Integration of the Truckworld with CLIPS

Umur Ozkul
Levent Akin
Ethem Alpaydin
Ufuk Caglayan
Selahattin Kuru

Department of Computer Engineering
Bogazici University

e-mail: ozkul@boun.edu.tr

June 8, 1994

Abstract

The potential mismatch between benchmark scores and performance on real tasks leads us to use testbeds to compare and evaluate expert system shells. In this paper we describe the resulting integration architecture in integrating a testbed, Truckworld and an expert system shell, CLIPS. We also propose truck agent designs.

Contents

List of Figures

Introduction

Expert system shells[[5]] are software packages containing a generic inference engine, a user interface, and a collection of other tools that enable users to develop and use expert systems. Using the shell a knowledge engineer can create and modify knowledge bases. The inference engine contains inference and control paradigms. The interface allows users to interact with the shell.
The process of evaluating expert system shells is difficult, and is not standardized. Benchmarks are a simple and useful method of evaluation. Nevertheless, the potential mismatch between benchmark scores and performance on real tasks leads us to use testbeds to evaluate expert system shells.
We started with integrating the testbed, Truckworld and the expert system shell CLIPS. In this paper, we describe the resulting integration architecture, and ideas about truck agents. We extended Truckworld[[2]] so that we can easily integrate it with expert system shells.

The Truckworld and CLIPS

The Truckworld[[2]] is a multi-agent testbed designed to test theories of reactive execution. The main commitment is to provide a realistic world for its agents. An agent is a truck consisting of two arms, two cargo bays, several sensors, and various other components like a fuel tank, a set of tires, and direction and speed

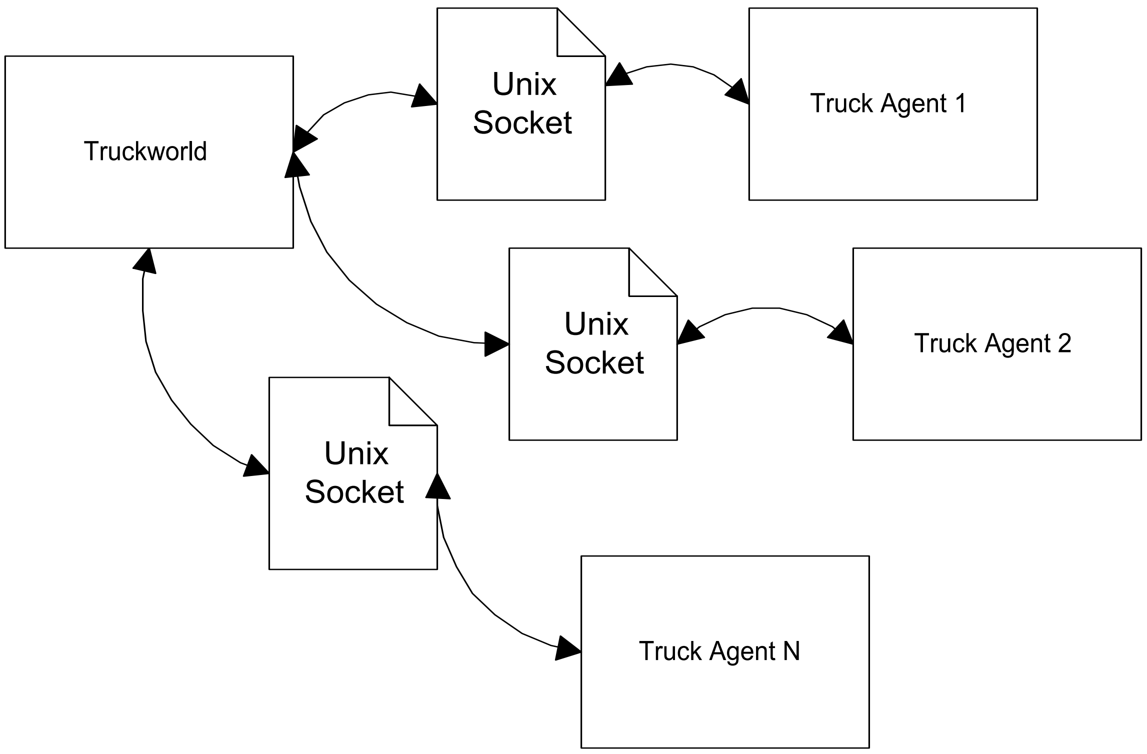

Figure 1 Multi-agent architecture of the Truckworld

controllers. It operates in a world of roads and locations. Objects populate locations connected by roads. Exogenous events like rainstorms occur periodically in the world. The Truckworld also contains facilities for communication and other interaction among agents.
CLIPS as an expert system shell is a software package containing a generic inference engine, a user interface, and a collection of other tools that enable users to develop and use expert systems. Using the shell's tools, a knowledge engineer can develop new knowledge bases, and can structure, add, delete, and modify the knowledge contained in them. The inference engine contains inference and control paradigms. The shell's interface allows users to interact with the expert system, extract knowledge from it, and request explanations as to the reasoning behind the system's conclusions. We use CLIPS to implement Truckworld agents.

Multi-agents of the Truckworld

CLIPS has two operational modes: Single agent, and multi-agent. In multi-agent mode, the Truckworld operates as server LISP process (Truckworld is implemented in LISP). Agents are client LISP processes. Agents and the Truckworld communicate through UNIX sockets [[6]] as illustrated in Figure 1. In single agent mode, the Truckworld and the agent are in a single process without using a socket.
The Truckworld is supplied with three truck agents implemented:

1. **Raw Client:** The truck command you type on the terminal window is simply sent to the Truckworld. Any data sent from the Truckworld is printed on the terminal window. There is no synchronization between input and output.
2. **Friendly Client:** Input and output are synchronized. Thus, when you enter a truck command, the execution results of the command are printed on the terminal window.
3. **Functional Client:** The functional client is like the friendly client but the communication with the Truckworld is now through a LISP function instead of a terminal window. The user can issue calls to
   (execute-truck-command <truck-command>)
   which will then act like the friendly client, advancing until the truck command has completed. The return value of the function is the response from the Truckworld.

The function (execute-truck-command …) is provided so that it can be used to execute commands from a planner.

Alternative Approaches to Integration

Now we can consider integrating the Truckworld and CLIPS (or other expert system shells). The first approach is obvious: Call the function supplied with the functional client (Figure 2).

