this state information by itself, which would amount to rewriting parts of the ARC/INFO functionality within SNACTor. Another reason for choosing the protocol design is portability. Calling FORTRAN subroutines from Common Lisp is different for every implementation of Common Lisp. The protocol design, however, allows SNACTor to be run in various different Common Lisps yet still allowing communication with ARC/INFO without having to change the interface.

The communication medium chosen is the file system. Commands from SNACTor to ARC/INFO are issued by generating little command files, results from command executions in ARC/INFO are brought back into SNACTor via watch files (a special mechanism provided by ARC/INFO to record ARC/INFO interactions, very similar to the script facility in UNIX[1]).[2] The reasons for choosing the file system as a medium were portability and simplicity. Portability, because the interface to the file system is nicely standardized in Common Lisp, and simplicity, because it was easier to write a simple communication program this way then, say, writing a full fledged server/client package that communicates via Internet ports. Even though our solution does not give a fully transparent communication

[1]UNIX is a trademark of AT&T Bell Laboratories.

[2]The interface described in this report was developed under the UNIX operating system. There will be various UNIX specific terms used throughout this document. A basic familiarity with the UNIX operating system is assumed.

across the network, it is possible to run ARC/INFO and SNACTor on different machines as long as the file system of one machine on which either ARC/INFO or SNACTor are running is visible from the other machine, e.g., by remote mounting via NFS (the network file system).

A simple handshake protocol is used, where SNACTor acts as a master and ARC/INFO as a slave. ARC/INFO waits for a command to be issued, SNACTor issues a command and waits for a result to be generated, ARC/INFO executes the command, generates a result file and waits for the next command to be issued, SNACTor reads the result file, acts accordingly and generates the next command and so on. Synchronization is achieved via file existence tests.

3.1.1 The Interface Directory

As mentioned above the communication medium chosen is the file system. All communication is done by writing files to and reading files from an interface directory. This directory has to be accessible from ARC/INFO as well as SNACTor which is trivially achieved if both processes run on the same machine. If the two programs run on separate machines one machine must be able to read from and write to the file system of the other machine.

The interface directory has to be defined for ARC/INFO as well as SNACTor to allow proper communication. This can be done by editing pathname variables in files that will be described below.

3.1.2 The ARC/INFO side

The ARC/INFO side of the communication makes use of AML, the Arc Macro Language[1]. AML provides control constructs and a variety of functions that allow one to write programs containing arbitrary ARC/INFO commands. These programs can be run inside ARC/INFO just as ordinary commands (UNIX shell scripts provide a similar mechanism). What ARC/INFO does is to periodically look into the interface directory for a file `nextcommand.aml`. If it exists it executes it, records the output of the execution into a file `comout.log` and then deletes the file `nextcommand.aml`. Then it waits for a new file `nextcommand.aml` to be generated.

What we have here is a simple command loop. Read a command, execute it, print the results and wait for the next command. This algorithm itself is defined in an AML script called `command-loop.aml`. Before interaction with SNACTor can start ARC/INFO has to be started and `command-loop.aml` has to be executed. A special UNIX shell script `arcloop` has been written that does all this for the user, so all s/he has to do to start the ARC/INFO side of the communication is to call `arcloop` at the UNIX prompt.

Here is what `command-loop.aml` does:

```
clear the interface directory
repeat
  if (EXISTS nextcommand.aml) then
     run nextcommand.aml
     delete nextcommand.aml
  else
     sleep for a second
until (EXISTS exit)
return from command-loop.aml
```

In the first step it clears the interface directory, i.e., it deletes `nextcommand.aml` and `exit` files that might have existed in the directory from previous runs. Then it starts the main part of the command loop. Note that there is nothing in the algorithm that deals with recording of results. This is handled by the file `nextcommand.aml` itself and the command loop does not have to know about this part. The "sleeping" in the else part of the if statement is to break the loop and free the cpu for other things (otherwise the endless looping itself may use up a lot of cpu cycles). The termination criterion is the existence of an `exit` file. Once this file has been created by SNACTor the command loop will terminate and return to the ARC/INFO top-level.

