Vtcl 2 Prolog Project

Building Prolog Applications

Antonio Pertíñez Pertíñez
&
Gustavo Fernández Fernández

Authors

Gustavo Fernández Fernández

Engineering Computer Student of E.T.S.I.I (Escuela Técnica Superior Ingeniería Informática)

Address: Street: Artemisa
Building: Almeria
Number: 3 Letter: B
E-Mail: gustaf@fedro.ugr.es
gff@ruc.dk
gustavofdez@hotmail.com
Telephone: +034 958 16 08 39

Antonio Pertíñez Pertíñez

Engineering Computer Student of E.T.S.I.I (Escuela Técnica Superior Ingeniería Informática)

Address: Street: Carril del Picón
Number: 26, 1D
E-Mail: swot@fedro.ugr.es
app@ruc.dk
bobbin@hotmail.com
Telephone: +034 958 27 30 32

PROJECT SUPERVISOR:

John Gallagher

Professor of Computer Science
Roskilde University
Computer Science, Building 42.1
P.O. Box 260
DK-4000 Roskilde, Denmark
Phone: +45 4674 2196
Fax: +45 4674 3072
jpg@ruc.dk

INDEX

FIGURE INDEX

FOREWORD AND ACKNOWLEDGMENTS

Every road has an end… and we are arriving to our end as Computer Engineering students. So we decided to finish our degree doing something special, and here we are in Denmark, after five months of working in our project.

Everything was new here; life style and language were pretty different to ours. Also the people and the way they live.

We learnt here not only computer theory and practice, we also learnt how to live in a new country.

We were living in a Danish student Residence, so we met many different students who are in many degrees, and so, we saw how they work. We also had to get used to study in English. Nowadays, Computer Science speaks English, so it was so good for us to develop our project into a completely English environment.

Our knowledge not only grew inside the university, but also out of these buildings. Five months are not enough to appreciate the beauty of this wonderful country. Nevertheless, we could take something of that to bring with us to Spain.

At the beginning, it was difficult to get used to the many new things that we encountered here, but little by little, we made Roskilde our home, and it will be so hard to say goodbye.

We also want to thank the people who helped us in the developing of the project.

Juan Carlos Cubero, Manuel Hermenegildo, Daniel Cabeza and our supervisor John Gallagher, thank you.

INTRODUCTION

Our project started long time before coming to Denmark. We knew it would be difficult to work in a different language, so we wanted to prepare ourselves before coming. We tried to be assigned to a supervisor who indicated us in which areas we were going to work.

First of all, we contacted **Henning Christiansen**, (Associate professor, Ph.D. (lektor) in Computer Science at Roskilde University) who is the Erasmus Coordinator in Datalogi. He told us the different subjects in which they work and the various options for us.

We also contacted **Niels Christian Juul** (Associate professor, Ph.D., Head of the Computer Science section) but he was starting to manage other administrative duties so he couldn't supervise us. But actually, he put us in contact with our present supervisor, **John Gallagher** (Professor of Computer Science).

John Gallagher specializes in Logic Programming and Program Analysis and Transformations. When we contacted him, he proposed us to develop our project in these areas, which seemed very interesting.

One of the projects, which is being developed in those areas is **ASAP Project: Advanced Specialization and Analysis for Pervasive Computing**. We will give a brief description about the ASAP project later.

Four institutions are developing ASAP:

- *Technical University of Madrid, UPM (Spain)*
- *University of Bristol (UK)*
- *University of Roskilde (Denmark)*
- *University of Southampton (UK)*

The coordinator of the project is the **CLIP Research Group** (UPM, Spain). The Computational Logic, Implementation and Parallelism (CLIP) Laboratory has been active at UPM's School of Computer Science since 1990. The CLIP´s lines of research are:

- *Distributed execution, internet, WWW:* Libraries for programming distributed, multiple agents. WWW/Internet programming. Flexible layout languages.

- *Advanced program development tools:* Combined static/dynamic debugging. Program validation/certification. Automatic program documentation. Sequential and parallel (C)LP execution visualization.

- *Global program analysis:* Abstract interpretation frameworks and domains. Analysis of real-life languages (e.g., ISO-Prolog), CLP, languages with delay. Modular and incremental analysis. Inference of complex program properties (non-failure, determinacy, bounds on cost, rich moded types, etc.).

- *Program transformation/optimization:* Program optimization via multiple abstract specialization. Integration with partial evaluation and low-level optimization. Target abstract machines.

- *Programming language design:* (Concurrent) LP/CLP languages. Module systems, functions, higher order, and object-oriented extensions.

- *Parallelism:* Automatic program parallelization (including task granularity control). Optimization of concurrent and distributed programs. Parallel models and bounds. Parallel / distributed abstract machines.

The group is strongly committed to developing solutions which constitute at the same time quality research and clearly applicable technology. To this end, most of the group's results are tested in and integrated into the **Ciao program development environment**, which is used as the group's main implementation platform. Ciao is the result of many collaborations, to which the group is always open. It has proven useful, both for us and for other groups, not only for application development but also as a platform for the implementation of LP/CLP extensions, optimizations, and program processing tools: the open design of Ciao allows the integration of such tools into a real-life environment (and dealing with the related problems) with limited effort.

The Clip description above is taken from the Clip web page. For more details see *Reference [18].*

John spoke about this to us and he gave us several alternatives to work in. All of these alternatives are based in Logic Programming and the most common logic language: **Prolog**.

These alternatives were:

- *Static Analysis*. It is about finding out what a program is doing without executing it. Advanced debugging tools are based on static analysis.
- *Graphical Interfaces for Static Analysis Tools*: Static analysers need to be supported by good GUIs, possibly Web-bases.
- *XML-based program development*: Program generation based on XML definition is an increasingly common method of software development.
- *Graphical Simulation of Concurrent Systems.*

Concretely John offered us to work in a program, which let us develop an interface in a visual language using Prolog commands, and transform it later into a Prolog program. The visual language selected was Tcl/Tk, and its visual extension Vtcl.

Before describing the characteristics of our project we will introduce all the languages and tools involved. These introductions are very important to understand our project.

PROLOG INTRODUCTION

The purpose of this report is not to give a completely description of the Prolog language, but it is important to understand the project, since Prolog is the main language used in it.

The next Prolog introduction is taken from one of the Prolog tutorials that we have used during the development of the project. *Reference [13].*

Prolog, which stands for PROgramming in LOGic, is the most widely available language in the logic-programming paradigm. Logic and therefore Prolog is based the mathematical notions of relations and logical inference. Prolog is a declarative language meaning that rather than describing how to compute a solution, a program consists of a database of facts and logical relationships (rules), which describe the relationships, which hold for the given application. Rather then running a program to obtain a solution, the user asks a question. When asked a question, the run time system searches through the database of facts and rules to determine (by logical deduction) the answer.

Among the features of Prolog are `logical variables' meaning that they behave like mathematical variables, a powerful pattern-matching facility (unification), a backtracking strategy to search for proofs, uniform data structures, and input and output are interchangeable.

Often there will be more than one way to deduce the answer or there will be more than one solution, in such cases the run time system may be asked find other solutions. Backtracking to generate alternative solutions. Prolog is a weakly typed language with dynamic type checking and static scope rules.

Prolog is used in artificial intelligence applications such as natural language interfaces, automated reasoning systems and expert systems. Expert systems usually consist of a database of facts and rules and an inference engine, the run time system of Prolog provides much of the services of an inference engine.

The Structure of Prolog Programs

A **Prolog program** consists of a database of facts and rules. There is no structure imposed on a Prolog program, there is no main procedure, and there is no nesting of definitions. All facts and rules are global in scope and the scope of a variable is the fact or rule in which it appears. The readability of a Prolog program is left up to the programmer.

A Prolog program is executed by asking a question. The question is called a query. Facts, rules, and queries are called *clauses*.

Syntax

Facts

A **fact** is just what it appears to be --- a fact. A fact in everyday language is often a proposition like ``It is sunny.'' or ``It is summer.'' In Prolog such facts could be represented as follows:

```
'It is sunny'.
'It is summer'.
```

Queries

A **query** in Prolog is the action of asking the program about information contained within its database. Thus, queries usually occur in the interactive mode. After a program is loaded, you will receive the query prompt,

```
?-
```

At which time you can ask the run time system about information in the database. Using the simple database above, you can ask the program a question such as

```
?- 'It is sunny'.
```

And it will respond with the answer

```
Yes
?-
```

A `yes` means that the information in the database is consistent with the subject of the query. Another way to express this is that the program is capable of proving the query true with the available information in the database. If a fact is not deducible from the database the system replies with a no, which indicates that based on the information available (the closed world assumption) the fact is not deducible.
If the database does not contain sufficient information to answer a query, then it answers the query with a no.

```
?- 'It is cold'.
no
?-
```

Rules

Rules extend the capabilities of a logic program. They are what give Prolog the ability to pursue its decision-making process. The following program contains two rules for temperature. The first rule is read as follows: ``it is hot if it is summer and it is sunny.'' The second rule is read as follows: ``it is cold if it is winter and it is snowing.''

```
'It is sunny'.
'It is summer'.
'It is hot' :- 'It is summer', 'It is sunny'.
'It is cold' :- 'It is winter', 'It is snowing'.
```

The query,

```
?- 'It is hot'.
Yes
?-
```

Is answered in the affirmative since both 'It is summer' and 'It is sunny' are in the data base while a query ``?- 'It is cold.' '' Will produce a negative response.
The previous program is an example of propositional logic. Facts and rules may be parameterised to produce programs in predicate logic. The parameters may be variables, atoms, numbers, or terms. Parameterisation permits the definition of more complex relationships. The following program contains a number of predicates that describe a family's genealogical relationships.

```
female(amy).
female(johnette).

male(anthony).
male(bruce).
male(ogden).

parentof(amy,johnette).
parentof(amy,anthony).
parentof(amy,bruce).
parentof(ogden,johnette).
parentof(ogden,anthony).
parentof(ogden,bruce).
```

The above program contains the three simple predicates: `female; male;` and `parentof.` They are parameterised with what are called `atoms.' There are other family relationships, which could also be written as facts, but this is a tedious process. Assuming traditional marriage and child-bearing practices, we could write a few rules which would relieve the tedium of identifying and listing all the possible family relations. For example, say you wanted to know if `johnette` had any siblings, the first question you must ask is ``what does it mean to be a sibling?'' To be someone's sibling you must have the same parent. This last sentence can be written in Prolog as

```
siblingof(X,Y) :-
parentof(Z,X),
parentof(Z,Y).
```

A translation of the above Prolog rule into English would be ``X is the sibling of Y provided that Z is a parent of X, and Z is a parent of Y.'' X, Y, and Z are variables. This rule however, also defines a child to be its own sibling. To correct this we must add that X and Y are not the same. The corrected version is:

```
siblingof(X,Y) :-
parentof(Z,X),
parentof(Z,Y),
X  Y.
```

The relation `brotherof` is similar but adds the condition that X must be a male.

```
brotherof(X,Y) :-
parentof(Z,X),
male(X),
parentof(Z,Y),
X  Y.
```

From these examples we see how to construct facts, rules and queries and that strings are enclosed in single quotes, variables begin with a capital letter, constants are either enclosed in single quotes or begin with a small letter.

Once we have explained the Prolog basics, we will give a small example from another reference. *Reference [17].*

A Small Example

Let us consider the following description of a ``system'';

```
Ann likes every toy she plays with. A doll is a toy. A
train is a toy. Ann plays with trains. John likes
everything Ann likes.
```

To express this in Prolog we must:

1. Identify the entities, or actual things, mentioned in the description
2. Identify the types of properties that things can have, as well as the relations that can hold between these things
3. figure out which properties/relations hold for which entities.

There is really no ``unique'' way of doing this; we must decide the best way to structure our data (based on what we want to do with it).

We will choose the following:

Things: Ann, Sue, doll, train
Properties: ``... is a toy''
Relations: ``... likes ...'', ``... plays with ...''

Constructing our knowledge base then consists of writing down which properties and relationships hold for which things.

We write:

```
likes(ann,X) :- toy(X), plays(ann,X).
toy(doll).
toy(train).
plays(ann,train).
likes(john,Y) :- likes(ann,Y).
```

What it means

There are three important logical symbols; they are:

```
Symbol |  Meaning
=======+=========
       |
  :-   |    if
       |
  ,    |    and
```

```
        |
;       |     or
```

(Only the first two are used in our program).
As regards the other symbols in the program:

X and Y are *variables*
ann, john, doll and train are *constants*
likes, toy and plays are *predicate* symbols

A **variable** represents some unspecified element of the system. A **constant** represents a particular, known, member of the system. A **predicate** represents some relation or property in the system.

Note that:

- Variables always start with an upper-case letter or an underscore
- Predicates and constants always start with a lower-case letter or digit

Each line in a Prolog program is called a **clause**.

There are two types of clauses - facts and rules.

- **Rules** are clauses which contain the ``:-'' symbol.
- **Facts** are clauses which don't.

Each *fact* consists of just one predicate.

Each *rule* consists of a predicate, followed by a ``:-'' symbol, followed by a list of predicates separated by ``,'' or ``;''.

Every clause is terminated by a ``.'' (full-stop).

In a rule, the predicate before the ``:-'' is called the **head** of the rule. The predicates coming after the ``:-'' are called the **body**

For example:

```
likes(ann,X)   :-  toy(X), plays(ann,X).

<---Head--->       <-------Body------->
```

We ``define a predicate'' by writing down a number of clauses which have that predicate at their *head*.

(NB: The *order* in which we write these down *is* important)

Any predicates mentioned in the *body* must either:

- be defined somewhere else in the program, or
- be one of Prolog's ``built-in'' predicates.

Defining a predicate in Prolog corresponds roughly to defining a procedure in Pascal.

Predicates occurring in the body of a clause correspond roughly to procedure calls in Pascal.

Note also that:

- Constants and variables will never appear ``on their own'' in a clause. They can only appear as the ``arguments'' to some predicate.
- Predicates will (almost) never appear as arguments to another predicate

What it says

So, after all that, what does our little program say?

Having all the relations expressed as a predicate followed by arguments is not particularly intuitive, so with some suitable swapping-around we get:

- For any X, (ann likes X) if (X is-a-toy) and (ann plays-with X).
- (doll is-a-toy).
- (train is-a-toy).
- (ann plays-with train).
- For any Y, (john likes Y) if (ann likes Y).

Running the program

So how do we run it?

We run it by giving Prolog a query to prove.

A query has exactly the same format as a clause-body: one or more predicates, separated by ``,'' or ``;'', terminated by a full-stop.

Thus, we might enter in the following as a query:

```
likes(john,Z).
```

- Logically, this can be interpreted as ``is there a Z such that john likes Z?''
- From a relational point of view, we can read it as: ``List all those Z's that john likes''
- In general terms we call the query our ``goal'', and say that Prolog is being asked to (find ways to) ``satisfy'' the goal.

This process is also known as **inferencing**: Prolog has to infer the solution to the query from the knowledge base.

Note that solving a query results in either:
- failure, in which case ``no'' is printed out, or
- success, in which case all sets of values for the variables in the goal (which cause it to be satisfied) are printed out.

How it works

So how does Prolog get an answer?

We have to solve `likes(john,Y)`, so we must examine all the clauses which start with the predicate `likes`. The first one is of no use at this point, since it only tells us what `ann` likes.

The second rule for `likes` tells us that in order to find something that `john` likes, we need only to find something which `ann` likes. So now we have a new goal to solve – `likes(ann,Z)`.

To solve this we again examine all the rules for `likes`. This time the first rule matches (and the second doesn't), and so we are told that in order to find something which `ann` likes, we must find something which is a `toy`, and which `ann` plays with. So first of all we try to find a `toy`. To do this we examine the caluses with `toy` at their head. There are two possibilities here: a `toy` is either a `doll` or `train`.

We now take these two toys, and test to see which one `ann` plays with; that is, we generate two new sub-goals to solve:
`plays(ann,doll)` and `plays(ann,train)`.

In general, to solve *these*, we must look at the clauses for `plays`. There is only one: since it is for `train`, we conclude with the answer:

```
Z = train.
```

To solve a goal consisting of just one predicate `p` say, Prolog:

- Looks at each caluse with `p` at its head in turn
- For each of these it matches up the constants and variables in the goal with those in the head, and performs the appropriate replacements in the body.
- This new version of the body is the new goal.

CIAO PROLOG INTRODUCTION

As we introduced before, we will develop our project under a Prolog environment. We will manage as well Tcl language to generate interfaces, but it will be oriented to work together with Prolog programs.

So the programming environment we will work it is called **Ciao Prolog**, a public domain, next generation multiparadigm programming environment with the next set of features:

- Ciao offers a complete Prolog system, supporting *ISO-Prolog*, but its novel modular design allows both *restricting* and *extending* the language. As a result, it allows working with *fully declarative subsets* of Prolog and also to *extend* these subsets (or ISO-Prolog) both syntactically and semantically. Most importantly, these restrictions and extensions can be activated separately on each program module so that several extensions can coexist in the same application for different modules.

- Ciao also supports (through such extensions) programming with functions, higher-order (with predicate abstractions), constraints, and objects, as well as feature terms (records), persistence, several control rules (breadth-first search, iterative deepening,...), concurrency (threads/engines), a good base for distributed execution (agents), and parallel execution. Libraries also support WWW programming, sockets, external interfaces (C, Java, TclTk, relational databases, etc.), etc.

- Ciao offers support for *programming in the large* with a robust module/object system, module-based separate/incremental compilation (automatically --no need for makefiles), an assertion language for declaring (*optional*) program properties (including types and modes, but also determinacy, non-failure, cost, etc.), automatic static inference and static/dynamic checking of such assertions, etc.

- Ciao also offers support for *programming in the small* producing small executables (including only those builtins used by the program) and support for writing scripts in Prolog.

- The Ciao programming environment includes a classical top-level and a rich emacs interface with an embeddable source-level debugger and a number of execution visualization tools.

- The Ciao compiler (which can be run outside the top level shell) generates several forms of architecture-independent and stand-alone executables, which run with speed, efficiency and executable size which are very competive with other commercial and academic Prolog/CLP systems. Library modules can be compiled into compact bytecode or C source files, and linked statically, dynamically, or autoloaded.

- The novel modular design of Ciao enables, in addition to modular program development, effective global program analysis and static debugging and optimization via source to source program transformation. These tasks are performed by the **Ciao preprocessor** (*ciaopp*, distributed separately).

- The Ciao programming environment also includes *lpdoc*, an automatic documentation generator for LP/CLP programs. It processes Prolog files adorned with (Ciao) assertions and machine-readable comments and generates manuals in many formats including postscript, pdf, texinfo, info, HTML, man, etc. , as well as on-line help, ascii README files, entries for indices of manuals (info, WWW, ...), and maintains WWW distribution sites.

Ciao is distributed under the **GNU General Public License.**

We will briefly introduce a few of the extensions that Ciao brings to the Prolog language.

Predicates and their components

In Prolog, procedures are called *predicates* and predicate calls *literals*. They all have the classical syntax of procedures (and of logic predications and of mathematical functions). Predicates are identified in this manual by a keyword 'PREDICATE' at the right margin of the place where they are documented.

Prolog instructions are expressions made up of control constructs and literals, and are called *goals*. Literals are also (atomic) goals.

A predicate definition is a sequence of clauses. A clause has the form "`H :- B.`" (ending in '.'), where `H` is syntactically the same as a literal and is called the clause *head*, and `B` is a goal and is called the clause *body*. A clause with no body is written "`H.`" and is called a *fact*. Clauses with body are also called *rules*. A Prolog program is a sequence of predicate definitions.