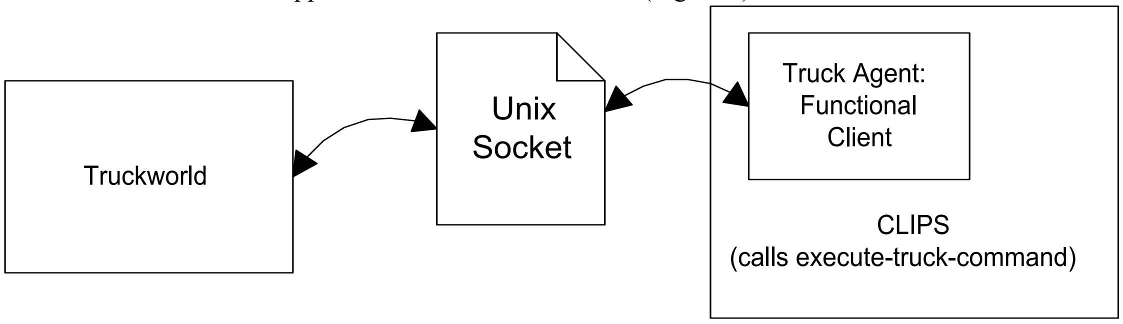

Figure 2 Embedding the functional client into CLIPS

This approach has many drawbacks. Calling LISP functions is not easy. In the LISP interpreter we use, this requires linking CLIPS and the friendly client into a single executable. CLIPS is embeddable into other packages but this might not be true for other expert system shells. Furthermore there is no obvious way of converting the LISP data structure into a form acceptable by CLIPS or any other computer language. All you get from a LISP function is a pointer to an s-expression. The callability of LISP functions from other languages is meant for functions returning primitive data types. Also, neither CLIPS nor other shells can process recursive data structures.
To overcome the barrier of converting LISP data structures, we can instead embed CLIPS into the truck agent. Then the main module of the truck agent issues calls to the function execute-truck-command and to functions supplied by CLIPS (Figure 3). Still this requires linking the LISP interpreter and CLIPS. However embeddability is not possible for every expert system shell.

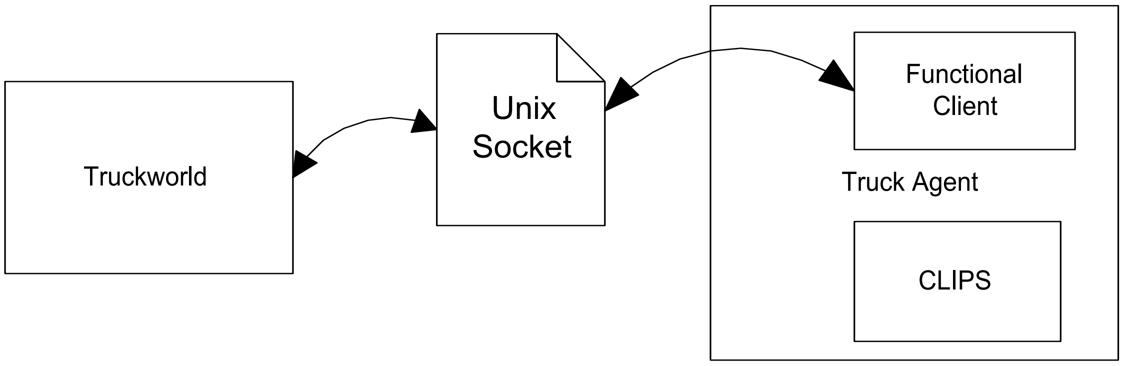

Figure 3 Embedding a functional client and CLIPS into a truck agent.

Till now, this approach has been considered to escape from implementing the friendly client again. Now if you implement the friendly client in C all over then you can link it to CLIPS (Figure 4). Still this approach has a drawback. It is neither easy nor possible to link external functions with every expert system shell. However, friendly client in C seemed to be the obvious approach at the beginning. Steve Hanks [[3]] confirmed this approach (We still believe it would be easy to code the functional client in C would be easy).

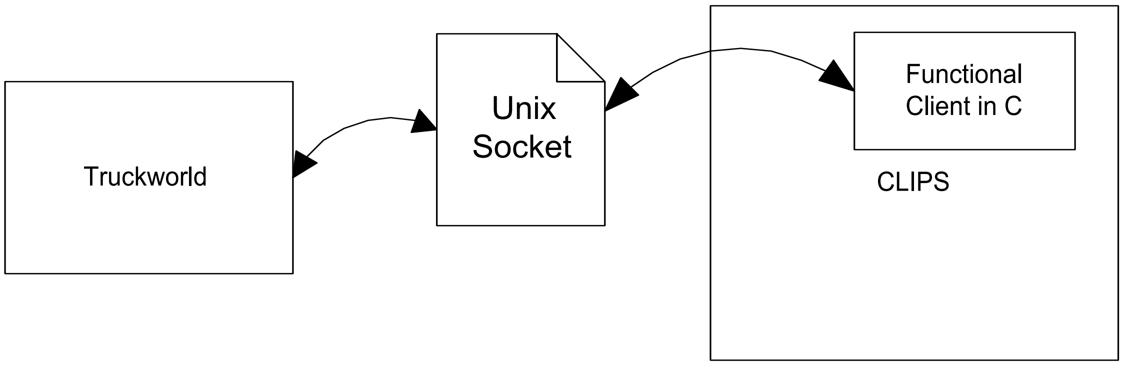

Figure 4 Functional client in C linked to CLIPS

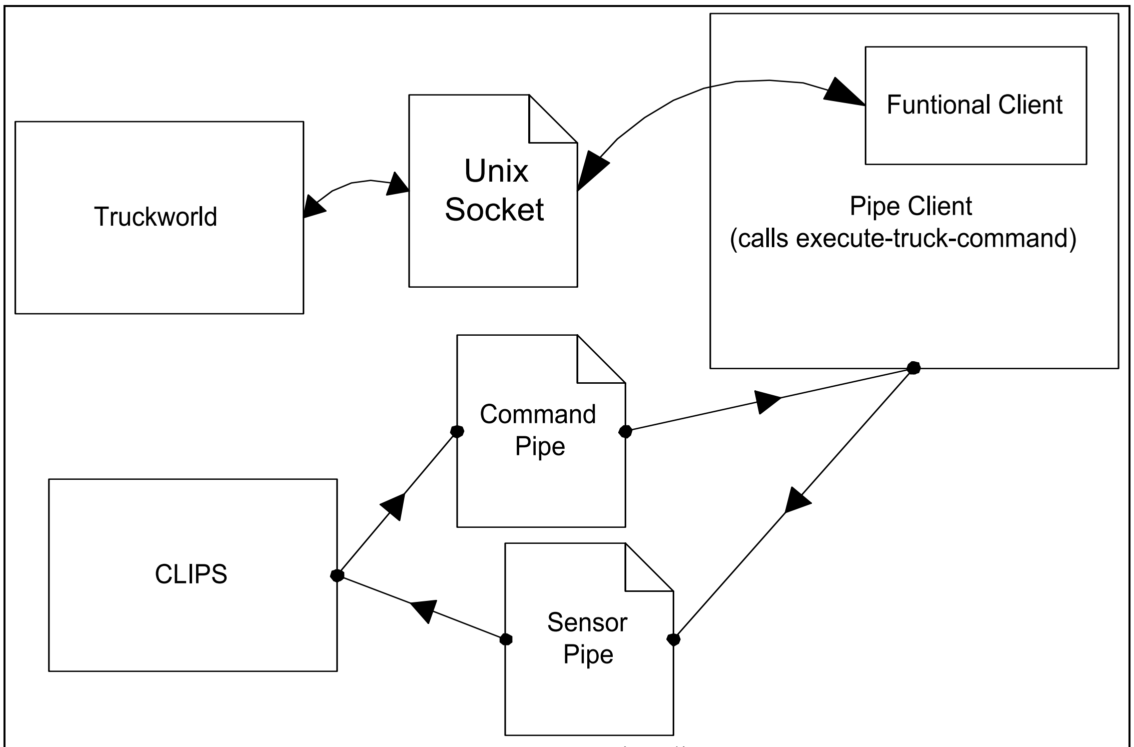

Figure 5 Pipe client

Nevertheless, learning socket programming worth us a month's struggle and many workstation shutdowns. The socket program developed is now used as a demonstration in a network lesson. Still there were much to code. Therefore we concluded for our final approach**:** pipe client. The pipe client does not require external function linkage from the expert system shells. Therefore you can use it with most expert system shells. The only requirement that the shell should have file IO operators.