3.1.3 The `nextcommand.aml` file

Here is what an actual `nextcommand.aml` file generated by SNACTor looks like:

```
&args interface
&watch %interface%/comout.log
ARCPLOT
&watch &off
&return
```

The symbols starting with an '&' are special AML directives. The nextcommand.aml file takes one parameter called interface. It supplies the location of the interface directory. The &watch directive turns on the recording of command output. The output will be sent to the file comout.log in the interface directory. Then the actual ARC/INFO command (ARCPLOT in our example) gets executed. After that output recording is turned off which will close the file comout.log and make sure that only output of the command issued in this nextcommand.aml file appears in comout.log. Then the command terminates and returns to command-loop.aml (see above) where the file nextcommand.aml will be deleted in the immediately following step.

3.1.4 The SNACTor side

The SNACTor side of the communication is defined in the Common Lisp file driver.lisp. The central function defined in this file is the function execute-command. Here is its definition and the definition of an auxiliary function that it uses (the variables *nextcom* and *comout-name* hold actual pathnames as their values):

```
(defun wait-for-completion ()
  "Waits until the file nextcommand.aml gets deleted by the
AML command loop running in the ARC process. Returns once
nextcommand.aml does not exist anymore."
  (loop (unless (probe-file *nextcom*)
          (return))
        ;; give system some time to breathe
        (sleep 0.1)))

(defun execute-command (command-string)
  "Takes a COMMAND-STRING and writes a file nextcommand.aml into the
communication interface directory. The command loop running in the ARC
process waits for this file, executes it and deletes it. The output
generated by the command gets written to a command output (watch) file.
This allows to get results back into the LISP process."
  (wait-for-completion)
  (with-open-file (nextcom *nextcom* :direction :output)
    ;; nextcommand.aml takes as an argument the name of the communication
    ;; directory viewed from the ARC process, into which ARC will
    ;; write the comout file
    (format nextcom
            "&args interface~%~
             &watch %interface%/~a~%~
             ~a~%~
             &watch &off~%~
```

```
               &return~%"
              *comout-name* command-string))
    (wait-for-completion))
```

Suppose we want to send the command `ARCPLOT` to the ARC/INFO process. What we have to do is to evaluate `(execute-command "ARCPLOT")` in our Lisp process. Then `execute-command` does the following: (1) It waits for the completion of a command that might currently be executed by the ARC/INFO process. Completion is indicated by the non-existence of the file `nextcommand.aml` (remember, in the command-loop algorithm described above `nextcommand.aml` gets deleted immediately after its execution). (2) After the preceding command got completed `execute-command` generates a file `nextcommand.aml` that contains `ARCPLOT` as its main command (see above for description of `nextcommand.aml`). (3) It waits for the completion of the current command in the ARC/INFO process and returns after that.

Note that `execute-command` does not deal with retrieving or printing of results. After command execution `execute-command` returns and control is back at the caller. The caller might then choose to read the file `comout.log` to process results. This can be done using the function `print-result`. `print-result` takes a stream as an argument. If the supplied value is `NIL` the result will be returned as a string. If the result is not handled it will be overwritten during the next call to `execute-command`.

Calling the function `stop-arc` will generate an `exit` file and terminate the ARC/INFO command loop (see above).

3.2 An Example Interaction of ARC/INFO and SNACTor

The following is an annotated demonstration showing a simple example interaction with the system. The file `arcdemo.lisp` defines this example, and you can run it by using the `demo` command of SNePS-2.1. The output shown below has been edited to save space[3], but it stems from an actual run using the new interface. User input is shown at "*" or ":" prompts. The system's response follows after the user input separated by a blank line.

[3]Timing information and excess blank lines have been removed, some lists have been "prettified" to overcome problems of Lisp's pretty-printer

This example shows only the SNACTor part of the interaction. ARC/INFO actions resulting from commands sent to ARC/INFO from SNACTor will not be shown, because they involve drawing of maps or input from the mouse. These ARC/INFO actions will be described at the appropriate places.