Characters and character strings

We adopt the following convention for delineating character strings in the text of this manual: when a string is being used as a Prolog atom it is written thus: `user` or `'user'`; but in all other circumstances double quotes are used (as in "hello").

When referring to keyboard characters, printing characters are written thus: a, while control characters are written like this: `^A`. Thus `^C` is the character you get by holding down the CTL key while you type c. Finally, the special control characters carriage-return, line-feed and space are often abbreviated to RET, LFD and SPC respectively.

Predicate specs

Predicates in Prolog are distinguished by their name *and* their arity. We will call `name/arity` a *predicate spec*. The notation `name/arity` is therefore used when it is necessary to refer to a predicate unambiguously. For example, `concatenate/3` specifies the predicate which is named "concatenate" and which takes 3 arguments.
(Note that different predicates may have the same name and different `arity`. Conversely, of course, they may have the same `arity` and different name.)

Modes

When documenting a predicate, we will often describe its usage with a mode spec which has the form `name(Arg1, ..., ArgN)` where each `Arg` may be preceded by a *mode*. A mode is a functor which is wrapped around an argument (or prepended if defined as an operator). Such a mode allows documenting in a compact way the instantiation state on call and exit of the argument to which it is applied.

These modes are specified with the symbols -,+ or nothing.

- + means that the argument is an input argument.
- - means that the argument is an output argument.

- If the argument doesn't have neither + or – it means that the argument could be an input or an output .

For example `foo(+Input,-Output,Nothing)` denotes a predicate that has:

- One argument for input (Input).
- One argument for output (output).
- And an argument that can be input or output.

This introduction about Ciao Prolog has been taken from the Ciao System web page. *Reference [19].*

TCL/TK INTRODUCTION

Our project will create a "bridge" between a graphical language and a Prolog system. The main purpose of our project is to build a nice interface using a Rapid **Application Development** (RAD) programming tool and then, convert it to a Prolog module. For many reasons we chose Tcl/Tk as the graphical language, but the most important factor is that it is very well connected to Ciao Prolog.

"Tcl, Tool Command Language, is an interpreted language with programming features, available across platforms running Unix, Windows and the Apple Macintosh operating system. Tk, the associated toolkit is an easy and efficient way of developing window based applications. Application tasks are split into modules and any new application specific task is written and compiled as C or C++ program and

exported as a new Tcl command. Then a Tcl script, a series of existing and new Tcl commands, is composed to make the overall application. The scripting language, much like any shell language, has the ability to access and execute any other programs. Therefore several Tcl based applications could be made to work together to create or extend into a new application.

Tcl consists of few syntax rules and a (still growing) set of core commands. Tk provides a higher-level application-programming interface for developing interactive widgets based applications, particularly for those who wish to concentrate on the functionality of their application and have no need to gain indepth-programming expertise in the underlying window system and/or verbose toolkits such as OSF/Motif. Tcl/Tk is free, available now on Apple Macintosh and Windows and has a wide user base with a rich and growing mass of useful contributed software. The wider availability, usage and ease of teaching and learning of Tcl/Tk makes it the most appropriate tool for teaching the principles of Graphical User Interface design and development. " Reference [26]

As well as we did with Prolog; we will give a brief description of Tcl syntax and some simple examples.

Following example is taken from *Reference [20].*

For a scripting language, Tcl has a simple syntax.

```
cmd arg arg arg
```

A Tcl command is formed by words separated by white space. The first word is the name of the command, and the remaining words are arguments to the command.

```
$foo
```

The dollar sign ($) substitutes the value of a variable. In this example, the variable name is foo.

```
[clock seconds]
```

Square brackets execute a nested command. For example, if you want to pass the result of one command as the argument to another, you use this syntax. In this example, the nested command is clock seconds, which gives the current time in seconds.

```
"some stuff"
```

Double quotation marks group words as a single argument to a command. Dollar signs and square brackets are interpreted inside double quotation marks.

```
{some stuff}
```

Curly braces also group words into a single argument. In this case, however, elements within the braces are not interpreted.

```
\
```

The backslash (\) is used to quote special characters. For example, \n generates a newline. The backslash also is used to "turn off" the special meanings of the dollar sign, quotation marks, square brackets, and curly braces.

A Little Example

Below is a Tcl command that prints the current time. It uses three Tcl commands: `set`, `clock`, and `puts`. The `set` command assigns the variable. The `clock` command manipulates time values. The `puts` command prints the values.

```
set seconds [clock seconds]
puts "The time is [clock format $seconds]"
```

Note that you do not use $ when assigning to a variable. Only when you want the value do you use $. The `seconds` variable isn't needed in the previous example. You could print the current time with one command:

```
puts "The time is [clock format [clock seconds]]"
```

Grouping and Substitution

The Tcl syntax is used to guide the Tcl parser through three steps: argument grouping, result substitution, and command dispatch.

a. **Argument grouping.** Tcl needs to determine how to organize the arguments to the commands. In the simplest case, white space separates arguments. As stated earlier, the quotation marks and braces syntax is used to group multiple words into one argument. In the previous example, double quotation marks are used to group a single argument to the puts command.
b. **Result substitution.** After the arguments are grouped, Tcl performs string substitutions. Put simply, it replaces $foo with the value of the variable foo, and it replaces bracketed commands with their result. That substitutions are done *after* grouping is crucial. This sequence ensures that unusual values do not complicate the structure of commands.
c. **Command dispatch.** After substitution, Tcl uses the command name as a key into a dispatch table. It calls the C procedure identified in the table, and the C procedure implements the command. You also can write command procedures in Tcl. There are simple conventions about argument passing and handling errors.

Another Example

Here is another example:

```
set i 0
while {$i < 10} {
    puts "$i squared = [expr {$i*$i}]"
    incr i
}
```

Here, curly braces are used to group arguments without doing any substitutions. The Tcl parser knows nothing special about the while command. It treats it like any other command. It is the implementation of the while command knows that the first argument is an expression, and the second argument is more Tcl commands. The braces group two arguments: the boolean expression that controls the loop and the commands in the loop body.

We also see two math expressions: the boolean comparison and multiplication. The while command automatically evaluates its first argument as an expression. In other cases you must explicitly use the expr command to perform math evaluation.

Command Dispatch

Lastly, Tcl calls something else to do the hard work. We've seen that Tcl uses expr to perform math functions, puts to handle output functions, and set to assign variables. These Tcl commands are implemented by a C procedure that has registered itself with Tcl. The C command procedures take the string arguments from the Tcl command and return a new string as their result. It is very easy to write C command procedures. They can do everything from accessing databases to creating graphical user interfaces. Tcl, the language, doesn't really know what the commands do. It just groups arguments, substitutes results, and dispatches commands.

One Last Example

Here is the factorial procedure:

```
proc fac {x} {
    if {$x < 0} {
            error "Invalid argument $x: must be a
            positive integer"
    } elseif {$x <= 1} {
        return 1
                } else {
        return [expr {$x * [fac [expr {$x-1}]]}]
    }
}
```

For more details Appendix A, where we have included a Tcl manual.

VTCL INTRODUCTION

There are a few RAD tools available for developing GUI's in Tcl but Vtcl has a feature set and RAD philosophy that is straight forward, hence forth Vtcl for a RAD environment. Vtcl allows the developer to test and prototype on the go. Developing applications with Vtcl is almost as effortless. Providing easy and quick application development via an intuitive interface is a strength for any user. Vtcl provides a powerful vehicle to solve problems with the power and flexibility of Tcl/Tk…

So we will use to program our interfaces the extension of Tcl/Tk, Vtcl.

Visual Tcl is a freely available, high-quality application development environment for UNIX, Windows, Macintosh and AS400 platforms. Visual Tcl is written entirely in Tcl/Tk and generates pure Tcl/Tk code. This makes porting your Visual Tcl applications either unnecessary or trivial. The GNU general public licence covers visual Tcl. We will use the UNIX version of Vtcl.

Some features interesting of Vtcl are these:

- 100% pure Tcl/Tk. No external libraries required.

- Extensible widget and geometry manager support.
- Create compound widgets and widget libraries.
- GUI interface for most aspects of Tcl/Tk development.
- Support for user images and fonts in your project.
- Imports pre-existing Tcl/Tk code.
- Built-in support for widget toolkits including: [incr Widgets], BLT, TkTable
- Visual Tcl features new ready-to-use widgets: combo box, multicolumn list box, progress bar
- Predefined compounds available including scrolled text, scrolled list box, scrolled canvas, horizontal and vertical splitters
- Exports Tclets which run in Netscape/MSIE.
- Support for freewrap. Generate binaries for Windows or Linux.

Reference [34].

As well as we did with Tcl, we have included a Vtcl Manual in the Appendix B with some examples.

CIAO PROLOG TCL/TK INTERFACE

Ciao System provides a library package that is a bi-directional interface to the Tcl language. This library has been very useful for the development of the project. For this reason we want to include the description of the library. **This library is completely implemented in Prolog.**

The following description is taken from the Ciao Manual, *Reference [19].*

TCL/TK Library

The interaction between both languages is implemented as an interface between two processes, a Tcl/Tk process and a Prolog process. The approach allows programmers to program both in Tcl/Tk and Prolog.

Prolog - Tcl/Tk interface structure

The interface is made up of two parts: a Prolog part and a Tcl/Tk part. The Prolog part encodes the requests from a Prolog program and sends them to the Tcl/Tk part via a socket. The Tcl/Tk part receives from this socket and performs the actions included implied by the requests.

Prolog side of the Prolog - Tcl/Tk interface

The Prolog side receives the actions to perform in the Tcl/Tk side from the user program and sends them to the Tcl/Tk side through the socket connection. When the action is finished in the Tcl/Tk side, the result is returned to the user program, or the action fails if any problem occurs.

Tcl/Tk side of the Prolog - Tcl/Tk interface

The Tcl/Tk side waits for requests from the Prolog side, executes the Tcl/Tk code sent from the Prolog side. At the same time, the Tcl/Tk side handles the events and exceptions raised in the Tcl/Tk side, possibly passing on control to the Prolog side.

Low-level interface library to TCL/TK

The tcltk_low_level library defines the low level interface used by the tcltk library. Essentially it includes all the code related directly to the handling of sockets and processes. This library should normally not be used directly by user programs, which use tcltk instead. On the other hand in some cases it may be useful to understand how this library works in order to understand possible problems in programs that use the tcltk library.

Reference [19].

PROBLEM DESCRIPTION

MAIN PROBLEM

Up to now, there is no Visual application builder that generates programs in Prolog code. Nevertheless, Ciao System provides a library package that is a bi-directional interface to the Tcl language, but unfortunately developing a application using this package directly is quite complicated since the library implements several low level predicates.

The objective of this library is that the programmer can call Prolog predicates from a visual application. For this reason it could be very useful to have a tool that takes an existing application (implemented in Tcl language) and generates

automatically the same application but in Prolog code. So our main problem and the purpose of the project is to provide the tool.

This tool can be based on the existing low level libraries in Ciao Prolog.

What should this tool do?

In general terms it has to take a Tcl file (generated by Vtcl) and to generate a Prolog module that provides the same functionality that the original Tcl file, using some of the facilities of the Ciao Prolog Tcl/Tk interface.

To illustrate this problem we can describe the general use of the final project. Suppose we have several Prolog predicates implemented in different modules, and we want to make a nice visual interface for them. But to do it in Prolog is not possible so we have to use a visual language for this purpose.

From the point of view of the user, a general example of the problem can be the following one:

Suppose we have three Prolog modules, with different utilities and we want to build a graphic interface for them in Vtcl. From the point of view of the user, also we would want that the code generated by Vtcl was translated to Prolog, so that we can utilize the application as if it was a predicate itself.

The first step for the user is to build his application where he calls the predicates that he wants to utilize (in the figure, he calls the predicates from the application buttons). Once it finishes the application, it is necessary to convert the code generated by the visual development application (Vtcl) to a Prolog module in order to allow the user to compile and execute the application, as it was a normal Prolog predicate.

For this we should provide to the user the corresponding application (the tool in the figure) that makes a program generated from Vtcl to a set of predicates and facts in Prolog. Besides the communication between Ciao Prolog and Vtcl should be transparent, so we have to add to our problem the corresponding procedures that negotiate said communication.

We will call this tool **TRANSLATOR** because it translates *Tcl code* into *Prolog code*

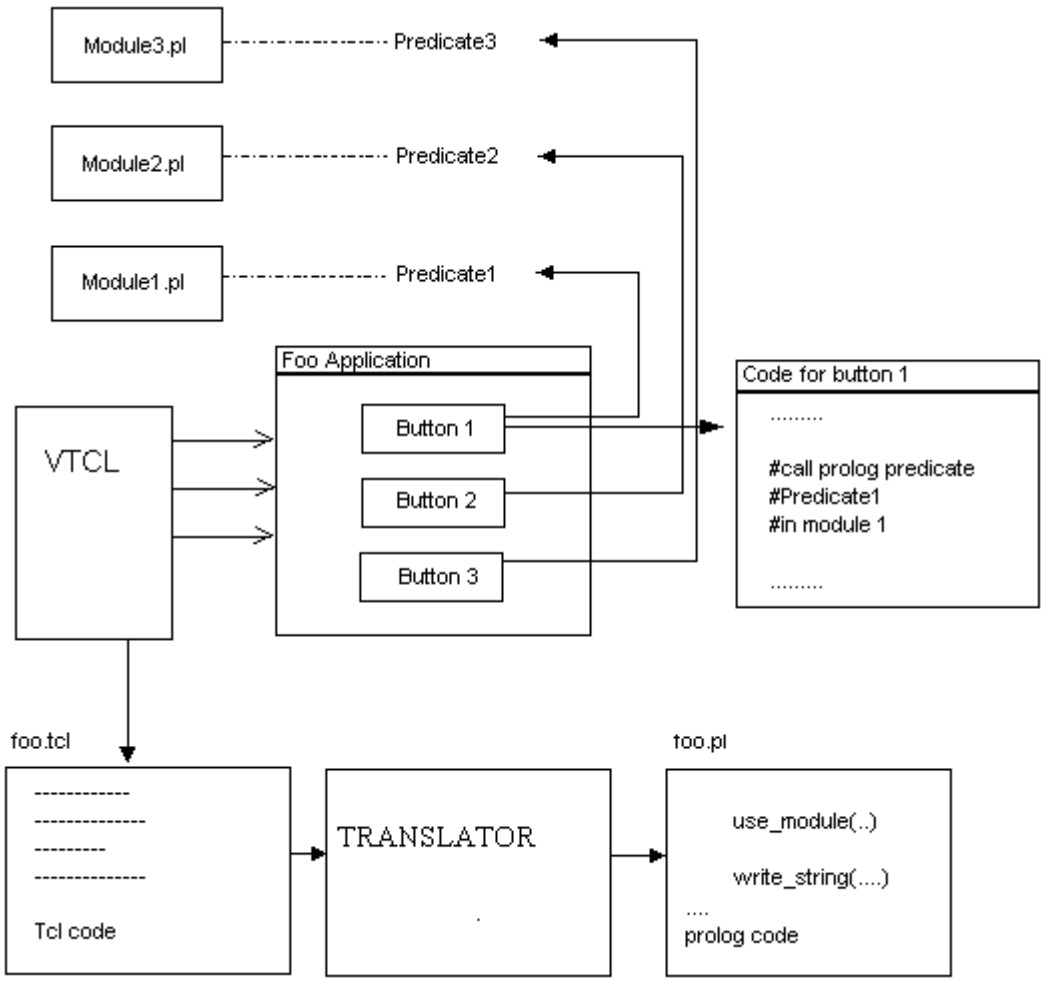

Figure 1. General Problem Description

PARTS OF THE PROJECT

The main parts of the project are three:

- Familiarization with the problem.
- To design and program, the translator, that generates the Prolog code from the graphic application written in Vtcl.

- To build the corresponding Tcl libraries to permit the user to utilize Prolog code when he develops his application in Vtcl.

Familiarization with the problem

Before beginning to give a solution to any problem, we must familiarize ourselves with the problem domain. This includes, studying the system where the solution will be developed; Ciao Prolog, as well as all the tools and languages, which should be used.

First of all, we should learn how Ciao Prolog and its main tools work. This includes reviewing and developing our knowledge about the programming language Prolog.

Moreover, we should familiarize ourselves with the other languages involved, which we didn't know until now. Therefore, we will study Tcl, its graphic extension Tk, and the application developed environment Vtcl.

Translator description

The main objective of the translator is to take an application developed in Vtcl and gets a similar program in Prolog code, which carries out the same functions.

We can differentiate two main parts into this problem:

- Code Translation. How we will translate the Tcl code into Prolog code.
- Program Structure. How we will construct the structure of the Prolog program.

Should the Translator work only with files generated by Vtcl?

Vtcl is just a visual Tcl extension so; it generates 100% pure Tcl/Tk code. This means that a Tcl/Tk programmer could use the translator for Tcl/Tk files completely written by hand, as well as with the files generated by Vtcl.

From this moment we will use the notation *tcl file* for any Tcl code file, generated by Vtcl or by Tcl/Tk.

Tcl libraries

The objective of the libraries is to provide to the user a way to utilize Prolog code from the application in Vtcl.

These libraries should be implemented from the existing libraries in Ciao Prolog for the Tcl/Tk interface. Moreover, they have to provide a natural way of use, without going into details with the communication with Prolog.

RESOLUTION OF THE PROBLEM

FAMILIARIZATION WITH THE PROBLEM

Ciao Prolog

Our project will be developed and will work in **Ciao Prolog**, so the first thing we have to do is to study this system, above all the functionalities which will serve for our purpose.

For the beginning, we sought the necessary information about Ciao in its web page:

http://www.clip.dia.fi.upm.es/Software/Ciao *Reference[19]*

In this page, which is included in the bibliography and references, we found a brief description about the Ciao Prolog project. Also we can find the available components of the Ciao Prolog Development System.

These components are:

- **ciao**: The Ciao Prolog system.
- **ciaopp**: The Ciao preprocessor.
- **lpdoc**: the lpdoc documentation generator

Our project is mainly developed in the Ciao Prolog system, so we focussed in studying it. We will not write here the characteristics of Ciao, although we will include them in this documentation.

After studying the main characteristics of Ciao system, we installed the lastest available version of it, which is: 1.8#2 of 2002/6/14. This version is available for several platforms but we install the one corresponding to the Unix system, which is where we have worked.

In the installation instruction it is recommended to use the text editor **E-macs** so we started to use it, and therefore to study this editor.

After installing the Ciao system and configure the Unix system to work it correctly, our task of familiarisation with the system started. Here, not only we had to learn to use Ciao, although we had to recall our knowledge about Prolog language.

References [1] to [5] and [8] to [17].

From here we began to carry out some basic programs in Prolog that permit us to acquire the basic knowledge that we needed for the project. We focussed on file management and change of characters since the main part of the project is based of that.

Concretely inside the Ciao Prolog system, we were interested in the Prolog libraries, which implement the interface with Tcl/tk (`tcltk.pl` and `tcltk_low_level.pl`), so after studying Prolog, we focussed in them. These libraries were the ones that we were going to use in the project so it was very important to understand well its use and implementation.

A full description of the library is included in Appendix C.

We will describe in a general way that the interface works and how the connection between Prolog and Tcl/tk is created.

Creation and initiation of the connection between Ciao Prolog and Tcl/to

There are two ways of initiate the connection between Prolog and Tcl/Tk, both of them through **Prolog predicates, so it is from the Prolog side where the connection is always initiated**.