Pipe Client

The pipe client is built upon the functional client and pipes. In UNIX you can define pipes with the command:

```
mknod <pipe-name> p.
```

From then on, you use the pipe as if an ordinary file. The source process opens the pipe write only. The sink process opens the pipe read only. The pipe client (Figure 5) loops fulfilling the four jobs in order.

1. Read from the command pipe
2. Call execute-truck-command
3. Translate the result in a readable form for the specific shell.
4. Write the translated result into the sensor pipe.

Having written a command to the command pipe, CLIPS reads from the sensor pipe and asserts into its knowledge base the lines read. Now we can give a concrete example of this process.

Sample Usage of Truck Commands from CLIPS

1. Create the pipes necessary for the communication:

mknod ~/commandPipe p,
mknod ~/sensorPipe p.

2. Start the Truckworld [[2]] (in AllegroCommon LISP):

```
>   :cd ~/Truckworld
>   :cl loader.lisp
>   (load-integrated-simulator)
```

3. Start the pipe client(in AllegroCommon LISP):

```
>   :cl piper.lisp
>   (run-demo-functional-client :display "stardust")
>   (process-truck-pipes :iname "~/commandPipe" :oname "~/sensorPipe :answer #'answer-clips)
```

answer-clips is the name of the function that translates the Truckworld data in a suitable form for CLIPS. It is given as a parameter so that you can customize the process easily by supplying translators suitable for other expert system shells. **:cl** means "compile and load". In *piper.lisp,* the commands following it reside. **(run-demo-functional-client :display "stardust")** starts a demo-world on the X display *"startdust"* and a functional client. **process-truck-pipes** starts the pipe client accessing the functional client and the pipes given. Details about variations in starting the Truckworld and its clients are described in [[2]].

4. Let's see the CLIPS side. The truck agent implemented in CLIPS opens the pipes.

```
>   (bind ?commandFile "~/commandPipe")
>   (bind ?sensorFile "~/sensorPipe")
>   (open ?commandFile commandStream "w")
>   (open ?sensorFile sensorStream "r")
```

5. Now you start issuing truck commands:

```
(execute-truck-command truck-read truck-sensor).
```

Here **execute-truck-command** is a CLIPS function supplied by us. A truck-command without parenthesis follows the function name. The syntax of the command is as described in [[2]].
truck-read truck-sensor read the sensor named *truck-sensor* built into the truck. Thus the truck sees the objects around it. Instead of returning values, the function **execute-truck-command** asserts facts. When you issue a

```
(facts)
```

command in clips you will see additional facts asserted due to the command above:

```
((time 10) (sensor truck-sensor)(position 0) (kind rock))
...
```

The fact above is explicit. During time *10*, via the sensor *truck-sensor*, at position *0*, the truck has seen a *rock*.

Software Architecture

The integration software consists of collections of functions for the communicating parts: pipe client and truck agent. Data flow during the communication of the Truckworld and the CLIPS truck agent is illustrated in Figure 6. We will describe the relevant code in the following sections.

