First Mini-Project

Course of "Modelli e Sistemi Concorrenti"

```
 _________________________________
(              muCCS              )
(           version 2.0           )
 ---------------------------------
        o   ^__^
         o  (oo)\_______
            (__)\       )\/
                ||----w |
                ||     ||
```

ANDREA SIMONETTO
PAOLO PERFETTI
ODETA QORRI

Contents

Chapter 1

Specific Requirements

1.1 Initial requirements and design choices

This assignment is an effort to develop a program using a programming language that is based on a logic or symbolic approach (like Scheme, Lisp, Prolog ...) as detailed in our Professor's Guidelines. The program satisfies the following requirements:

- defines an appropriate structure for states and transitions;
- provides an appropriate representation of the SOS rules of the CCS;
- tests whether a transition is derivable from the SOS rules;
- given a state/process calculates all the transitions exiting from that state;
- given two states S1 and S2, verifies if S2 can reach S1;
- given a state S, determines its LTS associated.

Our choice of using Prolog as a programming language over any other language was based on the various advantages Prolog has as a Logic Programming Language, and over its compiler GNU-Prolog. First of all, we choose Prolog because of its essential incorporated mechanism of backtracking (the mechanism for finding multiple solutions). Second, we chose Prolog due to the fact that its syntax and semantic rules are based on Horn's clauses which facilitate the implementation of such rules.

Last but not least, we chose Prolog as it turned out to be a very interesting and challenging learning experience of a new programming language.

The idea behind the implementation of the entire project is based on the work of Gordon D. Plotkin and especially to his article “The Origins of Structural Operational Semantics” [Plo98]. This article guided our work, taught us new concepts on Structural Operational Semantics and also helped us understand the necessity of having syntax-directed rules to build this programming language.

In this paper we present Prolog, its compiler Gnu-Prolog, CCS Syntax, SOS Semantic Rules and the main project’s scheme. We also detail the tools used in building this project as well as the infrastructure that stands behind it all, based on all previous information collected.

Chapter 2

Introducing Prolog and its compiler GNU-Prolog

Prolog [Llo84], that stands for PROgramming in LOGic, was invented in the early seventies at the University Of Marseille and it it has been the first logic programming language that has been developed ever.

Unlike many other programming languages, Prolog is a declarative language in which writing an algorithm means executing a series of queries over a knowledge base of relations and facts previously defined. Given this particular programming paradigm, Prolog is often used to solve problems related to the database, symbolic mathematics, and language parsing applications.

Prolog is based on Horn's clause which states that a program consists of a set of procedures that are expressed in terms of Horn and are activated by an initial assertion. These terms can be of two types: facts and rules.

- Facts are used to state things that are unconditionally true in the domain of interest. Here is an illustration of a fact definition in Prolog:

```
parent(bob, mary).
parent(jane, mary).
parent(mary,peter).
parent(paul,peter).
```

- Rules state information that is conditionally true relating to the domain of interest and they are composed of two parts, the head and the body. An illustration of rules follows below:

```
anc(X,Y):- parent(X,Y).
anc(X,Z):- parent(X,Y), anc(Y,Z).
```

A general definition of a rule is as follows: if the body of the rule which appears on the right hand side of the clause is true, then the head of the rule, which is the left hand side of the clause, is true as well. This fundamental deduction step is what logicians call modus ponens.

- Queries ask Prolog's Inference Engine to interrogate its knowledge base, which is composed of the set of facts and rules. Based on the relations defined in the previous examples, we can state the following valid queries:

```
?- parent(bob, mary)
yes

?- parent (X,peter)
X = mary;
X = paul;
yes

?- parent (X,paul)
no

?- anc(bob, peter)
yes
```

The procedure used by Prolog is the principle of resolution: given a query, the inference engine attempts to find a resolution while trying at the same time to invalidate the negated query. If the negated query can be proven false, i.e. a solution for all free variables is found in order to make the union of the clauses true and at the same time, the single set of the negated query is found false, it follows that the original query, with the solution found, is a logical consequence of the program.