The following Prolog predicates are implemented in the library **`tcltk.pl`**

- `tcl_new (-Tclinterpreter).`

  Creates a new Tcl interpreter, initialises it and return a handle to it in `Tclinterpreter`.

- `tk_new(+Options,-Tclinterpreter)`

  Performs basics Tcl and Tk initialisation and creates the main window of the Tk application.

Both predicates call the same Prolog predicate, which is implemented in the low level library **`tcltk__low_level.pl:`**

- `new_interpreter(-Tclinterpreter,+Option)`

  Creates two sockets to connect to the wish process, the term socket and the event socket, and opens a pipe to process *wish* in a new shell.

  Note: Wish is the interpreter that provides Tk.

This last predicate takes charge of creating the connection and initiating it.

As we said before, this predicate creates two sockets for the process communication, as illustrate by figure 2.

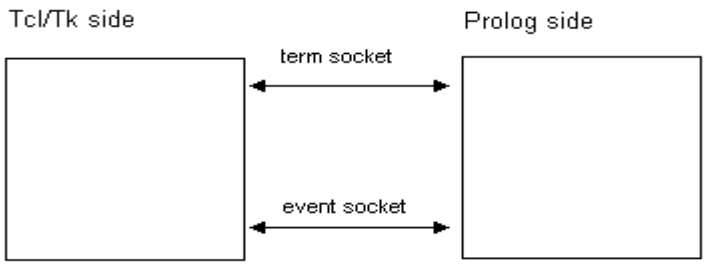

Figure 2. Process Communication

Once the connection is created, the **Prolog side sends a set of Tcl procedures** to the Tcl side. From this moment, these procedures are available to be used from Tcl, and they are the ones, which provide the basic functionalities of the interface in the Tcl side. Also some variables are creates in the Tcl side.

A brief description of the **Tcl variables** is:

- `event_socket.`
- `term_socket.`
  (These two variables are the connection between Prolog and Tcl.)

- `prolog_variables.`
  (It is an array that contains the results obtained from the Prolog side)

- `terms.`
  (list of terms)

A brief description of the **Tcl procedures** is:

prolog

Sends to `term_socket` the Prolog predicate `tcl_result` which contains the goal to execute. Returns the string executes and the goal.

prolog_event

This procedure takes an argument called `term` and adds the new `term` to the `terms` queue.

prolog_delete_event

Deletes the first `term` of the `terms` queue.

prolog_list_events

Sends all the `terms` of the `terms` queue by the `event_socket`. The last element will be `end_of_event_list`.

prolog_cmd

Receives as an argument the tcltk code. Evaluates the code and returns through the `term_socket` the term `tcl_error` if there was a mistake in the code or the predicate `tcl_result` with the result of the command executed. If the argument is *prolog* with a goal to execute, before finishing, the predicate evaluated by prolog is received. In order to get the value of the variables, predicates are compared using the `unify_term` procedure. Returns 0 when the script runs without errors, and 1 if there is an error.

prolog_one_event

Receives as an argument the *term* which is associated with one of the tk events. Sends through the `event_socket` the `term` and waits for the unified `term` by prolog. After that it calls the `unify_term` procedure to obtain the value of the `prolog_variables`.

prolog_thread_event

Receives as an argument the *term* which is associated with one of the tk events. Sends through the *event_socket* the *term* and waits for the *term* unified by prolog. After that the *unify_term* procedure is called to obtain the value of the *prolog_variables*. In this case the *term_socket* is non blocking. It means that the execution will continuing without waiting for the results from the Prolog side.

convert_variables

Receives as an argument a string which contains symbols that can't be sent by the sockets. This procedure deletes them from the input string and returns the new string.

unify_term

Receives as argument a prolog term.

The value of any of the variables in the goal that is bound to a term will be stored in the array prolog_variables with the variable name as index. The string which contains the printed representation of prolog *terms* is *Terms*.

Once the connection is created, the two process continue executing, and the Prolog part has the control. We could say that Ciao Prolog works like a server, and the Tcl/tk part like a client.

This description is taken from Ciao Prolog Manual. *Reference [19]*

Tcl/Tk Familiarization

Up to now, Tcl has been an unknown language for us, so we had to start from zero.

We found very useful the book *"Tcl and the Tk toolkit"*
Reference [6]
The main Web page dedicated to Tcl/Tk is:

http://www.tcl.tk *Reference [20]*

This page not only offers a great quantity of information about Tcl, but also provides the last versions of the language ready to download on the platform the user desires. It was here where we downloaded the Tcl/tk version which we have worked with:

Tcl/Tk 8.4.3 may 20, 2003

In the documentation section of this web page, we could find very interesting, links like Tcl books, tutorials, FAQs, and more things*. Reference [20]*

For us, the tutorials were very useful. We could learn about Tcl, and also we could see the results of our work learning and programming Tcl.

Once we downloaded the last version and installed it in Linux, we started to program basic applications, increasingly more complicated until we had a satisfactory knowledge to develop our project.

Vtcl Familiarization

Unfortunately the available information on Vtcl is not so much as Tcl/Tk. Nevertheless, we found very interesting Internet web pages that were very useful in order to make us to understand this development environment more clearly. The main page about Vtcl is:

http://www.vtcl.sourceforge.net *Reference [34]*

From here we downloaded the necessary software to work with Vtcl being the last version:

Vtcl 1.6.0b2

Also the documentation page was very useful. As we did with Tcl/Tk, we followed various tutorials that helped us to program and develop applications in Vtcl.

One of the things, which was useful to learn how to develop applications in Vtcl, were the samples included in this development environment.

For example we studied the code of a powerful text editor implemented with this tool and inTcl code, with this we learnt to handle files, menus, strings, and data I/O.

With Vtcl we had another task to do, since the code files generated by this application, were the ones that we were going to work with, in the development of our project. Because of this, we had to analyse and to understand the structure of a file generated by Vtcl, since we would have to translate this file to a Prolog code.

Working together

Once we knew all the parts separately, we were ready to work integrating them.

First of all, we studied all the samples, which Ciao provides. The main objective of these samples is to show how all the predicates provided in the library work. So we could realise which Prolog predicates and Tcl/Tk procedures were the most useful for us and for the project.

The most part of the samples are based in the message communication between Ciao Prolog and Vtcl, except one, which is based in the execution of Prolog predicates from Tcl, which was the thing we were interested in from the beginning.

This sample is a simple application, which implements an interface to calculate the factorial of a number. The user has to introduce an entry with the number, which will be calculating the factorial. After the user presses the corresponding button, the application sends to the Prolog part the predicate, and Prolog calculates the result. After Prolog sends this result to the application which shows it in the screen.

As an example of how difficult is to create an application directly implemented with the Prolog predicates provides by Tcl/Tk library, we will give the code of the sample described above:

```
:- module(tcl_factorial,[test/0]).
:- use_module(library('tcltk/examples/tk_test_aux')).
:- use_module(library(tcltk)).

test :-
        tcl_new(X),
        test_aux(X).

test_aux(X) :-
        tcl_eval(X,[button,'.b',min(text),dq('Factorial')],_),
        tcl_eval(X,[button,'.c','-text',dq('Quit')],_),
        tcl_eval(X,[entry,'.e1',min(textvariable),'inputval'],_),
        tcl_eval(X,[label,'.l1',min(text),dq('El factorial de ')],_),
```

```
tcl_eval(X,[pack, '.l1','.e1'],_),
tcl_eval(X,[entry,'.e2',min(textvariable),'outputval'],_),
tcl_eval(X,[label,'.l2',min(text),dq('es  ')],_),
tcl_eval(X,[pack, '.l2','.e2'],_),
tcl_eval(X,[pack,'.b','.c',min(side),'left'],_),
tcl_eval(X,[bind,'.b','<ButtonPress-1>',
            br([set,'inputval','$inputval','\n',

                prolog_one_event,dq(write(execute(tk_test_aux
                :factorial('$inputval','Outputval')))),'\n',
                set,
                'outputval','$prolog_variables(Outputval)'])]
                ,_),

tcl_eval(X,[bind,'.c','<ButtonPress-1>',
    br([prolog_one_event,dq(write(execute(exit_tk_event_loop)
    ))])],_),

tk_event_loop(X).
```

The main functions which are used in this sample are:

- **tcl_new/1**

  It is a Prolog predicate that we have described before.

- **tcl_eval/3:**

Usage: `tcl_eval(+TclInterpreter,+Command,-Result)`

  - *Description:* Evaluates the commands given in Command in the Tcl interpreter TclInterpreter. The result will be stored as a string in Result. If there is an error in *Command* an exception is raised. The error messages will be *Tcl Exception:* if the error is in the syntax of the tcltk code or *Prolog Exception:*, if the error is in the prolog term.
  - *Call and exit should be compatible with:* +TclInterpreter is a reference to a *Tcl* interpreter. (tcltk:tclInterpreter/1) +Command is a *Tcl* command. (tcltk:tclCommand/1) -Result is a string (a list of character codes). (basic_props:string/1)

- **prolog_one_event**

  This is a Tcl procedure which takes an argument, called goal, and sends it to Prolog. After this obtains the results sent by Prolog across the socket and calls `unify_term.`

- **unify_term**

This is also a Tcl procedure which receives two argument, one of them is the term sent to Prolog and another is the result. The function of this procedure is to unify the variables which were unified in the Prolog execution and adds them to the vector prolog_variables with the name as index.

- **`tk_event_loop`**

  This is a Prolog predicate which waits the arrival of a Prolog event which is a goal to be executed. After the execution, sends the result to tcl and remains waiting again.

A full description of these Prolog predicates can be found in Appendix C

TCL LIBRARIES

As we have described before the Tcl library should provide methods that permit calls to Prolog from Vtcl in a natural way.

These methods should be based on the functionalities that are provided in existing libraries in Ciao Prolog. Before continuing we will give a brief description of the facilities that those libraries give us, above all of those that we have used more.

The library, which implements the interface between Tcl/Tk and Prolog, is called tcltk. The interaction between both languages is implemented as an interface between two processes, a Tcl/Tk process and Prolog process.

The Prolog part encodes the request from a Prolog program and sends it to the Tcl/Tk via a socket.

As we said before, the useful predicates for us, the procedures and Prolog predicates, which make the communication between the two parts, are `prolog_one_event (Tcl procedure)`, in the Tcl side, and `tk_even_loop` (Prolog predicate) in the Prolog side.

- `prolog_one_event`: (Tcl Procedure)

  This Tcl procedure receives one argument (`a_term`), which it is sent across the event socket. After this, the Tcl procedure waits for the results in the term socket. Once we have the result, `prolog_one_event` calls the Tcl procedure `unify_term` which has two arguments (`a_term` and `result`). The part of the **Tcl code** corresponding to the main part of the Tcl procedure is:

```
puts $event_socket $a_term
flush $event_socket

gets $term_socket result

set ret [unify_term $a_term $result]
```

  This function is very useful for the Tcl library. The problem is that the use of this function is not so natural. For example, if we want call the predicate p_sample/2 which is in the module foo, we have to write:

```
prolog_one_event "execute(foo:p_sample(X,Y))"
```

- `unify_term`: (Tcl procedure)

  This Tcl procedure is the one, which unifies the result obtained from Prolog and the term as argument in the Tcl procedure `prolog_one_event`. Also adds in the array `prolog_variables` the variables that were unified in the call.

Errors in the Current Interface

When we started to make tests with these procedures, we realised that we obtained wrong results, so we analysed them step by step looking at any action which the code made. The result of the analysis was that we found some errors in the implementation of these two procedures, concretely in `unify_term`.

Error One:

When we executed `unify_term` the received arguments are:

- *term*:

```
execute(foo:p_sample(X,Y))
```

- *result*:

```
foo:p_sample(12,'string')
```

The objective of the Tcl function `unify_term` is to unify these two strings and add two more elements to the `prolog_variables` array:

```
prolog_variables(X) = 12

prolog_variables(Y) = 'string'
```

To do this, the first thing which is done by `unify_term` is to eliminate:

"`execute(`" and "`)`"

This is implemented in this way:

```
set long [string length $term]
incr long -3
set term [string range $term 8 $long]
```

This is written in pseudo code as:

```
long = length of the term
long = long -3
term = from term[8] to term[long]
```

The figure 3 shows where the error is:

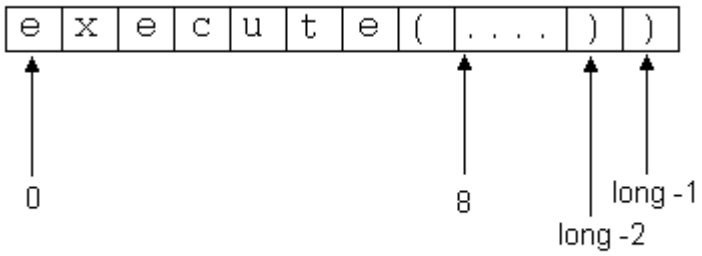

Figure 3. Programming Error

As we can see in the picture, the error is in the instruction that decrement `long` in 3 points, when it should decrement `long` in 2 points.

So the correct code is:

```
set long [string length $term]
incr long -2
set term [string range $term 8 $long]
```

After eliminating the "`execute(`" and "`)`" from term, the procedure `unify_term` calls an auxiliary procedure (`unify_term_aux`). This last procedure is the one that really unifies *term* and *result*. The other error is here, in `unify_term_aux`:

Error Two:

The Tcl procedure `unify_term_aux` works as show the figure 4. The error appears when the last argument of the predicate that has to be unified is of the form `head(…)`. When this happens, the unification takes the argument from the last position to the character before the first ")" in the string. This means that unifies only `head(…` without the ")" corresponding to the argument. For example, suppose that:

```
term = foo(X,p(Y))
result = foo(4,p('string'))
```

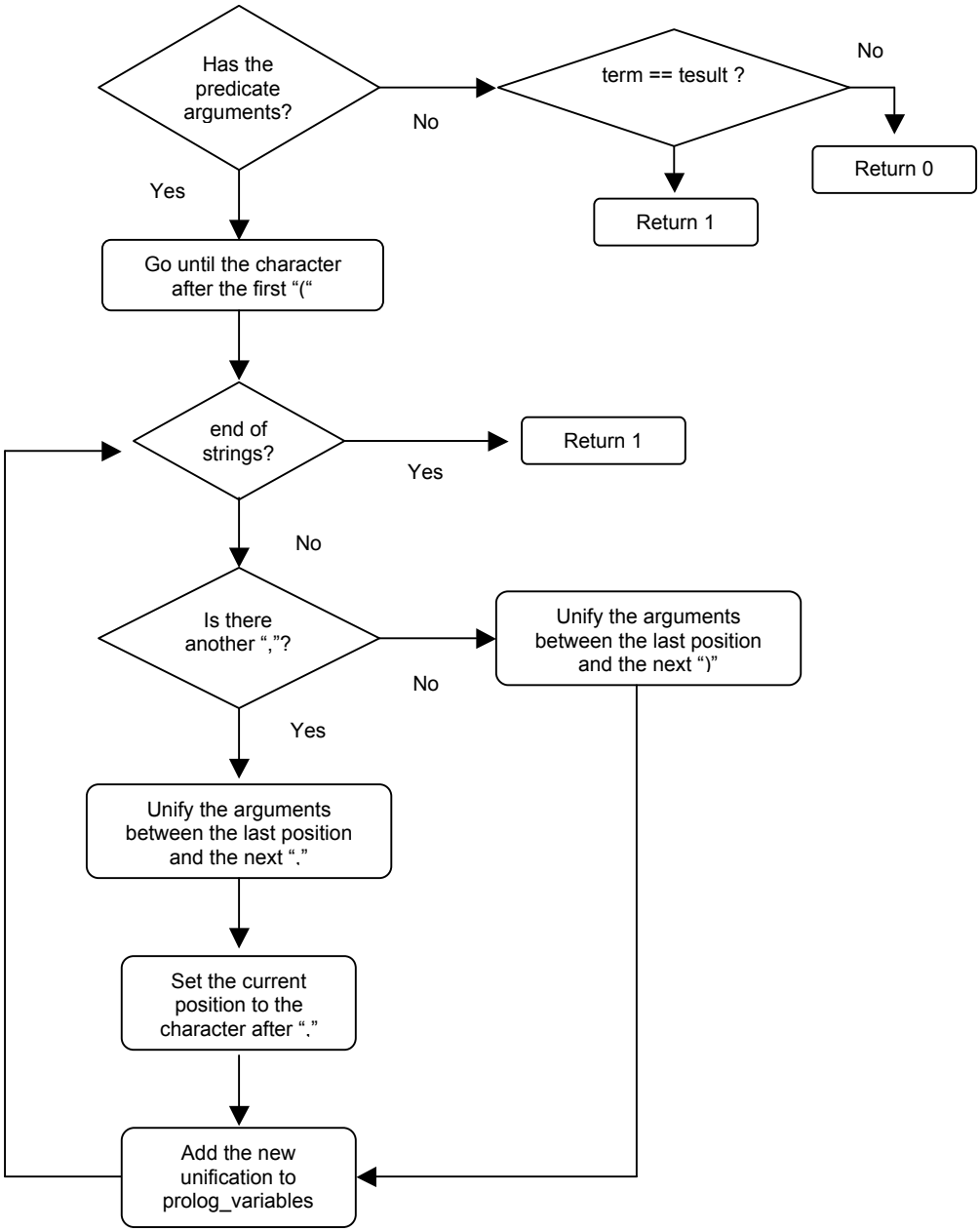

Figure 4. unify_term_aux Flow Diagram

Then, `unify_term_aux` unifies

```
X = 4, and
p(Y) = p('string'
```

when the correct unification is

```
X = 4
p(Y) = p('string')
```

A solution for this error is to check if there are more characters “)” when no more “,” are found. If yes, the unification has to consider this alternative. If no, then the procedure has to do the same it did before.

The errors explained above are one of the reasons because we decided to include in the project a Tcl library which solves the errors. Also because we wanted to provide to the user a natural way calling Prolog predicates. Anyway, we have modified the original libraries solving the errors, and we included in the project source these modified libraries.

Focussing ourselves in the design of the Tcl library, we considered the basic utilities that we had to provided to the future user. These are:

- A Tcl procedure which receives a term to be executed in the Prolog side (goal).
- A Tcl procedure that exits the application from the loop in the Prolog side.

These functionalities are enough to implement any future procedure that improves the library. But also, we wanted to add one more:

- A Tcl procedure that takes all the existing variables unified in the array `prolog_variables` and creates new variables that can be accessed directly by their names.

Implementation

The library had to be implemented in Tcl code, in order to give to the user an easy way including it in his Vtcl project. This library is `prolog.tcl`, and it is included in the source code of the project. It has to be included in the Vtcl project in order to be able to use its functionalities. We can find the following procedures in it:

`prolog_command.` This Tcl procedure implements basically the same functionality as `prolog_one_event`, but without using the Tcl procedure `unify_term`. Also it is easier to use it, because the argument is a Prolog call, and the string `"execute(…)"` is not needed. When the Tcl procedure finishes its execution, all the variables unified are available in the array `prolog_variables` with the name as index. The Tcl code of the Tcl procedure is:

```
proc prolog_command {term} {

 global event_socket
 global term_socket
 global prolog_variables

 set result 0

 puts $event_socket execute($term).
 flush $event_socket

 gets $term_socket result


 set ret [unify_term_aux $term $result]
}
```

Note: For the correct use of the Tcl procedure, the module containing the goal that the user wants to call has to be indicated, as in the original procedure `prolog_one_event`.