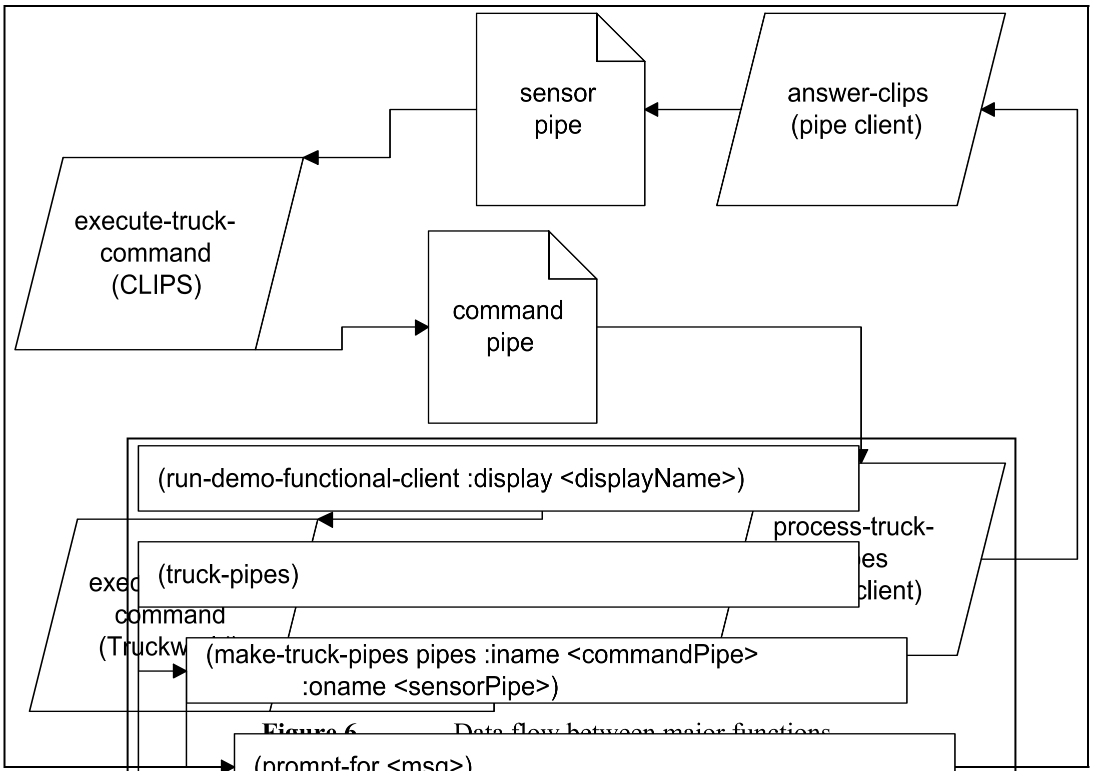

Figure 6 Data flow between major functions

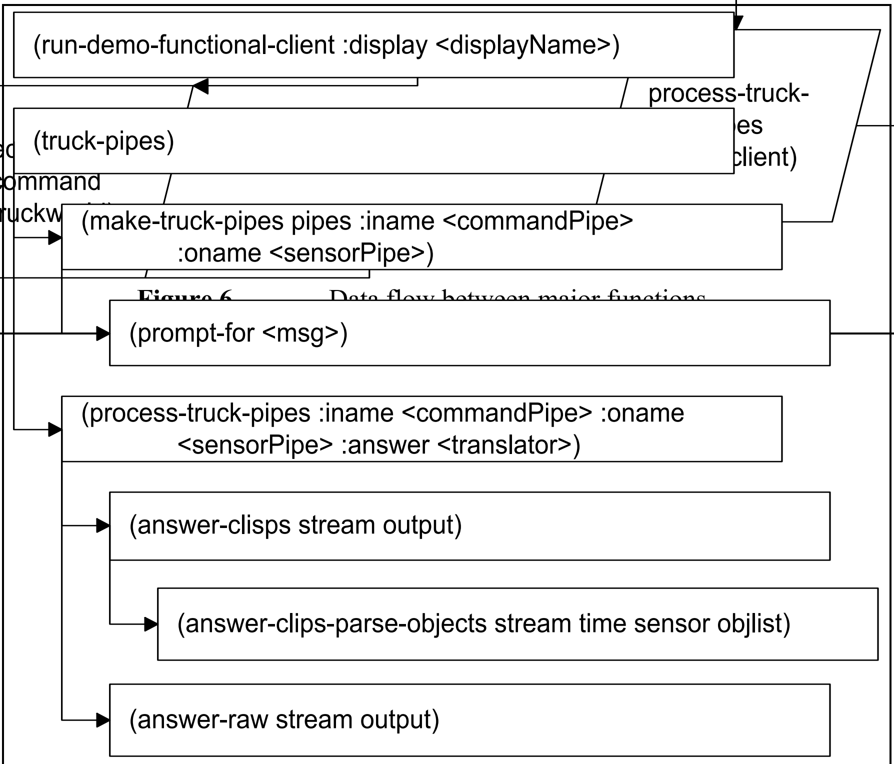
(run-demo-functional-client :display <displayName>)
(truck-pipes)
(make-truck-pipes pipes :iname <commandPipe> :oname <sensorPipe>)
(prompt-for <msg>)
(process-truck-pipes :iname <commandPipe> :oname <sensorPipe> :answer <translator>)
(answer-clisps stream output)
(answer-clips-parse-objects stream time sensor objlist)
(answer-raw stream output)

Figure 7 Call tree of functions in pipe client

Functions in Pipe Client

The functions that constitute the implementation of pipe client are described in alphabetical order. To aid inspection of source code, a call tree of functions is in Figure 7. The source code of the functions is listed in "Appendix A - Source Code for Pipe Client".

(answer-clips-parse-objects stream time sensor objlist)

This is a function to aid the implementation of "*(answer-clips stream output)*". *objlist* is the list of objects reported by *sensor* at *time*. *stream* is the pipe or file where the function output will be printed. *objlist* is decomposed and the description of each object is printed on a separate lined. *sensor* and *time* is included on each line. CLIPS can read and assert a properly formatted line into its knowledge base.

(answer-clips stream output)

output is data generated by the Truckworld in response to a truck command. *stream* is the pipe or file where *output* will be printed. This function transforms *output* into a form readable by CLIPS. The translation made is detailed in "Translation of Truckworld Output for CLIPS". To use this function as the translator, you pass it to the function *"(process-truck-pipes …)"*.

(answer-raw stream output)

This is an identity translation function. This function prints *output* on *stream.* You can use this function as an alternative of *"(answer-clips ...)"*. This function is useful in debugging the standalone operation of the pipe client as described in "Connecting a Terminal to a Pipe Client".

(make-truck-pipes pipes :iname <commandPipe> :oname <sensorPipe>)

This function calls UNIX commands to create *commandPipe* and *sensorPipe*. Currently, we prefer to create them from a UNIX shell. This function is included for completeness.

(process-truck-pipes :iname <commandPipe> :oname <sensorPipe> :answer <translator>)

This function starts *translator* on *commandPipe* and *sensorPipe*. Truck commands from *commandPipe* are fed to the Truckworld and its response is printed on *sensorPipe*. Currently available translators are *(answer-raw)* and *(answer-clips)*.

(prompt-for <msg>)

This function prints *msg* on the terminal and returns the answer typed.

(run-demo-functional-client :display <displayName>)

displayName is the X display where the Truckworld will open its window. This function creates a functional client[[2]] using the demo world and demo truck supplied with the Truckworld. Functional clients with different trucks on different worlds you supply can be created as described in [[2]].

(truck-pipes)

This function is the main entry point of the pipe client. It first creates the pipes and calls *(process-truck-pipes…)*. However, there is no need to recreate pipes before each truck session as the pipes are persistent. Therefore we generally use *(process-truck-pipes…)* as the main entry point.

Rules, Templates and Functions in Truck Agent

Here were describe the least required implementation of a CLIPS truck agent. The important parts are *position-info* -- type declaration of the incoming Truckworld data, *execute-truck-command* -- the function which communicates with the Truckworld. The rules that form the least required entry and exit protocol of a truck agent are listed in "Appendix B - Least Source Code for a Truck Agent".

position-info

```
(deftemplate position-info
        (field node)
        (field time)
        (field sensor)
        (multifield position)
        (field kind)
        (field sensor-id)
        (field color)
        (field bigness))
```

Deftemplate is analogous to a record structure in Pascal. That is, deftemplate defines a group of related fields in a pattern in the same way in which a Pascal record is a group of related data. In general, many types of objects have a natural structure in which their attributes can be related. Here the template *position-info* defines data about an object in Truckworld. The function *execute-truck-command* asserts *position-info* facts into its knowledge base if the last executed command is reading through a sensor in Truckworld. One important property of a template is that its fields are optional. Not specifying some of the field does not pose a problem. Getting partial data on an object is not an exceptional case because sensors report you partial data. If you use a color sensor you get

```
((sensor color-sensor) (time ...) (color red) (position 10)).
```

But if you use the truck-sensor you get

```
((sensor truck-sensor) (time ...) (kind rock) (position 10)).
```

It is your responsibility to infer that there is a red rock at position 10. It is also your responsibility to find and use proper sensors, to decide the quality of data you require. There might be *missing fields* in the current definition but they are easily added as the program is used.
Now let us describe the fields individually.

.1 *field node*

Truckworld consists of nodes connected with nodes. Depending on the will of a truck, it might want to remember in which node it has seen an object. However, node field is not reported by sensors. You are to fill in the node field if necessary. For example you might be implementing a tourist truck that is simply traveling the world and learning places of things. Another example is that a truck becomes aware that it needs an object it has previously seen. It needs to remember where it has seen the object.

.2 *field time*

This specifies when the object is seen. In the Truckworld the world changes due exogenous events and the actions of trucks in the world. If the *time* is too old, your data about an object is possibly wrong. In some cases it need not be too old. Around you, there might be another truck changing places of things.

.3 *field sensor*

Of course, a truck has to know trough which sensor it has seen an object. There is a typical example of the case. The truck has sensed some object through the *bad-color-sensor*. Then it has had the opportunity to use the *color-sensor*. The data reported by the *bad-color-sensor* is more erroneous than that of the *color-sensor.* It is possible that the truck senses a blue object as green. Therefore a truck must be able to make preference between facts from different sensors.

.4 *multifield position*

This field tells where an object is. A Truckworld node is an array of 16 positions. Thus *position* is a number among 0 and 15 in most cases. However the object might be in a container object (box, truck, arm). In this case *position* is a series of numbers. For example:

(position 10 3 6).

That is, the object is at position 6 of a container that is at position 3 of a container that is at position 10.