2.1 Prolog Syntax

Prolog contains only one type of data which is called a term. There are four kinds of terms: atoms, numbers, variables, and compound terms (or structures). Atoms and numbers are aggregated together under the heading constants, while constants and variables together make up the simple terms of Prolog.

An atom name can be either:
- a string of characters made up of upper-case letters, lower-case letters, digits, and the underscore character, that begins with a lower-case letter or

- an arbitrary sequence of character enclosed in single quotes.

Numbers can be floating point numbers or integers.

A variable is denoted by a string of upper-case letters, lower-case letters, digits and underscore characters that start either with an upper-case letter or with underscore.

Compound terms are build out of a functor followed by a sequence of arguments which are put in ordinary brackets, separated by commas and placed after the functor. The functor must be an atom; that is, variables cannot be used as functors. On the other hand, arguments can be any kind of term.

2.2 GNU-Prolog

The main tool used for this project is the GNU-Prolog [Dia07]. It is a Prolog compiler with constraint solving over finite domains developed by Daniel Diaz. The choice of this compiler depends upon the fact that it can produce native code to be fast and rather efficient instead of only interpreting Prolog instuctions. Another interesting feature of this compiler is that its executables are small; indeed, the code of most unused built-in predicates is not included in the executables at link-time.

In the interactive mode GNU Prolog top-level consists on a classical read-execute-write loop that also allows for re-executions and debugging. Instead, the GNU Prolog compiler is built on the Warren Abstract Machine [War83], compiling high level instructions in a WAM file, machine indipendent bytecode, which is later easily translatable in native code.

Furthermore, GNU Prolog provides an API toward C programming language, allowing to interrogate Prolog's engine from external programs written in this language. This interface provides different functions to convert data and structures between the environments (*Mk_Atom, Mk_Compund, Mk_Variable, Rd_Atom,...*), to indagate the types of terms (*Blt_Atomic, Blt_List,...*) and to submit queries (*Pl_Query_Call, Pl_Query_End,...*).

Chapter 3

Syntax and Semantic of CCS

3.1 CCS Syntax

According to Milner's definition of the Calculus of Communicating Systems, the collection $\mathcal{P}$ of valid CCS expressions is given by the following grammar:

P,Q ::= K $\Big| \ \alpha.P \ \Big| \ P + Q \ \Big| \ P \mid Q \ \Big| \ P\backslash L \ \Big| \ P[f]$
where

- K is a process name in $\mathcal{K}$,
- α is an action in Act,
- I is a possibly infinite index set,
- f : *Act* $\rightarrow$ *Act* is a *relabelling function* satisfying the constraints

$$f(\tau) = \tau \ , \ f(\overline{a}) = \overline{f(a)} \text{ for each label a, and}$$

- L is a set of labels from L.

In order to make the set of CCS expressions easily recognizable by our program and to avoid the re-invention of the wheel of language parsing we have chosen to use well known and standard tools like Flex (The Fast Lexical Analyzer) and Bison (GNU parser generator). The first one's duty is to parse input and generate the tokens used by the second one to validate well formed phrases according to a previously defined grammar. So we need now to translate CCS syntax in Backus-Naur Form as requested by Flex/Bison lexer and parser.

Backus-Naur Form (BNF) is a metasyntax used to express context-free grammars: that is, a formal way to describe formal languages. This context free grammar is used to define the syntax of a programming language by using two sets of rules: i.e., lexical rules and syntactic rules.

The lexical rules are defined in the configuration files of Flex (ccs.l). Below we show the main lexical rules defined to recognize the necessary tokens.

```
PID ::= [A-Z][a-zA-Z0-9_]*
OBS_ACT ::= [a-z][a-zA-Z0-9_]*
NIL ::= nil | 0
TAU ::= tau
```

On the other hand, the syntactic rules are defined (using tokens provided by Flex) in the configuration of Bison as below.

```
ProcDef ::= PID OP_DEF Proc
Proc ::= (Proc) | NIL | PID | TAU.Proc | OBS_ACT.Proc |
       | ~OBS_ACT.Proc | Proc + Proc | Proc|Proc |
       | Proc\{ObsActList} | (Proc)[RelList]

ObsActList ::= OBS_ACT | OBS_ACT,ObsActList
Rel ::=  OBS_ACT\OBS_ACT
RelList ::= Rel | Rel,RelList
```