Example of use:

```
prolog_command foo:my_predicate(X,Y)
```

Where foo denotes the Prolog module containing the Prolog predicate my_predicate/2.

`prolog_exit.` This Tcl procedure sends to the Prolog side the instruction to exit from the main loop, and ends the execution of the application. The main objective of this procedure is to hide the instruction `"exit_tk_event_loop"` , and make use of the low level libraries in Prolog transparent for the user. It has the same effect as:

```
prolog_one_event "execute(exit_tk_event_loop)"
```

and

```
prolog_command exit_tk_event_loop
```

but it is much easier. The Tcl code corresponding to the Tcl procedure is:

```
proc prolog_exit {} {
    prolog_command exit_tk_event_loop
}
```

`prolog_variables_out`. This predicate is an auxiliary function that we decided to add in the library in order to handle the results of the Prolog calls easier for the user. It has no arguments, and when it is called, takes out from the array `prolog_variables` all the existing variables. For each occurrence in the array, the procedure creates a new global variable with the same name that it had into the array (its index).

For example, suppose that:

```
prolog_variables(v1) = 23
prolog_variables(v2) = "John"
prolog_variables(v3) = 2.456
prolog_variables(v4) = foo(7,'string')
```

Then, the procedure creates the new variables (`v1, v2, v3, v4`) with the values:

```
v1 = 23
v2 = "John"
v3 = 2.456
v4 = foo(7,'string')
```

And the four variables can be accessed directly by their names.

The corresponding code for this procedure is:

```
proc prolog_variables_out {} {

 global prolog_variables

 set var_name 0
 set l [array get prolog_variables]

 set max [llength $l]

 for {set pos 0} {$pos<$max} {} {
     set var_name [lindex $l $pos]
     incr pos
     global $var_name
     set $var_name [lindex $l $pos]
     incr pos
 }
}
```

How to include this library in VTcl applications and Tcl programs.

The libray described before is implemented in the Tcl file

```
prolog.tcl
```

Included in the source directory of the project.

To include the functionality of this library into a Vtcl applications the user has to do the following steps:

1. Select in the Vtcl tool bar **File → Source.**
2. Select the `prolog.tcl` file.

From this moment the Tcl procedures are available to be used.

To include the library into a Tcl file, the Tcl programmer has to write the following Tcl command in the initialisation of his program:

```
source prolog.tcl
```

TRANSLATOR

The translator is the main part of the project. Its objective is to create a Prolog file which contain the encoding of the corresponding application developed in Vtcl. The reason for translating the application is that in this way we can compile it as a normal Prolog file, and we can execute it as a normal predicate.

Before starting to design the translator, we have to consider the different alternatives we have, when we want to translate Tcl code to Prolog code. This can be done in many different ways, so we have to analyse all of them and choose the one which could be best suited to our purpose.

As the developed application is going to be a Prolog program or a Prolog predicate, no matter which alternative we choose, the Tcl code will always be sent to the Tcl side from the Prolog side. This means that the main program, and the control of the execution will be on the Prolog side. Thus, the alternatives will be differentiated by the codification form for the Tcl code, and the way it is sent to the Tcl side.

Code Translation / Encoding

The first problem that we have is to decide how to encode and translate the Tcl code into Prolog code. This task is related with the form of sending the code to the Tcl side.

There are two ways of sending the Tcl commands by the sockets that connect both parts. The first one is sending every command codified according with the `tclCommand` specification that exists in the Ciao libraries. In the other hand we can send the Tcl commands as simple strings, which will be interpreted on the Tcl side. We will describe these two alternatives:

Command-by-Command.

From this point of view, a Tcl/Tk file could be seen as a set of Tcl commands, separated by a character of new line. So we can translate them separately into Prolog code. Here, we can use the facilities provided in the `tcltk` library in Ciao Prolog. In the library, a Tcl command is specified as follow:

```
Command          --> Atom  { other than [] }
                   | Number
                   | chars(PrologString)
                   | write(Term)
                   | format(Fmt,Args)
                   | dq(Command)
                   | br(Command)
                   | sqb(Command)
                   | min(Command)
                   | ListOfCommands
ListOfCommands   --> []
                   |[Command|ListOfCommands]
```

where:

Atom
denotes the printed representation of the atom.

Number
Denotes their printed representations.

chars(PrologString)
denotes the string represented by *PrologString* (a list of character codes).

write(Term)
denotes the string that is printed by the corresponding built-in predicate.

format(Term)
denotes the string that is printed by the corresponding built-in predicate.

dq(Command)
denotes the string specified by *Command*, enclosed in double quotes.

br(Command)
denotes the string specified by *Command*, enclosed in braces.

sqb(Command)
denotes the string specified by *Command*, enclosed in square brackets.

min(Command)
denotes the string specified by *Command*, immediately preceded by a hyphen.

ListOfCommands
denotes the strings denoted by each element, separated by spaces

Reference [19].

With this specification, we can sent any Tcl command using the following predicated implemented in `tcltk.pl`.

`tcl_eval(+TclInterpreter,+Command,-Result)` (This predicate was described before)

Then, the translator would work fetching one command each time, and translating it into a tclCommand specified above. The final Prolog file would be a list of calls to the predicate `tcl_eval`, every one of them containing the original Tcl command.

This alternative seems to be much too complicated and it will probably take a lot of computation time. The next one gives a better solution to this problem.

Sending the commands as strings.

It is also possible to send strings directly to the Tcl side. These strings are interpreted as commands, and thus executed.

So, from this point of view a Tcl/Tk file is also a string of characters itself, so we can send the whole file across the socket connection to the Tcl side. All the initialisation of the interface between Ciao and Tcl/Tk is done in this way.

The problem here is simplified to two simple actions:

1. Read the Tcl/Tk file
2. Store the file as a list of characters

Once we do that, we can consult the list stored and send it through the socket connection.

This alternative is more simple than the first one, and also gives an efficient solution.

Program Structure

Once we have decided how encode the Tcl file, we have to design the structure of the Prolog file that the translator have to generate automatically.

In this point of the project the help of Manuel Hermenegildo and Daniel Cabeza was very useful. They are members of the CLIP group (described above) and we had the opportunity to meet them in RUC, since they were here to participate in a meeting of the ASAP project.

We were talking about the main objectives that should have our project, and the general structure that the translator should provide.

From this point we started to consider the different ways in which we could design our project. Again we find various alternatives in the design.

1. Generate a Prolog file, containing the application, from which it is possible to obtain an executable directly.

2. Generate a Prolog file, which is a Prolog module, and a main Prolog file, which is the one that have to be compiled to obtain the executable. This last Prolog file includes the Prolog module, and both are generated automatically. The Prolog module includes the Tcl file codification and all the predicates necessary to handle the application.

3. Generate only a Prolog file that is a module that includes the codification of the Tcl file, and all the predicates necessary to handle the application from Prolog. This module can be included in any program, and the application can be called from it.

In the following figures, the different alternatives are explained graphically.

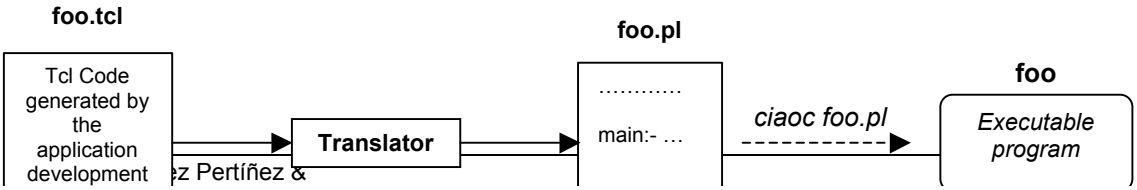

Figure 5. Alternative 1. The Prolog file can be compiled directly

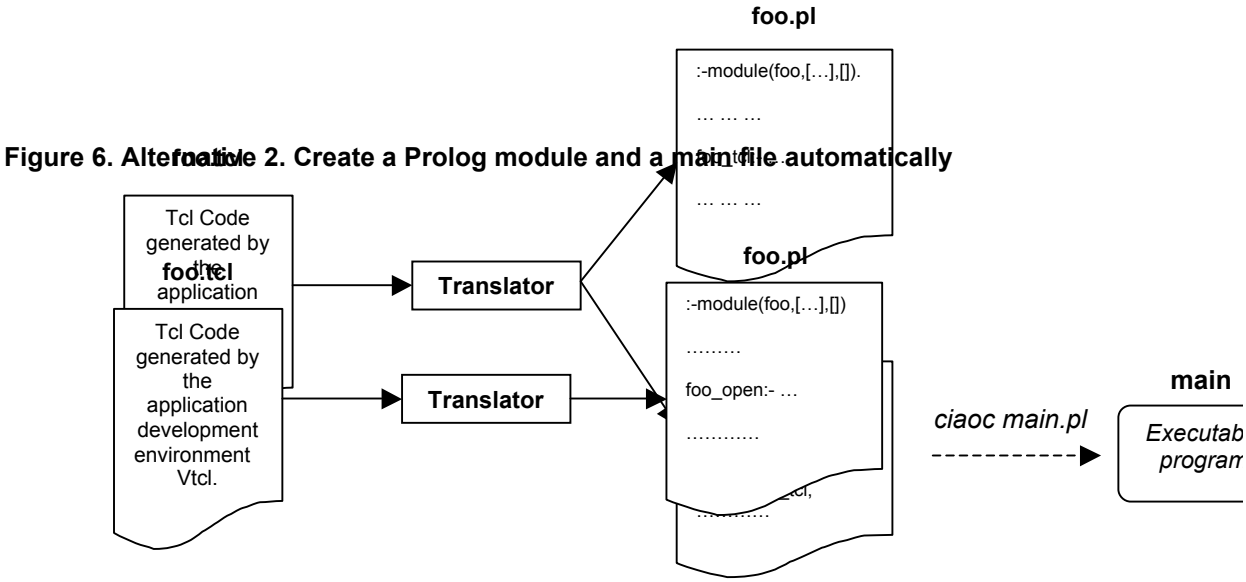

Figure 6. Alternative 2. Create a Prolog module and a main file automatically

Figure 7. Alternative 3. Create only a Prolog module

Now we have to consider the advantages and disadvantages for each given alternative.

1. The first of the alternatives is maybe the most intuitive for the future application programmer. In this case, the file can be directly compiled without needing to do anything else to run the application.

2. With the second alternative we obtain 2 files from the translator, and one of them can be compiled in order to run the application. Here, the module that codifies the Tcl code has some exported predicates that can be used from other modules and Prolog programs.

3. The third could seem the most simple of the three, but this is more an advantage than a disadvantage. The module generated by the translator provides not only the codification of the Tcl code, but also all the predicates necessary to manage the application from a Prolog program. The translator does not generate more files than the module, so the output can be easily handled.

From the point of view of a Prolog programmer, the most useful alternatives are the last two, so we discarded the first one from the beginning. The objective was to provide a flexible module from which the programmer could use and run the application, and not to create only an executable program.

So now, we have to decide which of the last two we should choose. These alternatives and their characteristics were discussed with our supervisor, John Gallagher, and also with the CLIP members Manuel Hermenegildo and Daniel Cabeza.

Since the second one is a solution between the first and the third, we decided early to continue with the design of it.

The file `main.pl` that should be generated by the translator is just a simple Prolog program that can be implemented easily by the programmer. But we want to provide also a template that can be edit and modify by the programmer in order to make his executable.

With this alternative, we not only give all the facilities that the third provide, also we indicate how to program the executable. If the programmer wants to use the application in other way, he has all the facilities included in the Prolog module.

Module Structure

The next step in the design was to decide which structure was going to be the module generated by the translator.

We have said before that the module should provide the predicates necessaries to manage the application. These predicates are:

- Exported Predicate: a predicate that initiates the application. This includes opening the wish process and setting the connection up.

- Internal Predicates: these are predicates used internally by the module. Here we include the predicates necessaries to create the wish interpret, send the Tcl code, and receive the events from the Tcl side.

- Facts: there is also a fact included in the module. This fact contains a list of characters that codifies the original Tcl code.

Concretely, these predicates are:

- **`<module_name>_tcl_open/0`**. This predicate is the only one which is exported and it has three simple actions:

    1. Create and initiate the interpreter.
    2. Send the application Tcl code.
    3. Wait for events sent by the Tcl side.

    `<module_name>` denotes the name of the module. It means that the name of the predicate changes with the different modules. This allows to the programmer to use different applications in the same program, and it is an easy way to remember with which application he is working in every moment. For example, if the module is called `foo`, then the predicate name is `foo_tcl_open`.

- **`create_interpret(-TclInterpret).`** This predicate creates and initiates the interpreter which is returned in `TclInterpret`.

- **`send_code_vtcl(+TclInterpret).`** Sends the Tcl code codified in the module to `TclInterpret`.

- **`vtcl_receive(+TclInterpret).`** This predicate waits for events sent by the Tcl side, and executes them.

The fact has the form:

- **`code_vtcl/1`**. Codifies all the Vtcl application code as a list of characters.

The translator generates all the corresponding code for the predicates described above automatically.

Main file structure.

The main file (`main.pl`) generated automatically by the translator has a very simple structure. Actually it is just a template which allow us to build the Prolog program easily.

The only thing that the programmer has to do is **include in the template all the modules that the application uses**. The structure of the template is:

```
:-module(_,[main/0],[]).

:-use_module(<module_name>).

%-------------------------------------------------------------
%Here the programmer has to included the modules used by the
%application
%
%:-use_module(…)
%… … …
%
%-------------------------------------------------------------

main:-
      <module_name>_tcl_open.
```

Where <module_name> is the name of the module created automatically by the translator. It is also the name of the Tcl file created by Vtcl.

Implementation

The implementation of the translator can be found in the directory /source in the project, in the file ***vtcl2prolog.pl***. It receives an argument which is the Tcl file that contains the implementation of the application developed with Vtcl.

At last, the translator only generates two Prolog files (the main file, and the module), so this is the main task of it. The following figure shows the normal execution of the translator:

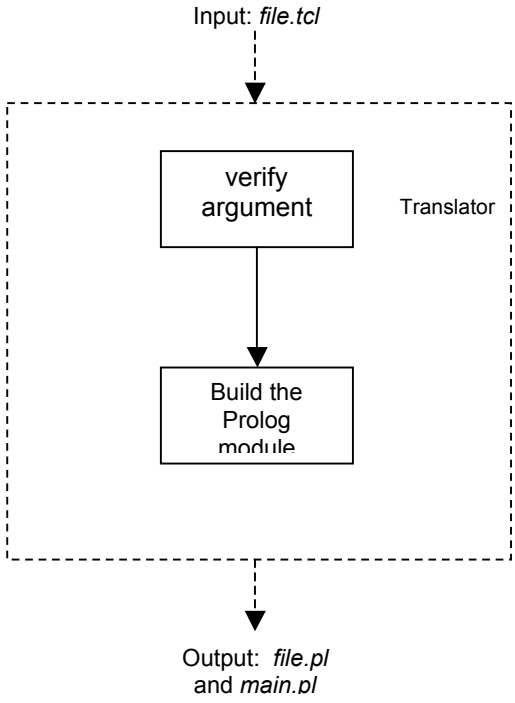

Figure 8

After the verification of the argument passed to the program, the translator builds the pl file as shows the next figure:

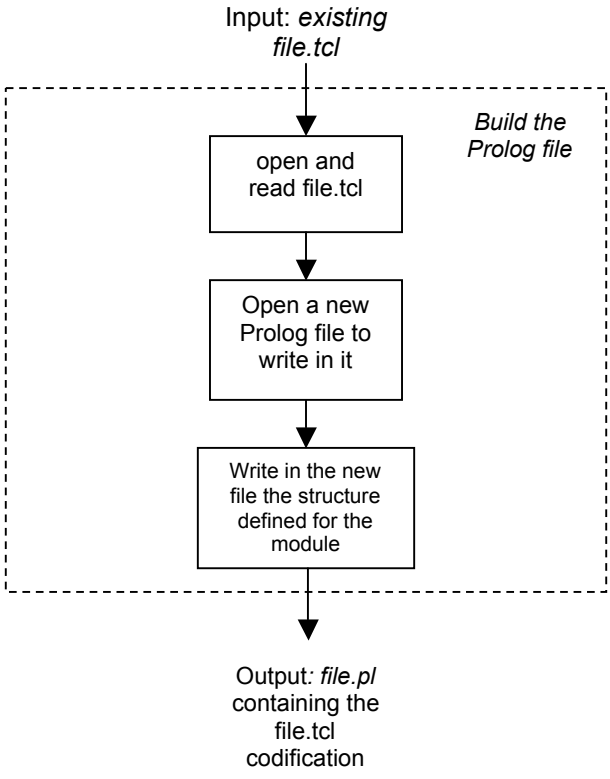

Figure 9. Build the Prolog file

SAMPLES AND TESTS

INTRODUCTION

Once we finished the project, it is time to make some tests to see that the results of these tests correspond with the hypothesis we have about them.

- Hypothesis:

  Suppose we have a few Prolog predicates, suppose also we want to do an interface for them using a visual language, and then, we want that the output of that visual language is converted to a Prolog module which could be used as a normal module in normal Prolog programs.

- Results expected:

  The visual language will be Vtcl, and so we will make a few interfaces for Prolog predicates in this language, and after this we will use our translator and then we will try to use the out of our translator in a normal Prolog program and as well as in the Program console.

The tests must be simple enough to understand the whole working of the translator and also enough powerful to see how it works with real programs and interfaces.

Now we will describe the two samples we implemented to check the right operability of the project.

TESTS

Sample 1 (Mini Calculator)

The first example we made to understand how the translator works is a very simple calculator.

We have a few Prolog predicates, which make a few mathematical operations such as plus, minus, divide and multiply.

The code of these Prolog predicates is:

```
:- module(plus,[plus/3],[]).
plus(In1,In2,Out):-
    number(In1),
    number(In2),
    Out is In1 + In2.


:- module(minus,[minus/3],[]).
minus(In1,In2,Out):-
    number(In1),
    number(In2),
    Out is In1 - In2.
```

```
:- module(multiply,[multiply/3],[]).
multiply(In1,In2,Out):-
    number(In1),
    number(In2),
    Out is In1 * In2.


:- module(divide,[divide/3],[]).
divide(In1,In2,Out):-
    number(In1),
    number(In2),
    Out is In1 / In2.
```

The steps we made are as follows:

1. First we designed the interface before implementing it in the visual language.
2. After this we implemented this interface using Vtcl.

As we can see in the figure 10 the interface has seven main widgets. These widgets are: three entries (Input 1, Input 2 and Output), and four buttons corresponding with the operations implemented n the Prolog modules.

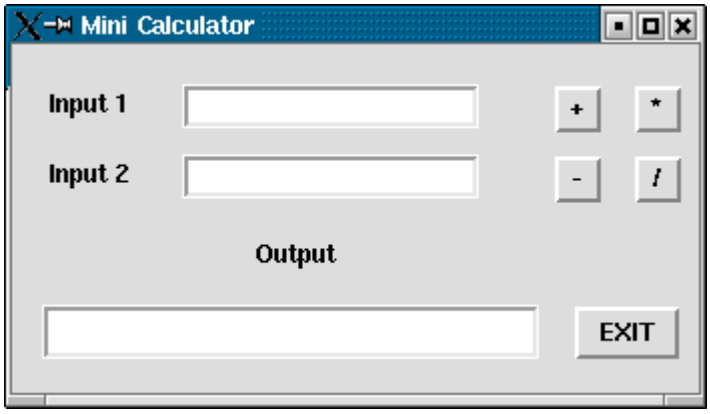

Figure 10. Interface for Mini Calculator

3. Now we have to write the Tcl code for the buttons.

   We use the Prolog libraries to make a call to the Prolog predicates, which will solve the mathematics operation.