.5 *field kind*

kind defines the object seen: truck, rock, box, etc.

.6 *field sensor-id*

If the kind of the object is *sensor*, *sensor-id* uniquely identifies the object. Otherwise this field is not reported.

.7 *field color*

As the name implies. What about a truck that collects blue rocks, and then exchanges them with the truck selling fuel for blue rocks?

.8 *field bigness*

Bigness is in fact the weight of an object. If an object is too heavy, a truck might not be able to hold it.

execute-truck-command

Using this function you command a truck. The implementation of this function is trivial as listed in “Appendix B - Least Source Code for a Truck Agent” for most of the job is done by the pipe client. The syntax of the commands is still as described in the Truckworld manual[[2]] and “Frequent Tasks of a Truck”. For example, if a command is defined as

(arm-1 move 5)

in the Truckworld manual, you should do the same thing in CLIPS by calling

(execute-truck-command arm-1 move 5).

Thus it is trivial to drive a truck through CLIPS. The side effects of this function is described in “Translation of Truckworld Output for CLIPS”.

Translation of Truckworld Output for CLIPS

Here we will describe and compare the output of the Truckworld and the output of the pipe client for CLIPS[[4]]. There are three types of output from the Truckworld according to the command given:

1. The command failed (e.g. grasping a too heavy object): The output is

(ERROR)

2. The command is not a sensor reading: The Truckworld returns the current simulation time.

(10)

Assume the current simulation time is 20. If you issue the command
Truckworld> (wait 5),
you will get

(25)

3. The command is for a reading. As an example, assume we have positioned the truck in the SENSOR-NODE location of the *demo-world*, and we are ready to use the built-in *truck-sensor* to sense other objects at that location.
   Truckworld> (truck-set truck-sensor)
   Truckworld> (wait 1)
   Truckworld> (truck-read truck-sensor)
   You get the response
   (3 (sensor truck-sensor
   (((position 0) (kind roadsign) (direction n) (condition good) (length 20))
   ...
   ((position 2) (kind rock))
   ...)))
   The very first number is the current simulation time as in case two. This is a tree structure describing objects seen through *truck-sensor* at time *3*.

Now we can examine what happens when we command through CLIPS.

1. The command failed. The fact assert in the knowledge base is

   (time ERROR).

2. The command is not a sensor reading: The fact asserted in the knowledge base is

   (time 10)

   Assume the current simulation time is 20. If you issue the command
   (execute-truck-command wait 5)
   you will get the fact asserted

   (time 25)

3. The command is for a reading. As an example, assume we have positioned the truck in the SENSOR-NODE location of the *demo-world*, and we are ready to use the built-in *truck-sensor* to sense other objects at that location.
   (execute-truck-command truck-set truck-sensor)
   (execute-truck-command wait 1)
   (execute-truck-command truck-read truck-sensor)
   You get the facts asserted
   ((time 3) (sensor truck-sensor) (position 0) (kind roadsign) (direction n) (condition good) (length 20))
   ...
   ((time 3) (sensor truck-sensor)(position 2) (kind rock))
   ...
   You can easily compare the facts asserted with the raw Truckworld output above. We have just flattened the tree structure of the Truckworld output for CLIPS cannot handle tree structures. Otherwise we are faithful to the original representation of the data.

Connecting a Terminal to a Pipe Client

Instead of CLIPS driving a pipe client, you might want to drive the pipe client. That is, you take the role of CLIPS. Thus, you might inspect the response of the Truckworld to the truck commands typed. This might be helpful before starting to design a truck agent, or to design a translator for different expert system shell.
For example, after starting up a pipe client with pipes named “~/command” and “~/sensor”, type the UNIX commands below.

```
>   cmdtool -I "cat >~/command" &
>   cmdtool -I "cat ~/sensor" &
```

Now you have gotten two X windows. One for you to type truck commands, the other to display the response of the Truckworld.

Observing the Communication between the Truckworld and the Truck Agent.

This is one of the most useful facilities of the pipe client. Without modifying anything in the pipe client and the truck agent, you can redirect the data flowing through them to X terminals. To do this, instead of creating a pair of pipes, create two pairs of pipes.

```
>   mknod command1 p
>   mknod sensor1 p
>   mknod command2 p
>   mknod sensor2 p
```

Now execute the script below to connect the pipe pairs.

```
>   cat command2 | tee command 1 &
>   cat sensor1 | tee sensor2 &
```

Then start the pipe client with *command1* and *sensor1* start the truck agent in CLIPS with *command2* and *sensor2*. As the truck agent comes to life, the data flow between the truck agent and the pipe client is displayed on your terminal. In fact, you have established a pipe connection as illustrated in Figure 8.