At this point ARC/INFO is running (assuming a startup procedure as described in the appendix). It was started with the command `arcloop` and is waiting for commands from SNACTor. An example database with different coverages has been installed (this example database and how it can be installed is described in [3]). A basic familiarity with the example database, its coverages and concepts as well as with basic concepts of ARC/INFO is assumed.

Now we start SNePS-2.1 and run the demonstration. The first three SNePSUL commands (SNePSUL is the SNePS User Language) deal with initializations. They reset the net, define a set of relations that will be needed later on, and a path that will be used in subclass/superclass reasoning.

```
> (sneps)

   Welcome to SNePS-2.1

 Copyright 1984, 88, 89 by Research Foundation of State University of New York

   10/28/1990 21:39:24

* (demo "~/snactor/arcdemo.lisp")

File ~/snactor/arcdemo.lisp is now the source of input.

* (resetnet t)

Net reset

* ; Arc labels for predicates and relations

(= (define action lex object1 object2  object3 sub sup agent component
           presumably act plan  goal effect then else condition until
           do member class object property rel arg1 arg2 precondition
           relation attribute translations value translated-value
           ) NEW-RELATIONS)

(ACTION LEX OBJECT1 OBJECT2 OBJECT3 SUB SUP AGENT COMPONENT PRESUMABLY
        ACT PLAN GOAL EFFECT THEN ELSE CONDITION UNTIL DO MEMBER CLASS
        OBJECT PROPERTY REL ARG1 ARG2 PRECONDITION RELATION ATTRIBUTE
```

```
        TRANSLATIONS VALUE TRANSLATED-VALUE)

* (define-path class (compose class (kstar (compose sub- sup))))

CLASS implied by the path (COMPOSE CLASS (KSTAR (COMPOSE SUB- SUP)))
CLASS- implied by the path (COMPOSE (KSTAR (COMPOSE SUP- SUB)) CLASS-)
```

Now we are ready to read in the grammar and the lexicon. The grammar used is almost identical to the grammar developed by Sy Ali (see [8]), apart from one little extension that allows for arbitrary quoted strings. The need for this will be explained later. The lexicon defines words used in descriptions of plans and actions[4].

```
* (^ (lexin "~/snactor/arclex.dat"))

undefined- (NIL NIL NIL NIL NIL NIL NIL NIL NIL NIL NIL NIL NIL NIL)
(arc arcplot .... table hand)

* (^ (atnin "~/snactor/arcgram.dat"))

State S processed.
State RESPOND1 processed.
  ......
State GTHRESH processed.
State END processed.

 Atnin read in states: (END GTHRESH ..... RESPOND1 S)
```

Now we set some variables that control input reading behavior as well as various forms of tracing. In this example all tracing has been turned off to save space. For debugging purposes it is possible to “fake” actions, i.e., to not actually send commands to ARC/INFO and just print them instead. We do not want to fake actions in this demonstration.

```
* (^ (setq parser:|*TERMINATING-PUNCTUATION-FLAG*| '("." "!" "?")
           snip:|*INFERTRACE*| nil
           parser::|*DEBUG*| nil))

* (^ (setf |*PLANTRACE*| (if (y-or-n-p "Plantrace?")
                             (if (y-or-n-p "Surface?")
                                 'surface
                                 t))))

Plantrace?(Y or N): n

* (^ (setf |*JUST-FAKE-IT*| (y-or-n-p "Fake actions?")))
```

[4]The demonstration uses the old morphological analyzer englex.lisp. The new implementation could not be used because of some differences in behavior.

```
Fake actions?(Y or N): n
```