   The Tcl code for the plus button is

```
prolog_command plus:plus($input1,$input2,Output).
set output $prolog_variables(Output)
```

   Where:

`input1` and `input2` are the variables assigned to the entry widget in the interface, These variables are not typed because in Tcl there are no types for the variables.

`Output` is where the result of the predicate will be stored. Actually the result will be stored in `prolog_variables(Output)` as we have explained before.

The rest of the buttons are implemented in the same way

4. We saved this interface and it generated a tcl file which was called:

```
interface.tcl
```

5. We gave this file to our translator as an entry using the next command:

```
./vtcl2prolog interface.tcl
```

6. Our translator generated two pl file which are called:

```
interface.pl
main.pl
```

7. Now we have to edit the file `main.pl`, in order to included in it the modules that the application uses (plus.pl, minus.pl,…). The code is:

```
:-module(_,[main/0],[]).

:-use_module(interface).

%-------------------------------------------------------------
%Here the programmer has to included the modules used by the
%application
%
:-use_module(plus).
:-use_module(minus).
:-use_module(multiply).
:-use_module(divide).
%… … …
%
%-------------------------------------------------------------

main:-
      interface_tcl_open.
```

We notice that we have to call our module with the name:

```
modulename_tcl_open
```

which is the way the module is generated automatically by our translator.

8. After this we just have to compile this main with Ciao Prolog command:

```
ciaoc  main.pl
```

9. And then just execute the main like

```
./main
```

10. Now we have our application running so to finish just try some entries and see if it works correctly. For example

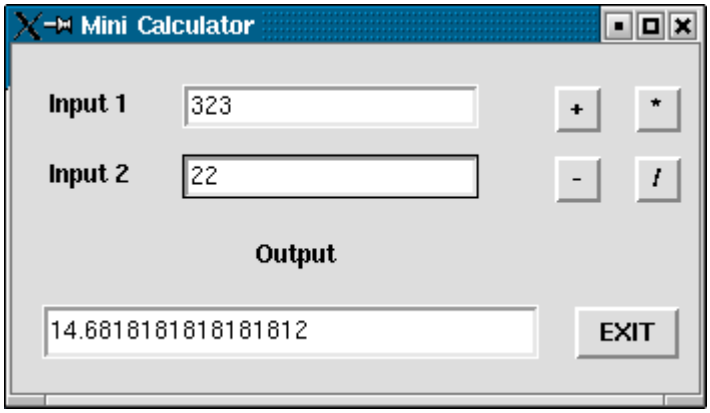

Figure 11. Testing Mini Calculator (Divide)

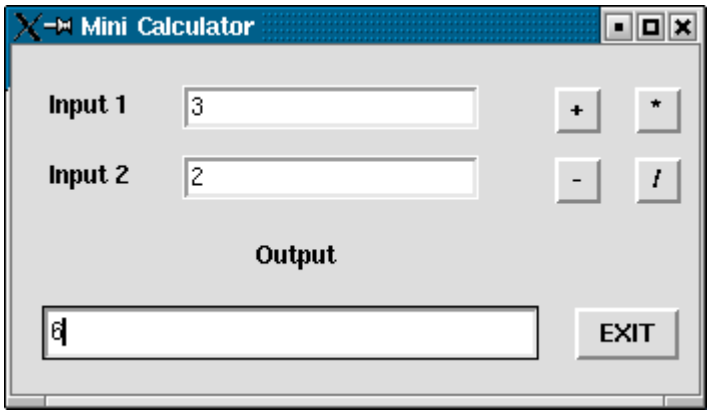

Figure 12. Testing Mini Calculator (Multiply)

Sample 2 (NFTA analyser interface)

Now we want to see if we can make interfaces for more complicated programs, so our Supervisor gave us an idea about make and interface for an analysis program, which he is programming.

This program is called NFTA Analyser, but we will not describe here how it is implemented or what it does because we will centre in the function of our translator, which doesn't need to know about the function of the program for which the interface is built.

The steps we will make are very similar to the steps we made in the sample 1

1. First step is again design the aspect of the interface; we made this with our supervisor John Gallagher, making some sketches about how it could be.

2. This interface is not like our mini calculator, because we have different application in one program so we will make a main window where the other application were called:

   The main window has this aspect:

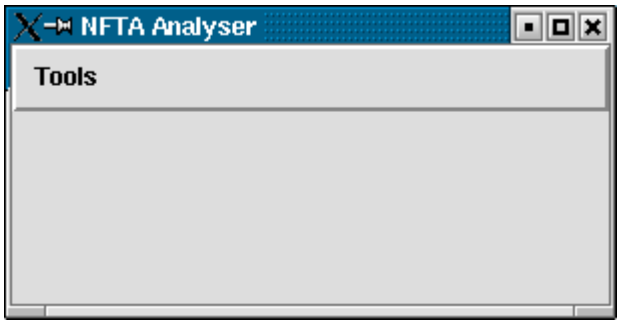

Figure 13. NFTA Analyser

3. This window has a menu and we can call the other applications, which would make the call to the Prolog predicates.

   The other application look like this:

   The first application is for the type analyser:

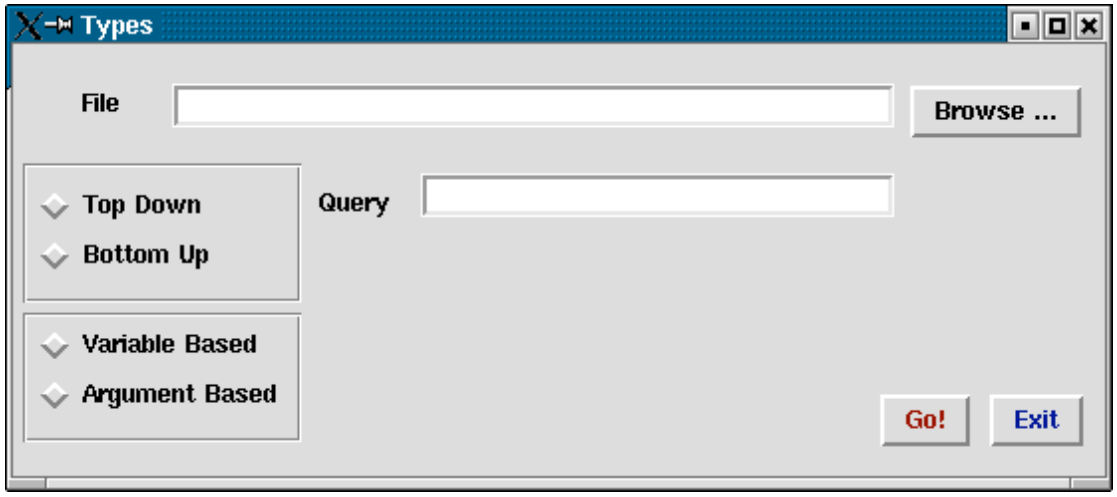

Figure 14. Types

The next application is for tp:

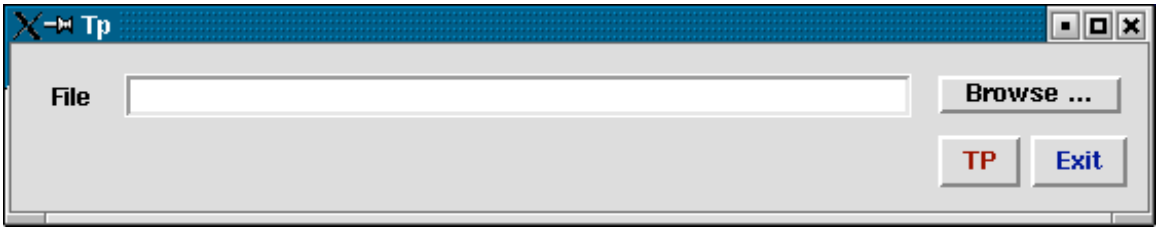

Figure 15. Tp

And the last one for QA Consult:

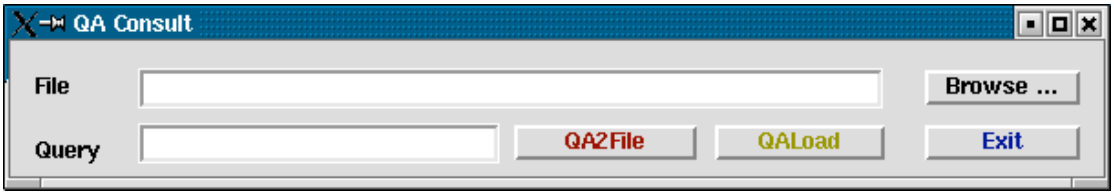

Figure 16. QA Consult

4. Using our Prolog library we call the Prolog predicates to make the analysis and other functions.

5. After finish the implementation of the interface we will save it and the Vtcl will generate this tcl file:

```
analyser.tcl
```

6. Now and as in the last sample we will use our translator with this file and it will generate a pl file which will be the module we wanted:

```
./vtcl2prolog analyser.tcl
```

7. Then we have the pl files, the module included which can be used like a module in another programs.

```
analyser.pl
main.pl
```

8. To check it works correctly we edit again the main Prolog file generated by the translator, in order to included the modules needed.
   This main is like this:

```
:-module(_,[main/0],[]).

:-use_module(analyser).
```

```
%-------------------------------------------------------------
%Here the programmer has to included the modules used by the
%application
%
:-use_module('type/main').
:-use_module('tpqa/qa_trans').
:-use_module('tpqa/tp').
%
%-------------------------------------------------------------

main:-
      <module_name>_tcl_open.
```

We have to remember to call the module in the way:

```
modulename_tcl_open.
```

9. Now we have to compile the main in Ciao Prolog:

```
ciaoc main.pl
```

10. And then execute this main:

```
./main
```

11. Now we have our application running, to make this work correctly we have to remember keep the auxiliary files in the correct directories because they will be using during the running of the application, these files are the Prolog predicates of the Analyser.

12. So we will make a few tests with the application to check the correctly working of it:

We open a file to be analyse:

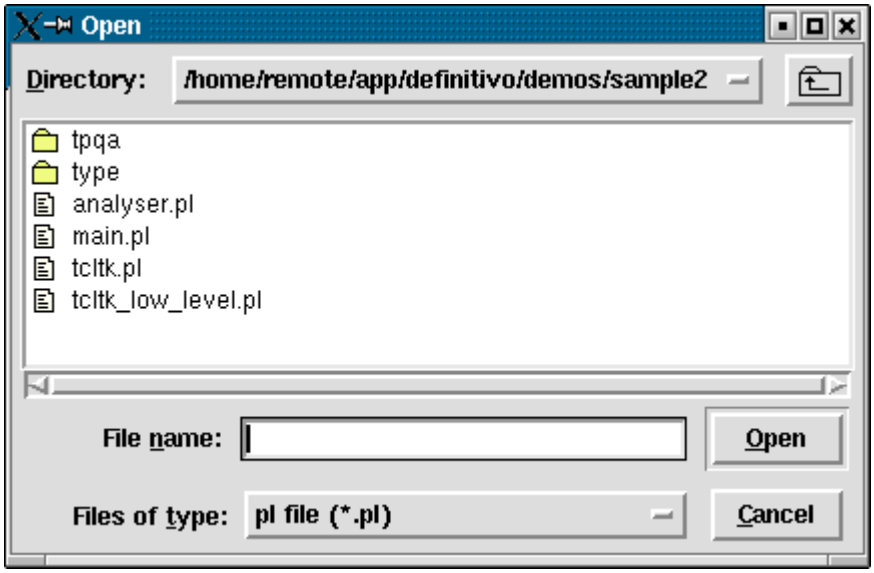

Figure 17. Browsing a file

Then we call the application we want to check, and put some entries, for example:

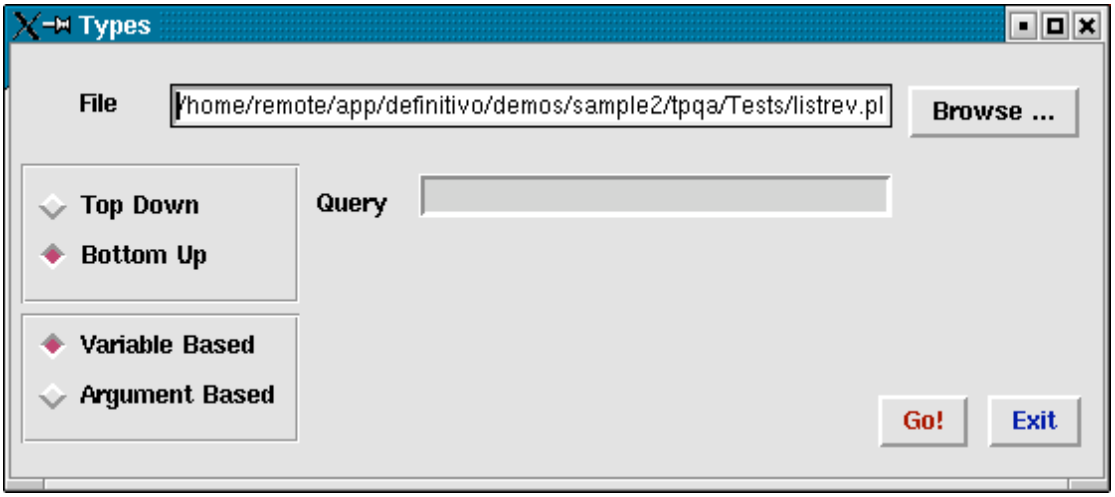

Figure 18. Testing Types Application

Then press the GO! Button and we see the results:

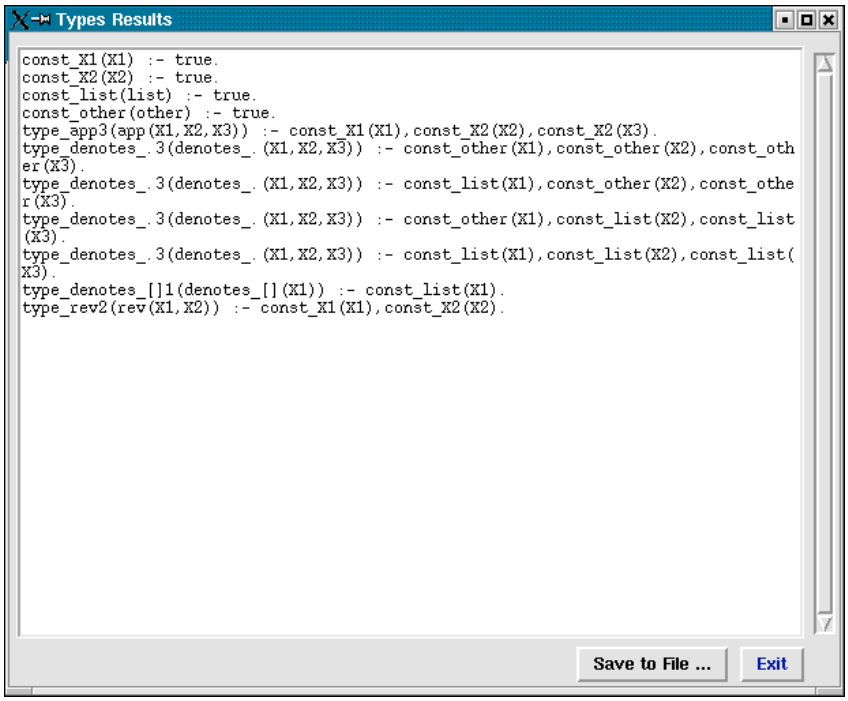

Figure 19. Types Results

And we can do the same with the other applications such us:

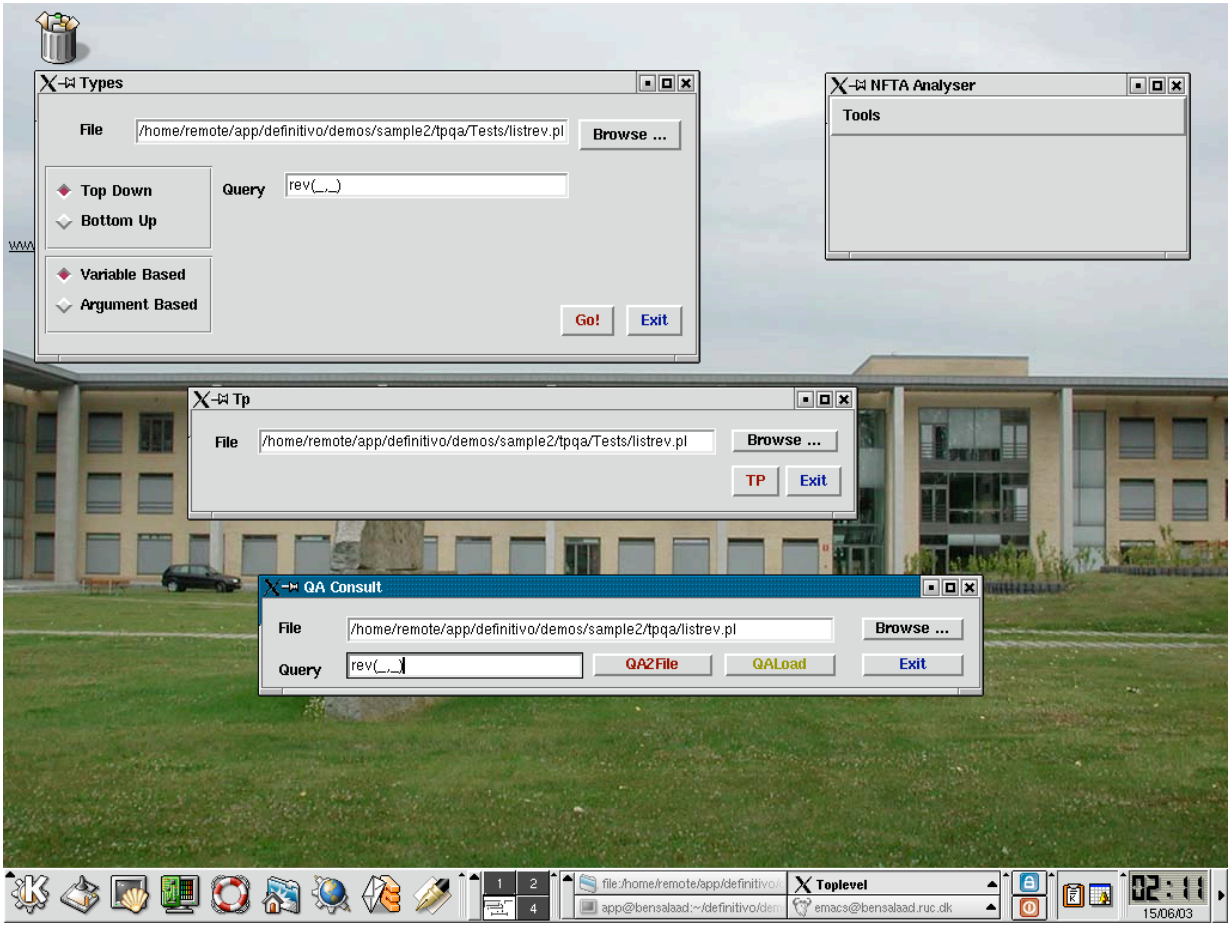

Figure 20. NFTA Analyser

USER MANUAL

The program developed in the project is quite simple to use, but we want to explain here some things related that have to be specified.

We will explain how to use our application following the steps that any future user will do when he wants to build an interface for his programs written in Prolog.