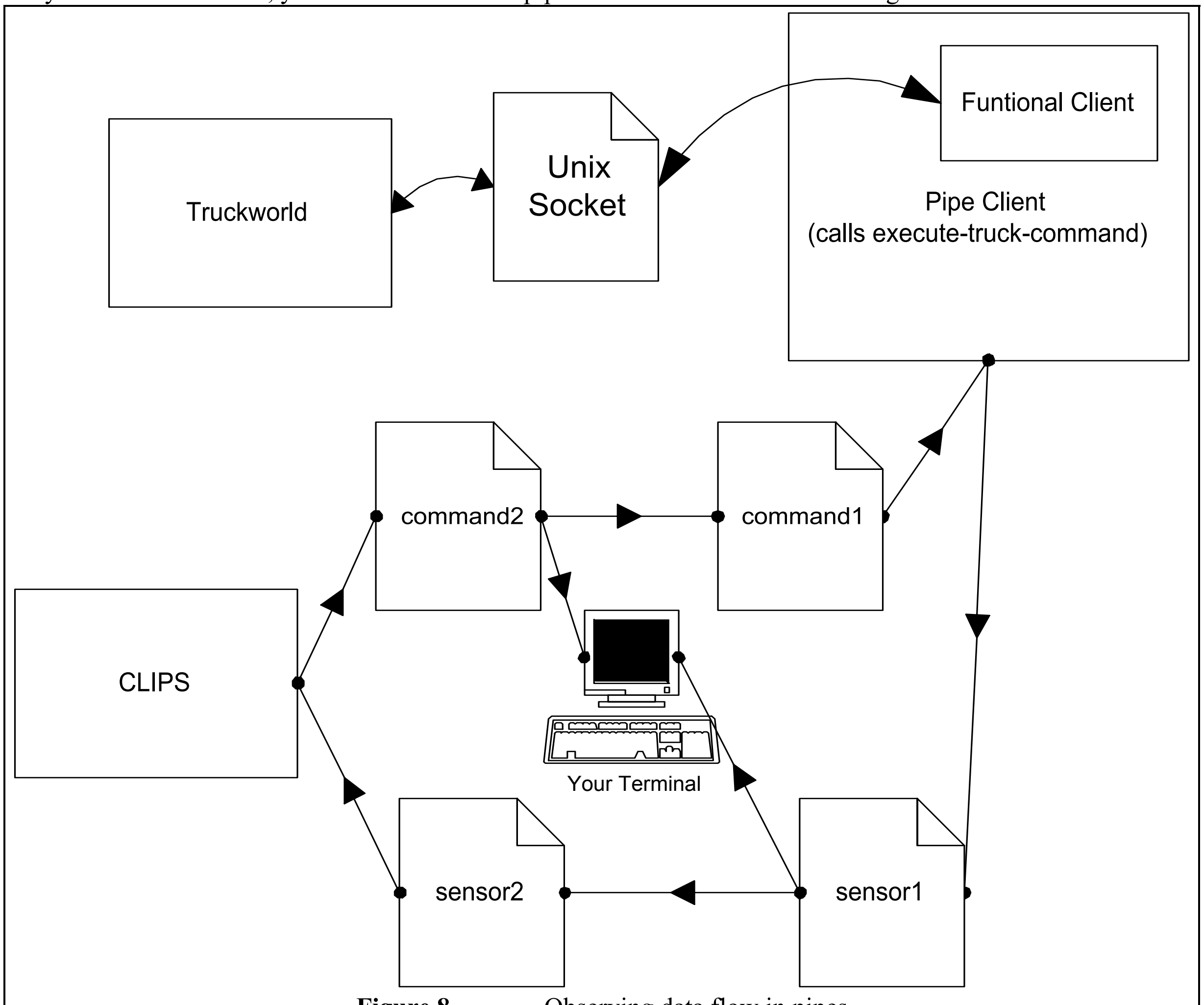

Figure 8 Observing data flow in pipes

Frequent Tasks of a Truck

- When you are designing a truck agent, it is wise to drive the truck yourself acting the role of a truck. Therefore here we list the frequent tasks performed by a truck. You can easily paste the commands listed to drive your truck. Rather then issuing primitive truck commands, it is easier to manipulate the truck using these high level command groups. You might visualize groups different from the ones described here as high level commands. High level commands give a clue of rules you should implement to try out basic reactive planning (See "Reactive Planning"). Let us call these command groups *compound motions*. Then your truck will decide to execute a *compound motion*. A *compound motion* can be interruptable. Hence, if the truck changes its mind in the midst of a *compound motion*, it might abort the motion or continue it later. Remember that the CLIPS equivalent of the command *(wait 1)* is *(execute-truck-command wait 1)*. Similarly for other commands.
- The truck has a built-in *truck-sensor*. The truck can see the objects around it using *truck-sensor.*
  (truck-set truck-sensor)(wait 1)(truck-read truck-sensor)
- Assume that the truck has found an X-ray machine at location 12. Our truck does not want to load the machine onto itself. The sets and reads the machine in its place using its arms.
  (arm-move arm-1 outside)(arm-move arm-1 12)(arm-set arm-1 nil)

(wait 5)(arm-read arm-1 nil)(arm-move arm-1 folded)
In the Truckworld you can find other kinds of sensors scattered around. You can use them in the same fashion.

- The truck is out of fuel and there is a fuel drum at position 6. The truck grasps the fuel drum, pours the fuel drum into its fuel tank, and puts the drum back into its place.
  (arm-move arm-1 outside)(arm-move arm-1 6)(arm-grasp arm-1)(arm-move arm-1 folded)
  (arm-move arm-1 inside)(arm-move arm-1 fuel-tank)(arm-pour arm-1 0)(arm-move arm-1 folded)
  (arm-move arm-1 outside)(arm-move arm-1 6)(arm-ungrasp arm-1 0)(arm-move arm-1 folded)(arm-move arm-1-1 outside)(arm-move arm-1 6)(arm-ungrasp arm-1 0)(arm-move arm-1 folded)
- The truck has a number of built-in internal sensors. Let out truck learn its speed, for example.
  (truck-set speed-sensor)(wait 1)(truck-read speed-sensor)
- Our truck wants to see what it is holding in its first arm.
  (truck-set arm-1-sensor)(wait 1)(truck-read arm-1-sensor)
- How much fuel do the truck have?
  (truck-set fuel-tank-sensor)(wait 1)(truck-read fuel-tank-sensor)
- Our truck starts moving north at medium speed.
  (change-heading N)
  (change-speed medium)
  (truck-move)

Reactive Planning

It is easier to manipulate the truck using these high level command groups. You might visualize groups different from the ones described here as high level commands. High level commands give a clue of rules you should implement to try out basic reactive planning. Let us call these command groups *compound motions*. Then your truck will decide to execute a *compound motion*. A *compound motion* can be interruptable. Hence, if the truck changes its mind in the midst of a *compound motion*, it might abort the motion or continue it later.
Let us establish a simple example. Our truck does not know its fuel level then attempts to learn it. (Another motivation to learn the fuel level might that the truck has not checked its fuel level for a long time.)

```
(defrule unknown-fuel-level
        (not (fuel-level ?))
->
        (assert (compound-read-sensor fuel-tank-sensor)))
```

The truck has decided to check fuel. Instead of executing the necessary commands procedurally. It fires successive rules. Hence higher priority rules might interrupt the sequence causing the truck to change its mind. First the truck sets the required sensor, which is *fuel-tank-sensor* in this case.

```
(defrule set-sensor
        (not (time ERROR)); Previous command was successful
        (pipes-open); Communication with the Truckworld OK.
        (compound-read-sensor ?sensor)
->
        (execute-truck-command truck-set ?sensor)
        (assert (?sensor was-set))
```

Now the truck waits for the sensor to operate.

```
(defrule wait-sensor
        (not (time ERROR)); Previous command was successful
        (pipes-open); Communication with the Truckworld OK.
        (compound-read-sensor ?sensor)
        ?r <- (?sensor was-set)
->
        (execute-truck-command wait 1)
        (assert (?sensor was-waited)
        (retract ?r))
```

We have issued a (wait 1) command. In a realistic program the waiting duration should be determined according to the sensor, urgency and importance of the job. Finally it reads the sensor

```
(defrule read-sensor
        (not (time ERROR)); Previous command was successful
        (pipes-open); Communication with the Truckworld OK.
        (compound-read-sensor ?sensor)
        ?r2 <- (?sensor was-waited)
->
        (execute-truck-command truck-read ?sensor)
        (assert (?sensor was-read))
        (retract ?r2))
```

We can now complete the job.

```
(defrule assert-fuel-level
        (not (time ERROR))
        ?r1 <- (compound-read-sensor ?sensor)
        ?r2 <- (?sensor was-read)
        ?r3 <- (position-info (fuel-level ?l))
->
        (assert (fuel-level ?l))
        (retract ?r1 ?r2 ?r3))
```

The truck has gotten curious about its fuel level, and it now knows it. To fire a series of commands, have asserted the fact *(compound-read-sensor fuel-tank-sensor)*. As you guess this story does not end here. Suppose the learned that it has not much fuel. It would fire another compound rule to look around for a fuel tank.

```
        (assert (compound-read-sensor truck-sensor))
```

If the truck has not seen a tank, it should leave the terrain to look for one. We have gotten a hungry truck. It might have other aims at the time. But food is important so we should set the priorities of the rules accordingly. Now let us think the other case: The truck has seen a fuel tank. In a similar fashion to the one above, the truck will assert a compound behavior to fill its fuel tank. The series of required commands have been listed in "Frequent Tasks of a Truck".

```
        (assert (compound-pour-tank <positionOfTank>))
```

But is it so simple? No. Remember that if an object is too heavy for the truck, it might be unable to grasp it (Also another truck might steal it, the truck's arms might be full). Therefore the truck will sometime check if it has grabbed the tank with its arm.

```
        (assert (compound-read-sensor arm-1-sensor))
```

What we should do if we had grasped the fuel tank? Depending on the destiny of the Truck, the quest for intelligence really becomes tough here.
We have tried to do some simple reactive planning. Thing might be very different from procedural programming. No plan can be expected to succeed, and plans might be changed on the fly. Rather you are trying to run ahead of time. This might be important in a war of trucks. It seems that, although not human-like, we might happily create **bug-like** trucks. There are many hard decisions still there. How should we decide which *compound-behavior* is more important at the current time?