Now we tell SNACTor about a set of primitive actions, i.e., actions that will not be further decomposed before they are executed. The effect of a primitive action is defined as a piece of Lisp code to be executed whenever the primitive action gets executed. The first set is a set of actions whose definitions already come with the SNACTor package. The second set is defined via the function `declare-primactions`, a function that declares all the primitive actions which were defined specifically for this demonstration.

```
* (describe
     (assert member ((build lex "snsequence")
                     (build lex "snif")
                     (build lex "sniterate")
                     (build lex "achieve")
                     (build lex "believe")
                     (build lex "forget"))
             class (build lex "primitive")))

(M8! (CLASS (M7 (LEX primitive)))
     (MEMBER (M1 (LEX snsequence))
             (M2 (LEX snif))
             (M3 (LEX sniterate))
             (M4 (LEX achieve))
             (M5 (LEX believe))
             (M6 (LEX forget))))

(M8!)

* (^ (declare-primactions))

(tell do-one say type do-all issue clear execute start interpret noop
      send stop)
```

The SNePS node described below defines a mapping between landuse codes used in the tables defining the `Landuse` coverage and their corresponding translations which are more intelligible for humans. This mapping will be used later when we try to identify a certain region on a map.

```
* (describe
   (assert relation landuse
           attribute lu-code
           translations ((build value 100 translated-value Urban)
                         (build value 200 translated-value Agriculture)
                         (build value 300 translated-value Brushland)
                         (build value 400 translated-value Forest)
```

```
                           (build value 500 translated-value Water)
                           (build value 700 translated-value Barren))))

(M29! (ATTRIBUTE LU-CODE)
      (RELATION LANDUSE)
      (TRANSLATIONS (M23 (TRANSLATED-VALUE URBAN)
                         (VALUE 100))
                    (M24 (TRANSLATED-VALUE AGRICULTURE)
                         (VALUE 200))
                    (M25 (TRANSLATED-VALUE BRUSHLAND)
                         (VALUE 300))
                    (M26 (TRANSLATED-VALUE FOREST)
                         (VALUE 400))
                    (M27 (TRANSLATED-VALUE WATER)
                         (VALUE 500))
                    (M28 (TRANSLATED-VALUE BARREN)
                         (VALUE 700))))
```

After resetting some internal parser variables we are ready to start up the parser. The prompt will be ":" from now on.

```
* (^ (reset-all))

* (^ (parse -1))

 ATN parser initialization...

 Input sentences in normal English orthographic convention.
May go beyond a line by having a space followed by a <CR>
To exit parser, write ^end.
```

Now we tell SNACTor about the domain. We will define some basic concepts and terminology. The next sentence, for example, tells SNACTor that `Landuse` belongs to the class `polygon coverages`. This will allow us later on to make assertions about `polygon coverages` that will automatically apply to `Landuse` as an instance of this class.

The responses of SNACTor are of the form "I understand that...". These responses are not just echoed input, they rather are generated from the SNePS nodes constructed during the parsing of the sentence with the help of a generation grammar. The generated sentence will match the input sentence if the input was properly understood by SNACTor.

```
 : Landuse is a polygon coverage.

 I understand that Landuse is a polygon coverage.
```

Define the class to which the `Streams` coverage belongs.

```
: Streams is a line coverage.

I understand that Streams is a line coverage.
```

The next sentences define subclass/superclass relationships between different types of coverages.

```
: Polygon coverages are coverages.

I understand that polygon coverages are coverages.

: Line coverages are coverages.

I understand that line coverages are coverages.
```

Now we tell SNACTor about peculiarities of the ARC/INFO environment. For example, that `arc` is a program and that `arcplot` is an embedded program. This is important later on, because, for example, before `arcplot` can be invoked `arc` (the frame program) has to be active.

```
: Arc is a program.

I understand that arc is a program.

: Arcplot is an embedded program.

I understand that arcplot is an embedded program.

: Embedded programs are programs.

I understand that embedded programs are programs.
```