BUILDING THE INTERFACE…

Once we have our Prolog code distributed in modules, we may want to build a graphical interface for them.

The first thing we have to do is design and implement the interface (What the program will look like) in the application development environment Vtcl. We included in this documentation a small and very easy to follow Vtcl tutorial, so we will not write here how to build the interface.

Suppose that the interface is now built. The next step is to add the code that the Vtcl application has to execute. Inside this code, we may want to call our Prolog predicates included in their respective modules.

To be able to call a Prolog predicate from the Vtcl application, we have to include the Prolog-Tcl library implemented in our project (`prolog.tcl`). This library contains a set of procedures that have to be added to the *Function List* of the application. To do this, we have to choose the following options in the main toolbar of Vtcl:

File, and then
Source

Then, we browse the file that implements the library (`prolog.tcl`), which is included in the project source. After this, we can see the functions added in the *Function List*, and we are able to use them.

In the description of the library, we explained how to use the procedures provided properly. But it is important to say that any Prolog predicate we want to call has to be preceded by the module where the predicate is implemented.

For example, if we want to call the predicate `foo(X,Y)`, which is included in the module *module1.pl*, the call has to be specified as `module1:foo(X,Y)`. So a sample command that implement this call in Tcl is:

```
prolog_command module1:foo(X,Y)
```

Handling variables and values

In Tcl all the variables and values are strings, so when we want to call a Prolog predicate from the Tcl side, the arguments and also the predicate itself are strings.

For this reason, the Prolog predicates and their arguments have not to be parsed in the Tcl side, since it is the Prolog side the one which do it.

For example, suppose that we have the following Prolog predicate:

```
module1:foo_out(+Value1,+Value2,-Output)
```

Where:

`+Value1` and `+Value2` denote that the first two arguments of the predicate are input arguments.

`-Output` is the output of the Prolog predicate.

Suppose now that we want to call the Prolog predicate above from the Tcl side with the values stored in the following variables:

```
var1, var2.
```

And we want that the result will be stored in:

```
prolog_variables(Out).
```

Then we have to write:

```
prolog_command module1:foo_out($var1,$var2,Out)
```

Once we have written all the code that makes the interface works, we have to save it. Vtcl creates a Tcl file which is the one that we have to work with in the next steps.

TRANSLATING THE APPLICATION…

Now we have a Tcl file, which implements our Vtcl application. The next thing to do is to execute the application `vtcl2prolog`, which is included in the directory /bin in the project.

This application receives an argument which is the file containing the Tcl code. If the application is executed without arguments, it shows a usage message in the standard output, as show the following figure:

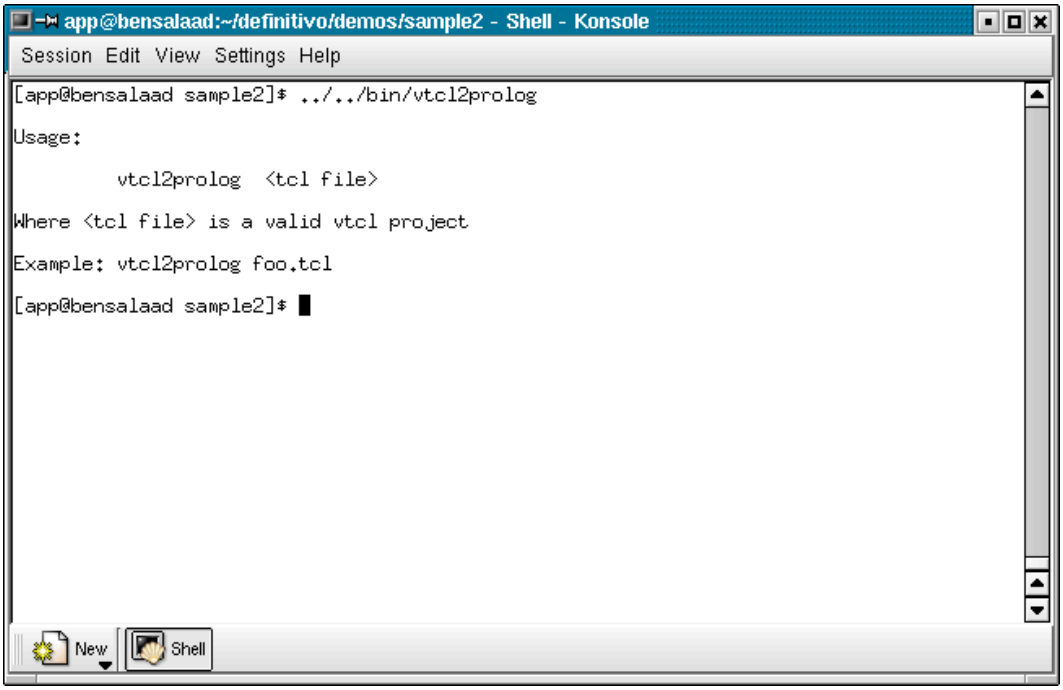

Figure 21. vtcl2prolog execution

For example, if the Tcl file is foo.tcl, we have to execute in the console:

```
vtcl2prolog foo.tcl
```

This will generate another file call with the same name as the application, but with Prolog extension.

In the example above would be `foo.pl`.

This file is a Prolog module that codifies the application developed in Vtcl. This module can be included in any Prolog program just adding the line:

```
:-use_module(<module_name>).
```

In the example above:

```
:-use_module(foo).
```

Then, we can use the predicate that exports the module built, just calling it by its name: `<module_name>_tcl_open`.

The translator also generates a simple template that is a main file which can be compiled (adding first the modules needed by the application), to generate an executable.

COMPILING THE MODULE…

The module generated automatically by the translator uses two additional libraries. These libraries are originally included in Ciao Prolog, but we have modified them to solve some errors encountered during the implementation of the project.

These libraries are: `tcltk.pl` and **`tcltk_low_level.pl`**. Their specification can be found in the Ciao Prolog Manual Reference.

So, for the correct use of our project, the modules generated by `vtcl2prolog` have to be compiled with the modified libraries.

By default, the libraries are included in the module as if they were in the same directory, so we have to copy them to the directory where the generated module is.

RESOURCES

HARDWARE RESOURCES

The computers which are in the Computer Science department are connected in a local net. We also have an account where we can save our projects and documents, so wherever we work we can reach our saved files, if it is inside the Computer Science Department.

The computers we have available in the Usability Labs were:

Pentium III Family 6 Model 8
Stepping 3
AT / AT Compatible
261.668Kb Ram

The Computers we have in our computer lab were the next:

Pentium III Family 6 Model 8
Stepping 3
AT / AT Compatible
261.668Kb Ram

Intel ® Pentium ® 4 CPU 1.50 GHrz
AT / AT Compatible
1.310.192 Kb Ram.

And also with full access to internet.

SOFTWARE RESOURCES

For this project we used two platforms, Linux and Windows. We worked mainly in Linux platform where we developed the most part of the project.

The Linux version we used in the development of the project was:

- **Red Hat Linux**

  Distribution Version: Red hat Linux release 7.1 (Seawolf).
  Operating System version: #1 SMP wed Mar 13 10:19:26 EST 2002

The windows version we used was the next:

- **Microsoft Windows 2000**
  5.00.2195
  Service Pack 3

The version of the program we used to develop the project were:

- **The Ciao Prolog Development System.**

  Current version (1.8#2 of 2002/6/14)

- **Tcl / Tk Programming Language**

  Current version Tcl/Tk 8.4.3 May 20, 2003

- **VTcl Visual Tcl Programming Language**

  Current Version Vtcl 1.5.5 June 27, 2001

- **Emacs text editor**

  Current version GNU 20.7.1

CONCLUSIONS AND POSSIBLE IMPROVEMENTS

In the last step of our way, it is time to look back and see what we did and take a while to think about it and about the next steps after finishing the project.

We think we made the most part of the things we wanted to do at the beginning of the semester. There were many alternatives to do and many interesting possibilities at the beginning, but obviously, it was not possible to carry on all of them

so we decided to take this, make the translator and develop our knowledge about Prolog (Ciao Prolog System) and visual languages.

Of course there are many things we still can do in our project but we think the main thing is finished and ready to work and to be improved easily just continuing in the way we started.

So now someone else can take our documentation and continue working in the same subject making some things that we also suggested in the possible improvements or just use our program like a tool to develop another projects.

POSSIBLE IMPROVEMENTS

Thinking about the results we have obtained in our project, we can give some possible improvements. This is a way to indicate to future project developers how to continue our job.

There are mainly two possible improvements that could be interesting to add in our project and in the libraries implemented in Ciao Prolog.

When we were testing our project, we realised that it could be interesting to execute two different applications from the same main Prolog program at the same

time. Also, when we execute an application from the Prolog console, it could be interesting that the application runs at the same time that we are making consults by hand. A possible solution for that is to introduce threads.

Another improvement that we have seen necessary is to add some functionality to the libraries `tcltk` and `tcltk_low_level`.

Adding Threads

Ciao Prolog provides functionalities that allow working with threads. It is possible to take advantage of them.

In general terms, it is possible to create a thread with the predicate or application that we want to execute. This thread can be running separately from the main process, and it can be managed with the handlers provided by the thread predicates.

We did not have time to design and implement this option, but we have studied the threads libraries, and we can give an idea about how to do it.

A way to develop this could be to add a new functionality to the module automatically generated by the translator. This could be a new predicate that runs the application in a thread. The library **`concurrency`** existing in Ciao Prolog gives some facilities to work with concurrent process.

Adding new functionalities to Tcl/Tk libraries

As we have seen during the development of the project, these libraries provide enough facilities to implement any application that use a connection between Prolog and Tcl/Tk. Nevertheless, it is possible to add some new predicates that could make easier developing said applications.

For example, and in relation with adding threads, there is not any predicate that allows to the Tcl side to take its time handling the events corresponding in running a graphical application (file events, window events, time events…). For this reason, we find interesting a new predicate that gives the control to the Tcl side for a while.

In order to manage easier the design of an application from the Prolog side, we should add more functionality oriented to this purpose.

For example, predicates that manage information about the widgets and windows existing in the application.

APPENDIXES

APPENDIX A: TCL MANUAL

We don't want to make a complete description about how to programme in Tcl because this is not the main purpose of this report, but we think that could be interesting to know a few things about Tcl, which could be useful in order to, understand better the project.

Structure of Commands

Basic Structure

To execute a command, you type the name of a command followed by a space-separated list of arguments.

Example:

```
set user_id 48290
```

The command above is named set and its two arguments are user_id and 48290. A command is terminated by a newline (no; is needed).

Nested Commands

You can also can nest commands by putting the embedded commands in square braces ([and]).

Example:

```
set first_element_of_list [lindex $my_list 0]
```

The Tcl interpreter first interprets `[lindex $my_list 0]` and then uses the results of that as the 2nd argument to the set command.

String Quotes

If an argument contains a space, you need to enclose it in string quotes: either quotation marks ("") or curly braces ({}).

Example:

```
set whole_name "Alex Samoyed"
```

The difference between quotation marks and curly braces is that variables are interpreted within quotation marks but not within curly braces. In the above example, we set the variable whole_name to have the value Alex Samoyed.
If we execute

```
set sentence "My favorite dog is $whole_name."
```

the value of sentence will be My favorite dog is Alex Samoyed. If we, instead, do

```
set sentence {My favorite dog is $whole_name.}
```

the value of sentence will be My favorite dog is $whole_name.

Variables

Data Types

Tcl has only three data types:

- *Strings.* In Tcl, virtually everything is a string. Text fragments, binary fragments, dates, numbers, etc. are all treated the same. This is ideal for web programming, since web browsers only understand strings.
- *Lists.* Lists are ordered collections of strings.
- *Arrays.* Arrays are key/value pairs.

Declaring Variables

You do not need to declare variables before using them in Tcl.

Setting Variables

```
set variable_name variable_value
```

Retrieving Variable Values

```
$variable_name
     OR
set variable_name
```

Seeing if a Variable is Set

```
info exists variable_name
```

This will return 1 if the variable has a value (may or may not be null), or 0 if the variable has never been set. This is useful, for example, when you are processing the results of an HTML form, and you don't know whether the user selected any of the radio buttons (if not, the variable associated with the buttons will not exist).

Strings

Things you can do with strings:

- set them:

  ```
  set dog_name "Alex"
  ```

- add to them:

  ```
  append dog_name " Samoyed"
  ```

  Now dog_name is Alex Samoyed.

- compare them:

  ```
  string compare $dog_name "Alex Samoyed"
  ```

- The above code fragment will return 0 if the two strings are identical. Example usage:

```
if { [string compare $dog_name "Alex Samoyed"]==0} {
   append page_contents "What a nice name you've chosen\
   for your dog!"
}
```

- check the length:

```
string length $my_string
```

  Example usage:

```
if { [string length $password] > 20 } {
append page_contents "The password you selected is\
too long."
}
```

- get part of the string:

```
string range $my_string 0 4000
```

- change it to lower or uppercase:

```
string tolower $my_string
         OR
string toupper $my_string
```

Naming Conventions

Although any character can theoretically be used in a Tcl variable name, it's best to stick to lowercase letters, numbers, and underscores (_). Use underscores to separate separate "words" within the variable name, e.g., my_files.
If your Tcl variable corresponds to a column name in the Oracle database (e.g., user_id), give it the same name in your Tcl script

Lists

Things you can do with lists:

- create an empty list:

```
set student_list [list]
```

- create a non-empty list:

```
set student_list [list "Joseph Tally" "Ricardo
Portillo" "Alex Samoyed"]
```

- add new elements to it:

```
lappend student_list "Ingrid Aquino"
```

- see how many elements are in it:

```
llength $student_list
```

- get a specific element from the list:

```
lindex $student_list 0
```

- Note that in Tcl, the first element of the list is element 0.

```
search for an element within the list:

lsearch -exact $student_list "Ingrid Aquino"
```

This function returns either the index of the element if it was found in the list, or -1 if it is not present.

Example usage:

```
if { [lsearch -exact $student_list $user_name] != -1 } {
append page_contents "You are already a member of this
class."
} else {
append page_contents "Would you like to register for this
class?"
}
```

- concatenate two or more lists:

```
set everyone_list [concat $student_list $ta_list
$professor_list]
```

- split a string into a list:

- split $student_string ", "

For example, say you have a string called student_string with the value Joseph Tally, Ricardo Portillo, Alex Samoyed. It may be convenient to turn it into a list, e.g., if you want to search for a name or loop through each of the values. To do this, type set student_list [split $student_string ", "].

- put the elements of a list into a string:

```
join $student_list "<br>"
```

This is useful, for instance, for printing out the values in a list on a web page.

- loop through the values in a list:

```
foreach student $student_list {
```

```
    # check if they've turned in their homework and, if
not,
    # send email to the instructor
}
```

Naming

You don't have to end list variable names with _list, but it's a nice convention.

Internal Representation

Internally, Tcl stores lists as strings. This means that you can perform many string functions, such as string tolower on lists.

Arrays

Creating Arrays

You can use any strings (including integers) as array keys.
Example:

```
set slide_title(0) "About Tcl"
set slide_title(1) "Structure of Commands"
set slide_title(2) "Strings"
```

Another example (associating colors with their hexadecimal values):

```
set color(red) "#ff0000"
set color(green) "#00ff00"
set color(blue) "#0000ff"
```

However, you have to be careful if your keys contain spaces. This won't work:

```
set color(light blue) "9999ff"
```

Instead you must use:

```
set color(light\ blue) "#9999ff"
```

or

```
set {color(light blue)} "#9999ff"
```

or

```
set current_color "light blue"
set color($current_color) #9999ff"
```

Retrieving values

```
$color(blue)
```

or

```
set color(blue)
```

or

```
$color($current_color)
```

Seeing what values are defined

View all keys:

```
array names color
```

This will return all keys in a Tcl list.
See if a specific key has been defined:

```
info exists color(beige)
```

This will return either 1 or 0 (just like when info exists is used with any other type of variable).

Numbers and Arithmetic

Arithmetic is not done directly by the Tcl interpreter. It is done by calling the C library using the expr command on arithmetic expressions. The detailed parsing rules for arithmetic expressions depend on the particular Unix implementation, but they are more or less like in C.

Examples:

Basic arithmetic:

```
expr $a + $b
expr $a - $b
expr $a * $b
expr $a / $b
```

Important: integer division truncates. Example:

```
set c [expr 10 / 4 ]
```

This will set c = 2.

To make division behave as expected, either the numerator or the denominator must be changed into a floating point number.

```
set c [expr 10.0 / 4]
```

Now c = 2.5, as expected.

Trick for turning an integer into a floating point number: put a decimal point (.) at the end of it. Example:

```
set c [expr $a. / $b]
```

Remainder:

```
set remainder [expr $a%$b]
```

Other examples:

```
set q [expr sin(.598) + cos(.44)]
set td_width [expr round(100/$number_of_columns)]
set degrees_celsius [expr (($degrees_farenheit -
32)*5)/9.]
```

Reference:

- abs(x)
- asin(x)
- acos(x)
- atan(x)
- atan2(y,x)
  atan2 returns the angle theta of the polar coordinates returned when (x,y) is converted to (r, theta).
- ceil(x)
- cos(x)
- cosh(x)
- double(x)
  returns x as a double or floating point.
- exp(x)
  returns e^x
- floor(x)
- fmod(x,y)
  returns the floating point remainder of x/y.
- hypot(x,y)
  returns the square root of the sum of x squared plus y squared, the length of the line from (0,0) to (x,y).
- int(x)
  truncates x to an integer.
- log(x)
  returns the natural log of x.
- log10(x)
  returns log base 10 of x.
- pow(x,y)
  returns x to the y power.
- round(x)
- sin(x)
- sinh(x)
- sqrt(x)
- tan(x)
- tanh(x)

Pattern Matching

Regular Expressions

Tcl has two regular expression commands: regexp and regsub.

- regexp

Generally you use regexp to:

(a) see if a variable value matches a pattern
(b) create new variables with values that are part of that pattern

Usage:

```
regexp expression string ?matchVar? ?subMatchVar
subMatchVar ...?
```

regexp returns 1 if string matches expression

I assume you've studied regular expressions in previous programming courses, but here are some examples to refresh your memory:

Plain string:

```
regexp "galileo" $email
```

Example usage:

```
if { ![regexp "galileo" $email] } {
   set error_message "You can't use this system."
}
```

One-character wildcard (.)

```
student..
```

will match the following:

```
student01
student23
studentaa
student%!
```

zero-to-infinity-character wildcard (*)

```
student*
```

will match the following:

```
student
student01
students in my class are very good
student01<br>student02<br>student03<br>...</body></htm>
```

The regular expression parser in Tcl is "greedy", meaning that it matches as many characters as it possibly can.