Ideas about Truck Scenarios

Our initial motivation is to form a test criterion of expert system shells. Truck agents should be implemented such that different implementations with different shells should worth comparing. You might exercise simple procedural programming with them as well. However, that is not expected to form a realistic test case with expert system shells. Simple reactive planning can be a good starting point to design test cases that creates a realistic stress on the shells tested. Thus, we might follow some primary problems to implement.
First, you might be implementing a tourist truck that is simply traveling the world and learning places of things. Considering the fueling scenario described in "Reactive Planning", it is sufficiently complicated.

Second, in the Truckworld the world changes due exogenous events and the actions of trucks in the world. If the *time* is too old, your data about an object is possibly wrong. In some cases it need not be too old. Around you, there might be another truck changing places of things. You might design a time aware truck.
What about a truck that collects blue rocks, and then exchanges them with the truck selling fuel for blue rocks? Another example is that a truck becomes aware that it needs an object it has previously seen. It needs to remember where it has seen the object.
Or you truck can be able to judge about sensors. Suppose the truck has sensed some object through the *bad-color-sensor*. Then it has had the opportunity to use the *color-sensor*. The reports of the *bad-color-sensor* are more erroneous than the reports of the *color-sensor.* It is possible that the truck senses a blue object as green. Therefore a truck must be able to make preference between facts from different sensors.
This field tells where an object is. A Truckworld node is an array of 16 positions. Thus *position* is a number between 0 and 15 in most cases. However the object might be in a container object (box, truck, arm). In this case *position* is a series of numbers. For example:

(position 10 3 6).

That is, the object is at position 6 of a container that is at position 3 of a container that is at position 10.

This field tells where an object is. A Truckworld node is an array of 16 positions. Thus *position* is a number between 0 and 15 in most cases. However the object might be in a container object (box, truck, arm). In this case *position* is a series of numbers. For example:

(position 10 3 6).

That is, the object is at position 6 of a container that is at position 3 of a container that is at position 10.

This field tells where an object is. A Truckworld node is an array of 16 positions. Thus *position* is a number between 0 and 15 in most cases. However the object might be in a container object (box, truck, arm). In this case *position* is a series of numbers. For example:

(position 10 3 6).

That is, the object is at position 6 of a container that is at position 3 of a container that is at position 10.

Finally, remember any object might be in a container object (box, truck, arm). In this case its *position* is a series of numbers. For example:

(position 10 3 6).

That is, the object is at position 6 of a container that is at position 3 of a container that is at position 10. You might create a truck with the obsession collecting things in boxes. The truck searching for the Box of Pandora.

This field tells where an object is. A Truckworld node is an array of 16 positions. Thus *position* is a number between 0 and 15 in most cases. However the object might be in a container object (box, truck, arm). In this case *position* is a series of numbers. For example:

(position 10 3 6).

That is, the object is at position 6 of a container that is at position 3 of a container that is at position 10.

Conclusion

We have implemented a pipe client for the Truckworld. Any expert system shell that has file IO operators can be used with the Truckworld the pipe client now. Using the pipe client does not require any other extensions or modifications to the expert system shell used. Depending on the shell used you might have to implement another data translator so that the shell easily parses the incoming data from the Truckworld. Nevertheless, using expert system shell with Truckworld for testing and comparison purposes will be easier.

Appendix A - Source Code for Pipe Client

```
(defun prompt-for (msg)
  (prog2 (format *query-io* "~S: " msg)
     (read *query-io* nil)))

 (defun answer-raw (stream output)
  (format stream "~S~%" output))

(defun answer-clips-parse-objects (stream time sensor objlist)
  (map    'nil #'(lambda (obj)
                         (progn
                          (format stream "position-info (time ~S) (sensor ~S)" time
              sensor) ; begin object
                                                  ; list attributes
                          (map 'nil #'(lambda (attr)
                                       (let ((type (car attr)) (value-list (cdr attr)))
                                        (progn
                                         (format stream "(~S" type)
                                         (map 'nil #'(lambda (value) (format stream " ~S"
                    value)) value-list)
                                         (format stream ")")
                                         )
                                        ))        ; print object attribute
                               obj)
                          (format stream "~%") ; end object
                          )
                         )
              objlist)
  )

(defun answer-clips (stream output)
  (let (
              (time (car output))
              (sense (cadr output))
              )
    (progn (format stream "time ~S~%" time)
                (if sense (let (
                                       (sensor  (cadar sense))
                                       (objlist (car (cddar sense)))
                                       )
                                 (answer-clips-parse-objects stream time sensor objlist)
                                 )                  ; end let
                  )                                          ; end if
                (format stream "end-of-answer~%")
                )                                            ; end progn
    )                                                        ; end let
  )
 (defun process-truck-pipes (&key
                                       (iname (prompt-for "Truck Command Pipe (in quotes)"))
                                       (oname (prompt-for "Truck Responce Pipe (in quotes)"))
                                       (answer #'answer-raw)
                                       )         ;end arguments

  (if (and (stringp iname) (stringp oname))
    (with-open-file (ifile iname :direction :input :if-does-not-exist :error)
                                   ; Instead of :error specifying nil causes hang up.
                                   ; No graceful exit possible? if file does not exist.
              (with-open-file (ofile oname :direction :output :if-exists :supersede)
                (do* ((input (read ifile nil) (read ifile nil))) ((null input)  nil)
                 (cond
                  ((listp input)
                   (format *terminal-io* "~%truckworld> ~S~%~%" input)
                   (funcall answer ofile (multiple-value-list (execute-truck-command                  input)))
                   (finish-output ofile))
                  ((stringp input)
                   (format *terminal-io* input))
                  (t
                   (format *terminal-io* "WARNING: strangeness in input file: ~S~%"
              input))
                  )                              ;end cond
                 )                               ;end do*
                ))                               ;end with-open-file's
                                                 ;else if
    (prog2 (format *terminal-io* "All inputs should be quoted!~%") t)
    )                                         ;end if
  )

 (defun make-truck-pipes  (&key
                            (iname (prompt-for "Truck Command Pipe (in quotes)"))
                            (oname (prompt-for "Truck Responce Pipe (in quotes)")))
                    ;end arguments
  (cond
    ( (not (stringp iname))
    (prog2
              (format *terminal-io* "Truck Command Pipe: Name was not quoted!~%") nil)
    )                                        ; end case
    ( (not (stringp oname))
    (prog2
              (format *terminal-io* "Truck Responce Pipe: Name was not quoted!~%")        nil)
    )                                        ; end case
    ( t
    (if (let ((err (excl:shell (concatenate 'string "mknod " iname " p"))))
                (= 0 err)
                )                                ;end zero test
              (if (let ((err (excl:shell (concatenate 'string "mknod " oname " p"))))
                   (= 0 err)
                   )                     ; end zero test
                  t
                                                 ;else
                (prog2 (format *terminal-io*
                                 "Cannot create responce pipe!~%") nil)
                )                                ; end if
                                                 ;else
          (prog2 (format *terminal-io*
                          "Cannot create command pipe!~%") nil)
         )                               ; end if
    )                                        ; end default case
    )                                         ; end cond
  )

(defun truck-pipes ()
  (let (
       (iname (prompt-for "Truck Command Pipe (in quotes)"))
              (oname (prompt-for "Truck Responce Pipe (in quotes)"))
              )
    (progn
         (make-truck-pipes :iname iname :oname oname)
         (process-truck-pipes :iname iname :oname oname)
```

```
      )                                                    ; end progn
    )                                                      ; end let
  )

(defun run-demo-functional-client (&rest keys &key (world "demo-world")        display)
  (apply #'start-functional-simulator
            :world world
            :truck-form '(make-truck :truck-id demo-truck)
           keys))
```