We have to tell SNACTor about the different states ARC/INFO can be in. For example, if `arcplot` is running (it is an embedded program) certain commands that are only valid in the frame program `arc` cannot be executed. I.e., `arcplot` is active and `arc` is not. Now, if a certain program needs `arcplot` to be active it should activate it if necessary. Activating means starting it if `arc` is active. A more complicated case with more than one embedded program could require quitting the other embedded program and then starting `arcplot`. We will not need this for our example. The connections between starting, stopping and activity of programs is defined as a set of effects of the actions `start` and `stop`. I.e., a sentence of the form `After ...ing a ...  P` defines P as an effect of the mentioned action.

```
: After starting a program the program is active.

I understand that after performing start on a program, the program
is active.

: After stopping a program the program is not active.

I understand that after performing stop on a program, the program
is not active.

: After starting an embedded program arc is not active.

I understand that after performing start on a embedded program,
arc is not active.

: After stopping an embedded program arc is active.

I understand that after performing stop on a embedded program,
arc is active.
```

Here is an interesting sentence: It describes a conditional plan. SNACTor can use it in the following way: Suppose it wants to activate `arcplot`. It will try to find a plan that tells it how to do this. It will find the plan described below because `arcplot` is of the class `embedded program`. Then it checks whether the plan is applicable by examining whether its condition holds. If so it executes the plan.

```
: If arc is active then
  a plan to achieve that an embedded program is active is to start
  the embedded program.

I understand that if arc is active then a plan to achieve that
a embedded program is active is by performing start on the embedded program.
```

Now we tell SNACTor how to plot a coverage. We do this using a special set of primitive actions that allow us to assemble a complex ARC/INFO command in a pseudo natural language style. The three primitive actions `say`, `issue` and `send` work very much like write, write-line and write plus send. The objects of the actions are collected until a `send` is encountered. Then the collected items are coerced into a single command and sent to ARC/INFO. This feature uses the extension of the grammar that allows reading of arbitrary quoted strings.

The nice thing about this method is that we do not have to define a primitive action for every slightly different ARC/INFO action that we want to perform. It also allows us to say things such as `issue the coverage` which will result in the right coverage name depending on the coverage that was used to instantiate the plan.

Note that in the plan below we hardcoded `lu-code landuse.lut` even though we use the general `coverage` as an argument to `polygonshades`. This was done for simplicity and because we knew that for this demonstration this plan would be used only to plot the `Landuse` coverage.

Note also that there is a bug in the generation below. Instead of generating `...performing say on the coverage...` it generates `...performing say on Landuse...`, i.e., it grabs an instance instead of a generic reference.

```
: A plan to plot a polygon coverage is to issue "clear"
  and then say "mapextent" and then issue the coverage
  and then say "polygonshades" and then say the coverage
  and then send "lu-code landuse.lut".

I understand that a plan for performing plot on a polygon coverage
is by performing issue on clear and then performing say on mapextent
and then performing issue on Landuse and then performing say
on polygonshades and then performing say on Landuse and then
performing send on lu-code landuse.lut.
```

Here is a precondition for `plotting` (see also the discussion above). A sentence of the form `Before ...ing a P` will attach `P` as a precondition to the action mentioned in the sentence, i.e., before the action can be executed `P` has to be satisfied. If `P` is not satisfied, SNACTor will try to find a plan to achieve `P`.

```
: Before plotting a coverage arcplot must be active.

I understand that before performing plot on a coverage, arcplot
must be active.
```

Here is an effect of `plotting`:

```
: After plotting a coverage the coverage is displayed.

I understand that after performing plot on a coverage, the coverage
is displayed.
```

We cannot say "`After clearing the display no coverage is displayed`" and get the implication that after `clearing the display` all of them are considered as `not displayed`, hence we have to formulate this effect as a conditional plan:

```
: If a coverage is displayed then after clearing the display
```

```
  the coverage is not displayed.

I understand that if a coverage is displayed then after performing
clear on display, the coverage is not displayed.
```

How do you display a coverage? Plot it.

```
: A plan to achieve that a coverage is displayed is to plot the coverage.

I understand that a plan to achieve that a coverage is displayed
is by performing plot on the coverage.
```

The main task of this demonstration is to display the `Landuse` coverage, a map that indicates different uses of the various regions with different colors and shadings, and then point at some region and have ARC/INFO identify it, i.e., say what use the particular region has.