Range of characters

```
student[0-9][0-9]
```

will match the following:

```
student01
student77
```

but will not match:

```
student6
student100 (in this case, student10 will be
matched)
studentaa
```

Note that when using square brackets in Tcl, Tcl will try to interpret everything inside the brackets, unless they are within curly braces. Therefore, you must use one of the two following forms for your regular expression:

```
regexp {student[0-9][0-9]} $class_information_page
regexp "student\[0-9\]\[0-9\]" $class_information_page
```

List of characters

```
student[123]
```

will match the following:

```
student1
student2
```

but not:

```
student12
```

Zero or more instances of a set of characters (*):

```
student([0-9]*)
```

will match:

```
student01
student4920482
student
```

One or more instances of a set of characters (+):

```
student([0-9]+)
```

will match:

```
student01
student4920482
```

but not:

```
student
```

Not (^)

```
student([^<]*)
```

This will match "student" and any characters after it, until it hits a less-than sign. If you're regexping through something like this

```
Team 1:<br>student01<br>student02<br>student03...
```

this will match

```
student01
```

A few special characters

- \n matches a newline
- \r matches a carriage return (used in Windows; in Unix these usually show up as ^M)
- \t matches a tab
- Using what the regexp matched:

```
regexp {student([0-9]+)} "student01<br>student02..." match
```

will create a variable called match with the value student01.

```
regexp {student([0-9]+)<br>student([0-9]+)}
"student01<br>student02..." match submatch1 submatch2
```

will create a variable called match with the value student01
student02, a variable called submatch1 with value 01, and a variable called submatch2 with value 02.

regsub

regsub uses the same regular expression rules as regexp. It is used to perform substitutions based on regular expression matching.
Usage:

```
regsub expression string
thing_to_replace_expression_with new_string
```

Example:

```
regsub -all "-" $creditcard_number ""
clean_creditcard_number
```

This will find all the dashes (-) in the creditcard_number, replace them with the empty string, and put the result into a variable called clean_creditcard_number. If you don't use the optional -all switch, regexp will only substitute the first occurrence of -.

String Match

For simple regular expressions, its more efficient to use the Tcl command string match than regexp.
Usage:

```
string match pattern string
```

It can match plain strings, or you can use the following wildcards:

- * (zero or more characters, same as in regexp)
- ? (a single character, like . in regexp)
- [] (lists and ranges of characters, same as in regexp)

Control Structures

Types of control structures:

- conditional
- looping (iteration)
- error-handling
- miscellaneous exit commands

Logic:

- AND: &&
- OR: ||
- NOT: !
- Equal: ==
- Inequalities: <, >, <=, >=
- Grouping expressions: use regular parentheses ( )

Conditional Statements

- IF

  Structure:

```
if {condition} {
   body
}
```

  or

```
if {condition} {
   body
} else {
   other_body
}
```

  or

```
if {condition} {
   body
} elseif {another_condition} {
   another_body
} elseif {one_more_condition} {
   one_more_body
} else {
   last_body
}
```

Example:

```
if { $order_total > 1000 } {
   append shipping_text "Shipping is free for orders
over Q1000!"
} elseif { $order_total > 500 } {
   append shipping_text "Your shipping cost:
Q[eve_shipping_cost $order_total],
   but if you spend Q[expr 1000 - $order_total] more,
your shipping
   will be free!"
} else {
   append shipping_text "Your shipping cost:
Q[eve_shipping_cost $order_total]"
}
```

- SWITCH:

  SWITCH is short-hand for some simple IF statements.

  Structure:

  ```
  switch value {
     pattern1 body1
     pattern2 body2
     ...
  }
  ```

  Here's an example of an IF statement that can be transformed into a SWITCH statement:

  ```
  if { $order_state == "authorized" } {
     set status_sentence "We are processing your
  order."
  } elseif { $order_state == "shipped" } {
     set status_sentence "All items in your order
  have shipped."
  } elseif { $order_state == "returned" } {
     set status_sentence "We have received your
  returned items."
  } else {
     set status_sentence "Unknown order status.
  Please contact customer service."
  }
  ```

  In the example above, $order_state is being compared to a string in each of the conditions. This can be rewritten more compactly as:

  ```
  switch $order_state {
     "authorized" {set status_sentence "We are
  processing your order."}
     "shipped"    {set status_sentence "All items in
  your order have shipped."}
     "returned"   {set status_sentence "We have
  received your returned items."}
     default      {set status_sentence "Unknown order
  status. Please contact customer service."}
  }
  ```

Looping Statements

- WHILE

Structure:

```
while { conditional_statement } {
   loop_body_statements
}
```

Example:

```
set i 0
while { [info exists slide_title($i)] } {
   append page_contents "<li> $slide_title($i)"
   incr i
}
```

- FOREACH

Structure:

```
foreach variable_name list {
   body
}
```

Example:

```
foreach student $student_list {
   # look up their grade in the database
   set grade [eve_get_student_grade $student]
   append page_body "<li> $student: $grade"
}
```

- FOR

Structure:

```
for start test next body
```

Example:

```
for {set i 0} {$i < $n_items_in_order} {incr i} {
   append page_contents [eve_item_description $i]
}
```

Error-handling

- CATCH

If your web script contains an error, the user will shown an error page. But you can prevent this:

- If it's a programming error, fix your bugs.
- If it's something you have no control over, wrap the statement in a CATCH.

CATCH returns 1 if there was an error, 0 otherwise.

Usage:

```
if { [catch {statement_that_might_fail}] } {
   stuff_to_do_if_the_statement_failed
} else {
   stuff_to_do_if_the_statement_succeeded
}
```

Example (using the AOLserver procedure ns_httpget which fetches a web page from foreign server:

```
if { [catch {set foreign_page [ns_httpget
"http://finance.yahoo.com/q?s=ORCL&d=t"]}] } {
   # our ns_httpget failed
   append page_contents "I'm sorry, we can't get
Oracle's
   stock quote right now. Please try again later."
} else {
   # success
   # do some regexp's to get the desired
information:
   # share_value and change_since_yesterday
   append page_contents "Value: $share_value


   Change since yesterday: $change_since_yesterday"
}
```

Important: do not overuse catch. You do not want users to believe something succeeded when it really failed.

Procedures

Defining Procedures

To define a procedure in Tcl use the following syntax:

```
proc procedure_name { list_of_arguments } {
   body_expressions
}
```

Note: in Oracle PL/SQL procedures that return values are called "functions." But in Tcl, a procedure is called a "procedure" regardless of whether it returns a value.

Example (no arguments):

```
proc eve_header_image {} {
   return "<img width=50 height=50 src=my-image.jpg>"
}
```

Example (2 arguments):

```
proc eve_header {page_title user_name} {

    set message_for_user "<p>
```

```
        Hello $user_name!"

        return "[eve_header_image]
        $page_title
        $message_for_user
        "
    }
```

Example (2 arguments, 2nd argument optional, defaulting to the empty string):

```
    proc eve_header {page_title {user_name ""}} {

        if { $user_name != "" } {
           set message_for_user "<p>
           Hello $user_name!"
        } else {
           set message_for_user ""
        }

        return "[eve_header_image]
        $page_title
        $message_for_user
        "
    }
```

Example (unknown number of arguments):

```
    proc eve_decode {args} {
        # This works like decode in Oracle.
        set args_length [llength $args]
        set first_value [lindex $args 0]

        # we want to skip the first & last values of args
        set counter 1
        while { $counter < [expr $args_length -2] } {
            if { [string compare $first_value [lindex $args
    $counter]] == 0 } {
                return [lindex $args [expr $counter + 1]]
            }
            set counter [expr $counter + 2]
        }
        return [lindex $args [expr $args_length -1]]
    }
```

Using Procedures

You use them just like any built-in Tcl command:

```
    eve_update_email_address $user_id $new_email
```

or

```
    set page_content "[eve_header "Useful Pi Formulas"
    $user_name]
    <ol>
      <li> pi = 4(1 - 1/3 + 1/5 - 1/7 + ...)
      <li> ...
    </ol>
    "
```

Reference [20], [27], [28] and [29]

APPENDIX B: VTCL MANUAL

Now we have an idea about the main things in Tcl we can start to learn how to program in Vtcl. This step should be easy if we know Tcl, because Vtcl is just an extension of Tcl, so the main things don't change, we only have more tools, and a very comfortable interface to build our program.

VTcl is also a full featured tool for generating mission critical applications and serves as a wonderful vehicle to meet application Graphical User Interface (GUI) needs. Let's take a quick peek at the layout and visual organization of VTcl.

- Main window:
    - General operations interface.

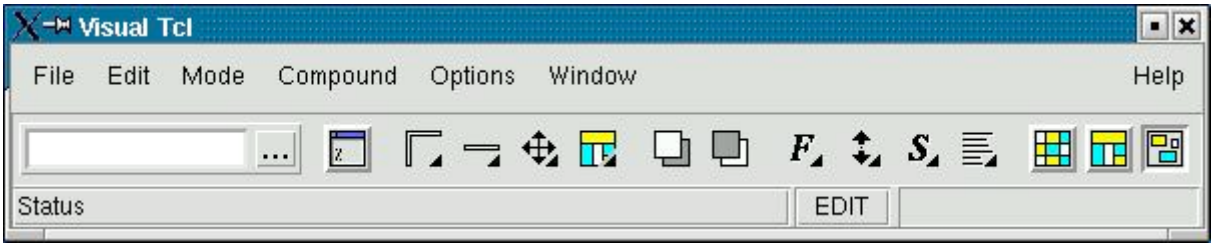

Figure 22. General Operations Interface

- Widget Toolbar window:

  - Provides easy access to available widgets for use in application development.

  - Toolbar configurable via vtsetup.tcl.

Figure 23. Widget Toolbar

- Attribute Editor window:

  - Provides easy access to widget attributes upon widget selection.

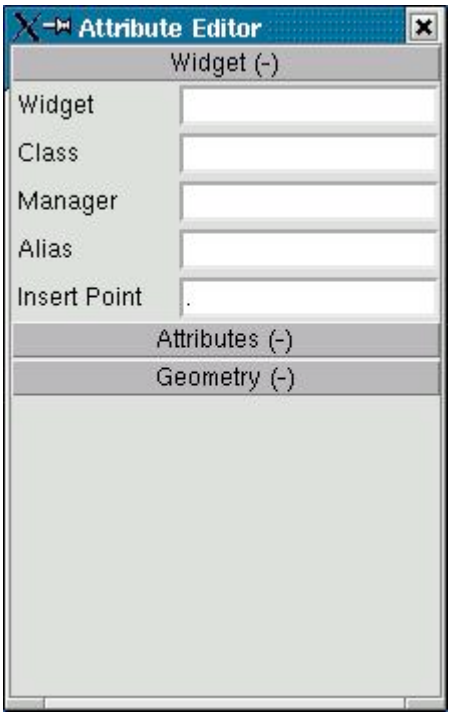

Figure 24. Attribute Editor

- Function List window:

  - Provides an easy method to create, modify, and delete functions (procedures).

Figure 25. Function List

- Window List window:

  - Provides access to toplevel windows.

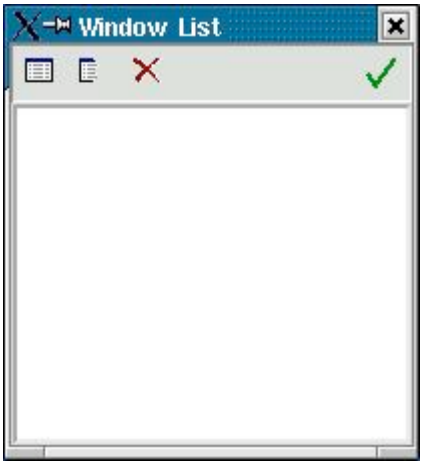

Figure 26. Window List

- Widget Tree window:

  - Provides a quick and easy method to navigate a project details.

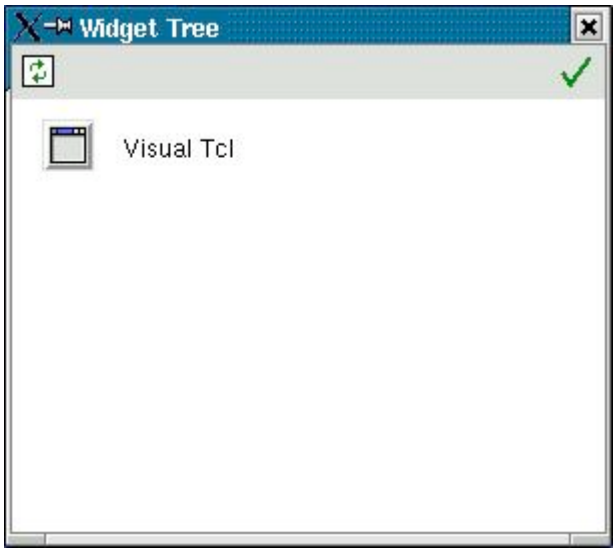

Figure 27. Widget Tree

so these are the main differents between Vtcl and Tcl.

And with the knowledge about Tcl and this interface it would be easy make our programs. Because as we said, Vtcl is just an interface tool for Tcl so we think we should not write any more about commands in vtcl because the main things are describe in the Tcl manual, as Variables, Strings and such kind of things which are the same for Vtcl.

Now, just an example of use. We will show how to program a simple application with VTcl. It seems that every book we have ever read starts with a simple "Hello World" application. So, we will do just that, for our "Hello World" application we will use:

- TopLevel - the main TopLevel.
- Buttons - event collection.
- 1 Label - display output.

The instructions for creating the "Hello World" application are as follows:

1 Start VTcl.

2 From the Toolbar window:

- select the toplevel icon (click once only) - this creates a new toplevel window for us to place our widgets on.
- select the button icon, then click on the toplevel we just created.
- repeat the above step for the second button.
- select the label icon, then click on the toplevel we created.

3 Reposition the widgets on the toplevel by merely selecting and dragging the selected widget where you desire it with your mouse.

4 Select the first button then on the Attribute Editor/Attributes/text, change the text on the button to say "Hello World!" and press enter to update the button.

5 Repeat the above step for the second button with the text "Hello User!".

6 Select the label and examine the alias setting on the Attribute Editor it should say Label1. If the alias is not set to Label1 you can set the alias by pressing Alt+A or RMouse on the widget and choose set alias. (note: if anitaliasing is enabled from File/Preferences then an alias will already be established for each widget.)

7 We are now ready to create some events for our application. To create events we will attach snippets of code to the widgets command operation. To bring up the command editor for a widget merely double click on the desired widget.

- Insert the following code into the command window for the "Hello World!" button:

```
$widget(Label1) configure -text "Hello World!"
```

- Insert the following code into the command window for the "Hello User!" button:

```
$widget(Label1) configure -text "Hello User!"
```

**Note: In 1.6.0, if widget command aliasing on, you can rewrite the above two lines of code as follows:

```
Label1 configure -text "Hello World!"
and
Label1 configure -text "Hello User!"
```

**Note: If widget command aliasing is on and you have multiple toplevels consider writing:

```
Toplevel1.Label1 configure -text "Hello World!"

instead of

$widget(Toplevel1,Label1) configure -text "Hello World!"
```

Now we are ready to go to Test mode from the Main window and run our application! Here are some screen shots:

Figure 28. Simple Example 1

Figure 29. Simple Example 2

Reference [35], [36] and [37].

APPENDIX C: CIAO PROLOG MANUAL

Ciao Prolog is a Prolog environment as we said, and so it has all the Prolog utilities necessaries to program in Prolog and more new libraries to make Prolog more powerful and with more communication with another languages like C, or Tcl.

In this project we tried to make that communication with Tcl better.

Here we will not try to explain the whole Ciao Prolog Utilities, just only to describe the main libraries we used to build our TRANSLATOR and other which can be interesting to understand that communication with Tcl.

So let's start describing the main part of Ciao Prolog

The Program development environment

This part documents the components of the basic Ciao program development environment. They include:

ciaoc:
the standalone compiler, which creates executables without having to enter the interactive top-level.

ciaosh:
(also invoked simply as ciao) is an interactive top-level shell, similar to the one found on most Prolog systems (with some enhancements).

debugger.pl:
a Byrd box-type debugger, similar to the one found on most Prolog systems (also with some enhancements, such as source-level debugging). This is not a standalone application, but is rather included in ciaosh, as is done in other Prolog systems. However, it is also *embeddable*, in the sense that it can be included as a library in executables, and activated dynamically and conditionally while such executables are running.

ciao-shell:
an interpreter/compiler for *Prolog scripts* (i.e., files containing Prolog code which run without needing explicit compilation).

ciao.el:
a *complete program development enviroment*, based on GNU emacs, with syntax coloring, direct access to all the tools described above (as well as the preprocessor and the documenter), atomatic location of errors, source-level debugging, context-sensitive access to on-line help/manuals, etc. The use of this environment is *very highly recommended*!
The Ciao program development environment also includes ciaopp, the preprocessor, and lpdoc, the documentation generator.

The ISO Prolog library (iso)

the *iso* package which provides to Ciao programs (most of) the ISO-Prolog functionality , including the *ISO-Prolog builtins* not covered by the basic library. All these predicates are loaded by default in user files

The Classic Prolog library (classic)

Some Ciao libraries which provide additional predicates and functionalities that, despite not being in the ISO standard, are present in many popular Prolog systems. This includes definite clause grammars (DCGs), "Quintus-style" internal database, list processing predicates, dictionaries, string processing, DEC-10 Prolog-style input/output, formatted output, dynamic loading of modules, activation of operators at run-time, etc.

Interfaces with other languages and systems

The following interfaces to/from Ciao Prolog are documented in this part:
External interface (e.g., to C).

- Socket interface.
- Tcl/tk interface.
- Web interface (http, html, xml, etc.);
- Persistent predicate databases (interface between the Prolog internal database and the external file system).
- SQL-like database interface (interface between the Prolog internal database and external SQL/ODBC systems).
- Java interface.
- Calling emacs from Prolog.

We will use the Tcl/Tk interface and will make some changes as well in it, that is the main thing in our project and so to understand how it work should be useful to explain a little bit more this library:

TCL/TK Library

The interaction between both languages is implemented as an interface between two processes, a Tcl/Tk process and a Prolog process. The approach allows programmers to program both in Tcl/Tk and Prolog.

Prolog - Tcl/Tk interface structure

The interface is made up of two parts: a Prolog part and a Tcl/Tk part. The Prolog part encodes the requests from a Prolog program and sends them to the Tcl/Tk part via a socket. The Tcl/Tk part receives from this socket and performs the actions included implied by the requests.

Prolog side of the Prolog - Tcl/Tk interface

The Prolog side receives the actions to perform in the Tcl/Tk side from the user program and sends them to the Tcl/Tk side through the socket connection. When the action is finished in the Tcl/Tk side, the result is returned to the user program, or the action fails if any problem occurs.

Tcl/Tk side of the Prolog - Tcl/Tk interface

The Tcl/Tk side waits for requests from the Prolog side, executes the Tcl/Tk code sent from the Prolog side. At the same time, the Tcl/Tk side handles the events and exceptions raised in the Tcl/Tk side, possibly passing on control to the Prolog side.

Library usage:

:- use_module(library(tcltk)).

Exports:

Predicates: tcl_new/1, tcl_eval/3, tcl_delete/1, tcl_event/3, tk_event_loop/1, tk_loop/1, tk_new/2, tk_next_event/2.

Regular Types: tclInterpreter/1, tclCommand/1.

Other modules used:

System library modules: tcltk/tcltk_low_level, write, strings, lists.

Figure 30. Library usage: tcltk

Documentation on exports (tcltk)

REGTYPE: `tclInterpreter/1`:

To use Tcl, you must create a *Tcl interpreter* object and send commands to it.

Usage: `tclInterpreter(I)`

- *Description:* I is a reference to a *Tcl* interpreter.