Appendix B - Least Source Code for a Truck Agent

```
 (deftemplate position-info
        (field node)
        (field time)
        (field sensor)
        (multifield position) ;BUG should be multi-field but rejected in 6.0?
        (field kind)
        (field sensor-id)
        (field color)
        (field bigness))

(defrule  mainBegin
        (initial-fact)        ; initialize the program
        =>
        (assert (pipes-will-open))
)

(defrule mainEnd ; end-up the program
        ?pe <- (program-exit)
        =>
        (retract ?pe)
        (assert (pipes-close)))

(deffunction get-pipe-names ()
        (format t "Enter command pipe name: ")
        (bind ?commandFile (readline))
        (format t "Enter sensor pipe name: ")
        (bind ?sensorFile (readline))
        (mv-append ?commandFile ?sensorFile)
)

(defrule pipesOpen
        ?pwo <- (pipes-will-open)
        =>
        (bind ?names (get-pipe-names))
        (bind ?commandFile (nth 1 ?names))
        (bind ?sensorFile (nth 2 ?names))
        (open ?commandFile commandStream "w")
        (open ?sensorFile sensorStream "r")
        (retract ?pwo)
        (assert (pipes-open)))

(defrule pipesClose
        ?fo <- (pipes-open)
        ?fc <- (pipes-close)
        =>
        (retract ?fo ?fc)
        (close))


(deffunction truck-command-validate  ($?command) TRUE)

(deffunction truck-listen (?sensorStream)
        (bind ?answerStr (lowcase (readline ?sensorStream)))
        (if (neq ?answerStr "end-of-answer") then
                ; (format t "        ...Received: %s%n" ?answerStr)
                (bind ?answerStr (str-cat "(" ?answerStr ")"))
                ; Surrounding paran. not needed till CLIPS v6.0
                (assert-string ?answerStr)
                (truck-listen ?sensorStream)
        else TRUE
        )
)

(deffunction truck-execute (?commandStream ?sensorStream $?command)
        (if (truck-command-validate ?command) then
                (bind ?commandStr (str-implode ?command))
                (format ?commandStream "(%s)%n" ?commandStr)
                (truck-listen ?sensorStream)
        else    FALSE
        )
)

(deffunction execute-truck-command ($?command)
        (truck-execute commandStream sensorStream ?command))
```

Appendix C - Sample Lisp Code

Here list of brief LISP code examples. This collection helped us a lot. If you are a new comer to LISP, try them. Instead of your reading a thousand page manual thoroughly they will guide you to the frequently used lisp constructs used in the implementation of the pipe client.

- Print on the terminal the value of *dummy* followed by a newline.
```
(format t "~S~%" *dummy*)
```
- Execute the UNIX shell command “ls” from within LISP.
```
(excl:shell "ls")
```
- This line returns “umur ozkul”.
```
(concatenate 'string "umur " "ozkul")
```
- Print the string “Umur Ozkul” vertically. Via *map* you enumerate the elements of collections.
```
(map    'list
        #'(lambda (x) (format t "--- ~S ----~%" x))
        "Umur Ozkul")
```
- Add two collections of numbers. The result is ‘(11 22 33).
```
(map 'list #'+ '(1 2 3) '(10 20 30) )
```
- Some functions return more than one value. Do not confuse; not a list of values! (floor 1.2) by default returns 1. However, its second return value is 0.2. Using *multiple-value-list* you collect all the return values in a list - ‘(1, 0.2). *(execute-truck-command…)* of the Truckworld is such a command. It has baffled us a while.
```
(multiple-value-list (floor 1.2))
```
- Make a pipe. Its name is fetched from the variable *sample-pipe-name*. *setq* is the assignment function.
```
(setq sample-pipe-name "dummy-pipe")
(execl:shell (concatenate 'string "mknod " sample-pipe-name " p") )
```
- Define an absolute file path name - /foo/umur/.
```
(make-pathname :name "umur" :directory '(:absolute "foo"))
```
- Print hello on the file “dummy2”.
```
(with-open-file
        (ofile "dummy2" :direction :output :if-exists :supersede)
        (format ofile "Hello~%"))
```
- Read a word from file “dummy1”.
```
(with-open-file
        (ifile "dummy1" :direction :input)

                (format t "~S~%"  (read ifile nil))
)
```
- Read and display all the contents of the file “dummy1”.
```
(with-open-file
        (ifile "dummy1" :direction :input)
                (do* ((input (read ifile nil) (read ifile nil))) ((null I
        nput)  nil)
                        (format t "~S~%"  input)
                )
)
```
- Copy from “dummy1” to “dummy2”.
```
(with-open-file (ifile "dummy1" :direction :input)
        (with-open-file (ofile "dummy2" :direction :output :if-exists
                :supersede)
                (do* ((input (read ifile nil) (read ifile nil))) ((null
                input)  nil)
                        (format ofile "~S~%"  input)
                )
        )
)
```

References

[1] Dat Nguyen, Steve Hanks, and Chris Thomas; The Truckworld Manual; Technical Report, University of Washington, Department of Computer Science and Engineering, 1993.

[2] Hanks, Steve, et al.; A Beginner's Guide to the Truckworld Simulator; Department of Computer Science and Engineering; University of Washington; Technical Report UW-CSE-TR 93-06-09 June 25, 1993

[3] Hanks, Steve, et al.; Benchmarks, Testbeds, Controlled Experimentation, and the Design of Agent Architectures; Department of Computer Science and Engineering, University of Washington; Technical Report UW-CSE-TR 93-06-05 June 17, 1993

[4] Joseph C. Giarratano, CLIPS Users' Guide VOLUME 1 Rules, CLIPS Version 5.1, September 10th 1991

[5] Stylianou, A. C., et al.; An Empirical Model for the Evaluation and Selection of Expert System Shells; The Belk College of Business Administration, University of North Carolina

[6] Romkey, J.; Networking with BSD-Style Sockets; UNIX World, July 1989