Here we tell SNACTor some things about identifications. First a little trick: We define `this region` as a member of the class `region`. This will allow us later to say "Identify this region" and use a plan that talks about regions in general.

```
: This region is a region.

I understand that this region is a region.
```

The following conditional plan describes how to identify a region. The condition says that it applies only if some polygon coverage is displayed. Then we have a collection of `says` and `sends` to assemble the necessary ARC/INFO command.

There is one new primitive action in this plan: The action `tell`. It allows us to send a message to the user just before the identification command is executed in ARC/INFO (that is why `tell` is the last command before the `send`). The motivation for this is that the user has to be told that action on his or her part is required and that s/he has to move the mouse and click on some polygon.

The approach chosen here to handle the multi-modality of the ARC/INFO SNACTor interaction is very primitive. A real application should handle this more smoothly which is a nontrivial problem to solve.

```
: If a polygon coverage is displayed then
  a plan to identify a region is to say "identify"
  and then say the coverage and then tell
  "Move the mouse over the coverage until cross hairs appear ~%
```

```
        and then choose an area by clicking on it"
  and then send "polys *" and then interpret the region on the coverage.

I understand that if a polygon coverage is displayed then a plan
for performing identify on a region is by performing say on identify
and then performing say on Landuse and then performing tell on
Move the mouse over the coverage until cross hairs appear
   and then choose an area by clicking on it
and then performing send on polys * and then performing interpret
on the region and Landuse.
```

Now we have told SNACTor enough to be able to actually do a few things. First we send a few initialization commands to ARC/INFO to enable it to find coverage files, and to tell it what kind of graphics hardware we are using.

The `Now doing: ...` indicates that the shown command is actually sent to the ARC/INFO process and executed there.

```
: Send "&workspace /u0/grads/hans/getstart/data".

I understand that you want me to perform
  send on &workspace /u0/grads/hans/getstart/data.

Now  doing: &workspace /u0/grads/hans/getstart/data

: Send "&station 9999".

I understand that you want me to perform send on &station 9999.

Now  doing: &station 9999
```

Initially, `arc` is active because we started ARC/INFO with the command `arcloop`.

```
: Arc is active.

I understand that arc is active.
```

Now we tell SNACTor to plot a coverage. Note that in order to plot a coverage it first had to find a plan for doing this. A precondition for plotting was that `arcplot` had to be active. Another plan told it that when `arc` is active `arcplot` can be activated by just issuing the command. This inference led to the execution of `arcplot`. The second `Now doing...` just shows an instantiation of the plan for plotting.

```
: Plot Landuse.

I understand that you want me to perform plot on Landuse.
```

```
Now  doing: arcplot

Now  doing: clear
mapextent Landuse
polygonshades Landuse lu-code landuse.lut
```

At this point ARC/INFO has actually displayed a nice colorful map of the `Landuse` coverage. Now we want to identify some region on this map. Admittedly, `Identify this region` is a very poor way of expressing that the user wants to know something about a region on the current display. But a more appropriate way would require quite a few extensions to the current grammar. That is why we did it in this simple fashion.

Again, SNACTor first tries to find a plan that describes how an identification can be done (see the conditional plan above). It finds one, verifies that some polygon coverage is actually displayed and hence executes an instantiation of the plan. Right after the message the user can move cross hairs over the coverage by moving the mouse. Clicking on an area will result in highlighting of the region and generating a result that contains a row of the table that describes the attributes of the polygons in the `Landuse` coverage. This result is then read by SNACTor, interpreted and translated using the landuse code translation table defined at the beginning of the example, and then the user is told the kind of region s/he pointed at. In our example it was `brushland`.

```
 : Identify this region.

 I understand that you want me to perform identify on this region.

*** Move the mouse over the coverage until cross hairs appear
    and then choose an area by clicking on it ***

 Now  doing: identify Landuse polys *

This area on the coverage Landuse is brushland.
```

This example showed the two-way nature of the interaction between ARC/INFO and SNACTor, i.e., that commands can be sent to ARC/INFO and that results of these commands can be read and interpreted by SNACTor after the command got executed.