REGTYPE: tclCommand/1:

A *Tcl* command is specified as follows:

```
Command          --> Atom  { other than [] }
                       | Number
                       | chars(PrologString)
                       | write(Term)
                       | format(Fmt,Args)
                       | dq(Command)
                       | br(Command)
                       | sqb(Command)
                       | min(Command)
                       | ListOfCommands
   ListOfCommands  --> []
                       |[Command|ListOfCommands]
```

where:

Atom
denotes the printed representation of the atom.
Number
denotes their printed representations.
chars(PrologString)
denotes the string represented by *PrologString* (a list of character codes).
write(Term)
denotes the string that is printed by the corresponding built-in predicate.
format(Term)
denotes the string that is printed by the corresponding built-in predicate.
dq(Command)
denotes the string specified by *Command*, enclosed in double quotes.
br(Command)
denotes the string specified by *Command*, enclosed in braces.
sqb(Command)
denotes the string specified by *Command*, enclosed in square brackets.
min(Command)
denotes the string specified by *Command*, immediately preceded by a hyphen.
ListOfCommands
denotes the strings denoted by each element, separated by spaces.

Usage: `tclCommand(C)`

- *Description:* C is a *Tcl* command.

PREDICATE: tcl_new/1:

Usage: `tcl_new(-TclInterpreter)`

- *Description:* Creates a new interpreter, initializes it, and returns a handle to it in TclInterpreter.
- *Call and exit should be compatible with:* -TclInterpreter is a reference to a *Tcl* interpreter. (tcltk:tclInterpreter/1)

PREDICATE: tcl_eval/3:

Usage: `tcl_eval(+TclInterpreter,+Command,-Result)`

- *Description:* Evaluates the commands given in Command in the Tcl interpreter TclInterpreter. The result will be stored as a string in Result. If there is an error in *Command* an exception is raised. The error messages will be *Tcl Exception:* if the error is in the syntax of the tcltk code or *Prolog Exception:*, if the error is in the prolog term.
- *Call and exit should be compatible with:* +TclInterpreter is a reference to a *Tcl* interpreter. (tcltk:tclInterpreter/1) +Command is a *Tcl* command. (tcltk:tclCommand/1) -Result is a string (a list of character codes). (basic_props:string/1)

PREDICATE: tcl_delete/1:

Usage: `tcl_delete(+TclInterpreter)`

- *Description:* Given a handle to a Tcl interpreter in variable TclInterpreter, it deletes the interpreter from the system.
- *Call and exit should be compatible with:* `+TclInterpreter` is a reference to a *Tcl* interpreter. (tcltk:tclInterpreter/1)

PREDICATE: tcl_event/3:

Usage: `tcl_event(+TclInterpreter,+Command,-Events)`

- *Description:* Evaluates the commands given in Command in the Tcl interpreter whose handle is provided in TclInterpreter. Events is a list of terms stored from Tcl by *prolog_event*. Blocks until there is something on the event queue
- *Call and exit should be compatible with:* +TclInterpreter is a reference to a *Tcl* interpreter. (tcltk:tclInterpreter/1) +Command is a *Tcl* command. (tcltk:tclCommand/1) -Events is a list. (basic_props:list/1)

PREDICATE: tk_event_loop/1:

Usage: `tk_event_loop(+TclInterpreter)`

- *Description:* Waits for an event and executes the goal associated to it. Events are stored from Tcl with the *prolog* command. The unified term is sent to the Tcl interpreter in order to obtain the value of the tcl array of *prolog_variables*. If the term received does not have the form execute(Goal), the predicate silently exits. If the execution of Goal raises a Prolog error, the interpreter is deleted and an error message is given.
- *Call and exit should be compatible with:* `+TclInterpreter` is a reference to a *Tcl* interpreter. (tcltk:tclInterpreter/1)

PREDICATE: tk_loop/1:

Usage: `tk_loop(+TclInterpreter)`

- *Description:* Passes control to Tk until all windows are gone.
- *Call and exit should be compatible with:* `+TclInterpreter` is a reference to a *Tcl* interpreter. `(tcltk:tclInterpreter/1)`

PREDICATE: tk_new/2:

Usage: `tk_new(+Options,-TclInterpreter)`

- *Description:* Performs basic Tcl and Tk initialization and creates the main window of a Tk application.Options is a list of optional elements according to:

  name(+ApplicationName)

Sets the Tk main window title to ApplicationName. It is also used for communicating between Tcl/Tk applications via the Tcl *send* command. Default name is an empty string.

display(+Display)

Gives the name of the screen on which to create the main window. Default is normally determined by the DISPLAY environment variable.

file

Opens the sript file. Commands will not be read from standard input and the execution returns back to Prolog only after all windows (and the interpreter) have been deleted.

- *Call and exit should be compatible with:* `+Options` is a list. (basic_props:list/1) `-TclInterpreter` is a reference to a *Tcl* interpreter. (tcltk:tclInterpreter/1)

PREDICATE: tk_next_event/2:

Usage: `tk_next_event(+TclInterpreter,-Event)`

- *Description:* Processes events until there is at least one Prolog event associated with TclInterpreter. Event is the term correspondig to the head of a queue of events stored from Tcl with the *prolog_event* command.

- *Call and exit should be compatible with:* `+TclInterpreter` is a reference to a *Tcl* interpreter. (tcltk:tclInterpreter/1) -Event is a string (a list of character codes). (basic_props:string/1)

Low level interface library to TCL/TK

The tcltk_low_level library defines the low level interface used by the tcltk library. Essentially it includes all the code related directly to the handling of sockets and processes. This library should normally not be used directly by user programs, which use tcltk instead. On the other hand in some cases it may be useful to undertand how this library works in order to understand possible problems in programs that use the tcltk library.

Library usage:

```
:- use_module(library(tcltk_low_level)).
```

Exports:

Predicates: new_interp/1, new_interp/2, new_interp_file/2, tcltk/2, tcltk_raw_code/2, receive_result/2, send_term/2, receive_event/2, receive_list/2, receive_confirm/2, delete/1.

Other modules used:

System library modules: terms, sockets/sockets, system, write, read, strings, lists, format.

Figure 31. Library usage: tcltk_low_level

Documentation on exports (tcltk_low_level)

PREDICATE: new_interp/1:

Usage: new_interp(-TclInterpreter)

- *Description:* Creates two sockets to connect to the *wish* process, the term socket and the event socket, and opens a pipe to process *wish* in a new shell.
- *Call and exit should be compatible with:* `-TclInterpreter` is a reference to a *Tcl* interpreter. `(tcltk_low_level:tclInterpreter/1)`

PREDICATE: new_interp/2:

Usage: `new_interp(-TclInterpreter,+Options)`

- *Description:* Creates two sockets, the term socket and the event socket, and opens a pipe to process *wish* in a new shell invoked with the Options.
- *Call and exit should be compatible with:* `-TclInterpreter` is a reference to a *Tcl* interpreter. `(tcltk_low_level:tclInterpreter/1)` `+Options` is currently instantiated to an atom. `(term_typing:atom/1)`

PREDICATE: new_interp_file/2:

Usage: `new_interp_file(+FileName,-TclInterpreter)`

- *Description:* Creates two sockets, the term socket and the event socket, and opens a pipe to process *wish* in a new shell invoked with a FileName. FileName is treated as a name of a sript file
- *Call and exit should be compatible with:* `+FileName` is a string (a list of character codes). (basic_props:string/1) `-TclInterpreter` is a reference to a *Tcl* interpreter. `(tcltk_low_level:tclInterpreter/1)`

PREDICATE: tcltk/2:

Usage: `tcltk(+Code,+TclInterpreter)`

- *Description:* Sends the `Code` converted to string to the `TclInterpreter`
- *Call and exit should be compatible with:* `+Code` is a *Tcl* command. `(tcltk_low_level:tclCommand/1)` `+TclInterpreter` is a reference to a *Tcl* interpreter. (tcltk_low_level:tclInterpreter/1)

PREDICATE: tcltk_raw_code/2:

Usage: `tcltk_raw_code(+String,+TclInterpreter)`

- *Description:* Sends the tcltk code items of the Stream to the `TclInterpreter`

- *Call and exit should be compatible with:* +String is a string (a list of character codes). (basic_props:string/1) +TclInterpreter is a reference to a *Tcl* interpreter. (tcltk_low_level:tclInterpreter/1)

PREDICATE: receive_result/2:

Usage: receive_result(-Result,+TclInterpreter)

- *Description:* Receives the Result of the last *TclCommand* into the TclInterpreter. If the *TclCommand* is not correct the *wish* process is terminated and a message appears showing the error
- *Call and exit should be compatible with:* -Result is a string (a list of character codes). (basic_props:string/1) +TclInterpreter is a reference to a *Tcl* interpreter. (tcltk_low_level:tclInterpreter/1)

PREDICATE: send_term/2:

Usage: send_term(+String,+TclInterpreter)

- *Description:* Sends the goal executed to the TclInterpreter. String has the predicate with unified variables
- *Call and exit should be compatible with:* +String is a string (a list of character codes). (basic_props:string/1) +TclInterpreter is a reference to a *Tcl* interpreter. (tcltk_low_level:tclInterpreter/1)

PREDICATE: receive_event/2:

Usage: receive_event(-Event,+TclInterpreter)

- *Description:* Receives the Event from the event socket of the TclInterpreter.
- *Call and exit should be compatible with:* -Event is a list. (basic_props:list/1) +TclInterpreter is a reference to a *Tcl* interpreter. (tcltk_low_level:tclInterpreter/1)

PREDICATE: receive_list/2:

Usage: receive_list(-List,+TclInterpreter)

- *Description:* Receives the List from the event socket of the TclInterpreter.The List has all the predicates that have been inserted from Tcl/Tk with the command prolog_event. It is a list of terms.
- *Call and exit should be compatible with:* -List is a list. (basic_props:list/1) +TclInterpreter is a reference to a *Tcl* interpreter. (tcltk_low_level:tclInterpreter/1)

PREDICATE: receive_confirm/2:

Usage: receive_confirm(-String,+TclInterpreter)

- *Description:* Receives the String from the event socket of the TclInterpreter when a term inserted into the event queue is managed.
- *Call and exit should be compatible with:* -String is a string (a list of character codes). (basic_props:string/1) +TclInterpreter is a reference to a *Tcl* interpreter. (tcltk_low_level:tclInterpreter/1)

PREDICATE: delete/1:

Usage: delete(+TclInterpreter)

- *Description:* Terminates the *wish* process and closes the pipe, term socket and event socket. Deletes the interpreter TclInterpreter from the system
- *Call and exit should be compatible with:* +TclInterpreter is a reference to a *Tcl* interpreter. (tcltk_low_level:tclInterpreter/1)

Documentation on internals (tcltk_low_level)

PREDICATE: core/1:

Usage: core(+String)

- *Description:* core/1 is a set of facts which contain Strings to be sent to the Tcl/Tk interpreter on startup. They implement miscelaneous Tcl/Tk procedures which are used by the Tcl/Tk interface.
- *Call and exit should be compatible with:* +String is a string (a list of character codes). (basic_props:string/1)

Other information (tcltk_low_level**)**

Two sockets are created to connect the *TclInterpreter* and the prolog process: the *event_socket* and the *term_socket*. There are two global variables: *prolog_variables* and *terms*. The value of any of the variables in the goal that is bound to a term will be stored in the array prolog_variables with the variable name as index. The string which contains the printed representation of prolog *terms* is *Terms*. These are the Tcl/Tk procedures which implement the interface (the code is inside the tcltk_low_level library):

prolog
Sends to *term_socket* the predicate tcl_result which contains the goal to execute. Returns the string executes and the goal.

prolog_event
Adds the new *term* to the *terms* queue.

prolog_delete_event
Deletes the first *term* of the *terms* queue.

prolog_list_events
Sends all the *terms* of the *terms* queue by the *event_socket*. The last element will be *end_of_event_list*.

prolog_cmd
Receives as an argument the tcltk code. Evaluates the code and returns through the *term_socket* the term *tcl_error* if there was a mistake in the code or the predicate *tcl_result* with the result of the command executed. If the argument is *prolog* with a goal to execute, before finishing, the predicate evaluated by prolog is received. In order to get the value of the variables, predicates are compared using the *unify_term* procedure. Returs 0 when the sript runs without errors, and 1 if there is an error.

prolog_one_event
Receives as an argument the *term* which is associated with one of the tk events. Sends through the *event_socket* the *term* and waits the unificated *term* by prolog. After that it calls the *unify_term* procedure to obtain the value of the *prolog_variables*.

prolog_thread_event
Receives as an argument the *term* which is associated with one of the tk events. Sends through the *event_socket* the *term* and waits for the *term* unified by prolog. After that the *unify_term* procedure is called to obtain the value of the *prolog_variables*. In this case the *term_socket* is non blocking.

convert_variables
Receives as an argument a string which contains symbols that can't be sent by the sockets. This procedure deletes them from the input string and returns the new string.

unify_term
Receives as argument a prolog term.

Reference [19].

REFERENCES AND BIBLIOGRAPHY

For the theory and practice we used many information which came from different books as well as many different web pages which we consulted.

It is impossible to write here the complete way we did through Internet pages iduring the making of the project but we can write the main web pages where is essential knowledge about the different developing systems we used in the project.

We have to say as well that one of the main thing in our learning were the Tutorials we made in internet, these tutorials, which were wrote by different universities, helped us in understanding the main things of the Tcl/Tk and Vtcl programming because otherwise it could be impossible to learn the thing we learnt in the time we did.

BOOKS

All the books we consulted can be fount in the Ruc library and also one of them in Internet.

Prolog

1. *The art of Prolog : advanced programming techniques* / Leon Sterling, Ehud Shapiro ; with a foreword by David H. D. Warren. Cambridge, Mass. : MIT Press, 1986. - xxiii, 437 s. - (MIT Press series in logic programming)

2. *Prolog through examples* / by I. Kononenko and N. Lavra¥c. Wilmslow : Sigma, 1988. - 197 s.Bibliografi: s. 197

3. *Problem solving with Prolog* / John Stobo. London : Pitman, 1989

4. *Programming in Prolog* / W. F. Clocksin, C. S. Mellish. - 3. rev. and extended ed. Berlin : Springer-Verlag, c1987

5. *Logic and Prolog* / Richard Spencer-Smith. London : Harvester Wheatsheaf, 1991

Tcl/Tk

6. *Tcl and the Tk toolkit* / John K. Ousterhout. Reading, Mass. : Addison-Wesley, c1994. - xx, 458 s. - (Addison-Wesley professional computing series)

And the book we fount in Internet:

7. *Practical Programming in Tcl and Tk* / Brent Welch January 13,1995. Internet version.

INTERNET: WEB PAGES…

We divide also this section in different subject which we studied:

1. Prolog language
2. Ciao Prolog System
3. Tcl/Tk
4. Vtcl
5. Emacs

Prolog Language

8. http://clip.dia.fi.upm.es/~vocal/public_info/seminar_notes/node42.html

This is an introductory course on constraint logic programming made by

Manuel Carro

With input from: Manuel Hermenegildo
Francisco Bueno
Daniel Cabeza
M ª José García
Pedro López
Germán Puebla

In the UPM, (Technical University of Madrid), where we could find and reviewed the main things about Prolog language and logic programming which were so useful in the understanding the problem

9. http://www.trinc-prolog.com/doc/pl_lang.htm

This page contains a short introduction of the Prolog syntax and semantics.

10. http://www.phon.ucl.ac.uk/home/hans/courses/plinc101/part2.html

These notes are in part based on the SICStus Prolog 2.1 User's Manual by Mats Carlsson and Johan Widen.

11. http://www.visual-prolog.com/vipcoursematerial/language.ppt

Introduction to Prolog programming by Leo Schou-Jensen

Some tutorial we made about Prolog:

12. http://www.csupomona.edu/~jrfisher/www/prolog_tutorial/contents.html

Prolog syntax and many essentials of Prolog programming through the use of carefully chosen sample programs.

13. http://cs.wwc.edu/~cs_dept/KU/PR/Prolog.html

Prolog Tutorial. JA Robinson: A program is a theory (in some logic) and computation is deduction from the theory.

14. http://kti.ms.mff.cuni.cz/~bartak/prolog/

On-line guide to prolog programming. Very interesting and useful.

15. http://www.cs.may.ie/~jpower/Courses/PROLOG/

In this tutorial they just want to have a first shot at running Prolog...

We found it so practise and easy to follow.

16. http://cs.wwc.edu/Environment/Prolog.html

Small tutorial but very complete.

17. http://www.compapp.dcu.ie/~alex/LOGIC/start.html

Ciao Prolog System

18. http://clip.dia.fi.upm.es

Official page of the Clip group

19. http://clip.dia.fi.upm.es/Software/Ciao/

The official web page about ciao Prolog system where we found the different versions of the system, the Manual and other interesting FAQs.

Inside this huge page, we can find all the necessary to understand how Ciao Prolog works, as well as its authors and programming equipment and last news about its developing.

This is one of the most important source of knowledge we used in our project, because we have to remember that we used the Ciao Prolog developing system.

Tcl/Tk

20. http://www.tcl.tk/

The Tcl developer site.

The official web page about Tcl/Tk where we fount the version of the program which we worked with and also several FAQs and Links so useful for our project.

21. http://hegel.ittc.ukans.edu/topics/tcltk/

A web page with many information about Tcl/tk as well as some interesting tutorials.

22. http://tcl.sourceforge.net/

This is the *core development* home for Tcl) and the Tk toolkit, that collects information about many Tcl-related Source Forge projects. The sample TEA extension and Thread extension are part of the SF Tcl project

23. http://www.sun.com/960710/cover/tcl.html

An introducing to Tcl/Tk, with many information for beginners in programming with Tcl.

24. http://www.slac.stanford.edu/~raines/tkref.html

Quick reference.

25. http://www.wjduquette.com/tcl/

Simple and efficient page about Tcl.

26. http://www.dci.clrc.ac.uk/Publications/Cookbook/intro.html

Some Tutorials about Tcl we made in Internet were:

27. http://hegel.ittc.ukans.edu/topics/tcltk/tutorial-noplugin/

Small tutorial about Tcl/Tk very easy to follow and with many samples.

28. http://users.belgacom.net/bruno.champagne/tcl.html

Tcl tutorial very well done and structured where we fount the answer to many of our doubts.

29. http://www.beedub.com/book/2nd/tclintro.doc.html

Tcl Fundamentals. This tutorial describes the basic syntax rules for the Tcl scripting language. ... Tcl Commands.

30. http://jan.netcomp.monash.edu.au/ProgrammingUnix/tcl/tcl_tut.html

This short tutorial will concentrate on the most basic concepts to get you started as quickly as possible

31. http://www.surpac.com/refman/tcl/tutorial/

This is a tutorial on the basics of TCL. And also a tutorial with advanced examples of TCL.

32. http://lists.cye.com.au/pipermail/ice-linux/2002-March/000369.html

A TclTutor very practise when we wanted to find a quick answer for some questions.

Vtcl

33. http://www.neuron.com/stewart/vtcl/

The official page of the Vtcl project.

34. http://vtcl.sourceforge.net/

Page with all the FAQs and information about the Vtcl we needed to start working with it.

Some tutorial about Vtcl we used:

35. http://vtcl.sourceforge.net/tutorial.html

Tutorial from the sourceforge page, the most important page about Vtcl.

36. http://www.rjent.pair.com/wwwbook/vtcl/

Quite good tutorial with many examples and pictures about how to program in Vtcl.

37. http://osr5doc.ca.caldera.com:457/man/html.VTCL/CONTENTS.html

Good tutorial very well structured by topics, easy to follow and make.

Emacs

38. http://www.lib.uchicago.edu/keith/tcl-course/emacs-tutorial.html

A Tutorial Introduction to GNU Emacs. Introduction and History. ...

39. http://www.geek-girl.com/emacs/emacs.html

Emacs reference materials.

40. http://vertigo.hsrl.rutgers.edu/ug/emacs_tutorial.html

Another Emacs tutorial which explained us the hot keys and other important tools to use the Emacs.