Now we clear the display to show that conditional plans are actually conditional.

```
 : Clear the display.

 I understand that you want me to perform clear on display.
```

```
Now  doing: clear
```

At this point the `Landuse` coverage has disappeared and the display is clear. When we try again to identify a region nothing happens because the condition of the plan that describes identification, that is that some polygon coverage must be displayed, is not met.

```
: Identify this region.

I understand that you want me to perform identify on this region.

:
```

4 Conclusion

We have described an interface between ARC/INFO, a geographic information management system, and SNACTor, the SNePS acting component. We demonstrated how ARC/INFO and SNACTor can interact by means of an example interaction developed as part of the project. Here are a few of the many interesting questions that have to be answered before a full fledged natural language frontend to ARC/INFO can be developed, for example, how much of the information contained in the ARC/INFO database has to be copied in one form or the other into the SNACTor knowledge base for proper operation (as done in the mapping between land usage codes and more humanly readable translations thereof), and how much of it can SNACTor find out itself by consulting the ARC/INFO database directly? Another question is how to handle the multi-modal interaction between the ARC/INFO-SNACTor system and the user? Even our simple demonstration showed the need for proper synchronization of natural language input and output, identification of regions with help of a mouse, and presentation of database information in graphic as well as tabular form. A lot more research and the development of bigger and more realistic applications is needed to provide answers to these questions.

5 Acknowledgment

The authors thank David Mark for his encouragement, advice, and help in obtaining funding.

References

[1] "AML Users Guide", manual, ESRI, Inc., Redlands, CA

[2] "ARC/INFO Users Guide", manuals, Vols.1&2, ESRI, Inc., Redlands, CA

[3] "Understanding GIS: The ARC/INFO Method", an introductory tutorial, ESRI, Inc., Redlands, CA

[4] Kumar D., Ali S.S., Haas J. and Shapiro S.C. "The SNePS acting system", in K.E. Bettinger and G. Srikantan, eds., *Proceedings of the Fifth Annual University at Buffalo Graduate Conference on Computer Science*, pp. 91-100, 1990

[5] Shapiro S.C. "The SNePS semantic network processing system", in N.V. Findler, ed., *Associative Networks: The Representation and Use of Knowledge by Computers*, pp. 179-203, Academic Press, New York, 1979

[6] Shapiro S.C., Rapaport W.J. "SNePS considered as a fully intensional propositional semantic network", in N. Cercone and G. McCalla, eds., *The Knowledge Frontier*, pp. 263-315, Springer Verlag, New York, 1987

[7] Shapiro S.C. "Representing Plans and Acts", in *Proceedings of the Third Annual Workshop on Conceptual Graphs,* The American Association for Artificial Intelligence, Menlo Park, CA, 1988

[8] Shapiro S.C., Woolf B., Kumar D., Ali S.S., Sibun P., Forster D., Anderson S., Pustejovesky J. and Haas J. "Discussing, using, and recognizing plans-Project Report", Technical Report, North-East Artificial Intelligence Consortium, 1989

A Appendix

A.1 Interface Installation

We will assume that the ARC/INFO system has been installed and that you have access to an up-to-date version of SNePS-2.1. You need to have the following files to set up proper communication between ARC/INFO and SNACTor:

```
command-loop.aml      ....the ARC/INFO command loop
arcloop               ....shell script that starts ARC plus command loop
driver.lisp           ....the Lisp side of the interface
```

The following files are necessary to run SNACTor and have it communicate with ARC/INFO, as well as running the example interaction described in this document:

```
arcgram.dat           ....the GATN grammar used in the example interaction
arclex.dat            ....the lexicon used in the example interaction
arcinfo.lisp          ....defines primitive actions used in example
arcdemo.sneps21       ....the demo file defining the example interaction
load-arc-snactor.lisp ....a load file for the ARC/INFO - SNACTor system
nlib.lisp             ....library functions for the parser
plantrace.lisp        ....plan trace utilities
snactor-imports.lisp  ....symbols imported into SNACTor
snactor21.lisp        ....main SNACTor code
.sunview              ....a simple .sunview file
```

In the current release of SNePS-2.1 all these files reside in the `.../demo/snactor` directory of the SNePS-2.1 tree. For the following let us assume you have copied all the files into the directory `/u0/snactor`. You have to decide what directory to use as an interface directory. We could use `/u0/snactor` but it is better to have a separate directory. Let us assume its name is `/u0/interface`. NOTE: Use directory names that do not contain upper-case letters, because ARC/INFO cannot handle them!

In order to install the system you have to edit pathname definitions in the following files:

A.1.1 `arcloop`

`arcloop` is a shell script that starts up ARC/INFO and then starts the command loop inside arc. There are two shell variables whose values you have to edit according to the directories you chose. Here is what they should look like for our example values:

```
set interfacedir = /u0/interface
set snactordir   = /u0/snactor
```

A.1.2 `command-loop.aml`

This is the AML file that defines the ARC/INFO command loop. If you always run `arcloop` to run ARC/INFO you do not have to edit this file. However, if you want to run ARC/INFO in an interactive fashion and you want to execute `command-loop.aml` you might want to edit the value of the variable `default-interface`. For our example it should look like this:

```
&s default-interface /u0/interface
```

The value of this variable will be used as the interface directory if no directory was supplied to `command-loop.aml` as an argument.

A.1.3 `driver.lisp`

This file defines the Lisp side of the communication. You have to edit the definition of the variable `*interface-directory*`. For our values it should look like this:

```
(defvar *interface-directory* "/u0/interface")
```

You can achieve the same effect without actually editing the file if you execute the following form after you have loaded SNePS-2.1 but before you load SNACTor:

```
(setq snepsul::*interface-directory* "/u0/interface")
```

A.1.4 `arcdemo.sneps21`

This modification is optional and only necessary if you want to run the example demonstration described in this report. What you have to modify is the pathname of the workspace where the example coverages can be found. Suppose these coverages are in `/u0/training/getstart/data` then you have to change the sentence that defines ARC/INFO's initial state to

```
Send "&workspace /u0/training/getstart/data".
```

A.1.5 Telling SNePS-2.1 where to find SNACTor files

If you copied the ARC/INFO - SNACTor files in a directory of your own you have to tell SNePS-2.1 the proper mapping between the logical pathname `sneps21:demo;snactor;` and its physical location. You can do this by setting the following variable before you load SNePS-2.1:

```
(setq user::*sneps21-user-translations*
      '(("demo;snactor;*.*.*" "/u0/snactor/*.*")))
```

A.2 Starting Up

We will assume that ARC/INFO has been installed for you and that you have access to a set of database files that define coverages. In the example below we will use the training set coverages used in [3].

Assuming that you have installed the SNACTor files according to the instructions given above, you can start interaction of ARC/INFO and SNACTor by doing the following:

1) Start ARC/INFO. Assuming that you work on a Sun workstation you have to start `sunview` first. So do this:

```
% cd /u0/snactor   # or whatever your snactor directory is
% sunview
% arcloop          # in a shell window in sunview
```

IMPORTANT: `arc` and its `command-loop.aml` have to be active BEFORE the first call to `execute-command` in SNACTor. So, make sure that you always start `arc` first.

2) Then start your favorite Common Lisp and load SNePS-2.1 and SNACTor by doing the following:

```
> (load "~snwiz/bin/sneps21") ;; or whatever is necessary to load SNePS-2.1
> (load "sneps21:demo;snactor;load-arc-snactor")
```

This will load SNePS-2.1, and then load all the files that define SNACTor. These files will also get compiled unless you already have compiled versions. Now you are ready to go.