For each rule defined in Bison, there can be some actions (written in C) that are executed when that rule is matched. Prolog terms are built up exploiting these actions during the parsing process by using GNU Prolog API for C language. For example:

```
Proc + Proc {
      PlTerm args[2];
      args[0] = $1;
      args[1] = $3;
      $$ = Mk_Compound(Create_Atom("sum"), 2, args);
}
```

3.2 SOS Semantic

In this section we will describe the rules of Act, Sum, Com1, Com2, Com3, Res, Rel, Con [AIL07] aand also show the corresponding translation in Prolog for each one of them. In the file *sos.pl*, the choice of using Prolog to implement Structural Operational Semantics is made obvious by the elegance and semplicity in which each rule is trivially translated in Prolog without the need of any other complex semantic definitions.

Let's concentrate on each of these rules.

- ACT

$$\text{ACT} \qquad \frac{}{\alpha.P \xrightarrow{\alpha} P}$$

  To emulate the Act axiom (a rule whithout premises) *pre(A,P)* functor is needed to express CCS *prefixing*. In this way, the Act rule is translated in Prolog in the following way:

```
move(pre(A,P), P, A).
```

  that means that process *pre(A,P)* can move to P with action A.

- SUM

$$\text{SUM1} \quad \frac{P \xrightarrow{\alpha} P'}{P+Q \xrightarrow{\alpha} P'} \qquad \text{SUM2} \quad \frac{Q \xrightarrow{\alpha} Q'}{P+Q \xrightarrow{\alpha} Q'}$$

  For the translation of this rule we had to write two predicates SUM1 and SUM2.

```
move(sum(P1,_), P1_1, A) :- move(P1, P1_1, A).
move(sum(_,P2), P2_1, A) :- move(P2, P2_1, A).
```

  These rules express the possibility to choose which process must prosecute in its computation.

- COM

$$\text{COM1} \quad \frac{P \xrightarrow{\alpha} P'}{P \mid Q \xrightarrow{\alpha} P' \mid Q}$$

$$\text{COM2} \quad \frac{Q \xrightarrow{\alpha} Q'}{P \mid Q \xrightarrow{\alpha} P \mid Q'}$$

$$\text{COM3} \quad \frac{P \xrightarrow{\alpha} P' \quad Q \xrightarrow{\overline{\alpha}} Q'}{P \mid Q \xrightarrow{\tau} P' \mid Q'}$$

The Com transition describes agent composition, which is the basic operator for concurrency. Agent composition represents how agents behave, both autonomously (COM1, COM2) and interactively (COM3). A fundamental activity within CCS is the handshake, which is a successful simultaneous communication between two agents. In order for a handshake to occur, two agents must simultaneously execute identical immediate actions, one of which is a co-action of the other. In COM1 and COM2, if either agent on the left and right of the composition operator "|" can produce a single transition, the whole expression makes the transition. These rules are defined by us in this way:

```
move(com(P1, P2), com(P1_1, P2), A) :- move(P1, P1_1, A).
move(com(P1, P2), com(P1, P2_1), A) :- move(P2, P2_1, A).
```

If both expressions make complementary transitions, a handshake results. This is represented by the τ-action and is defined in this way:

```
move(com(P1, P2), com(P1_1, P2_1), A):-
   move(P1, P1_1, A1), move(P2, P2_1, A2),
   can_com(A1, A2), A=tau.
```

where *can_com(A1, A2)* express the fact the *A1* and *A2* are opposite to each other.

- RES

$$\text{RES} \quad \frac{P \overset{\alpha}{\rightarrow} P'}{P \backslash L \overset{\alpha}{\rightarrow} P' \backslash L} \text{ where } \alpha, \overline{\alpha} \notin L$$

Restriction removes the specified actions in set L from being observed externally. This is useful to reserve some action to the communication of the processes inside the scope of restriction.

```
move(res(P1,L), res(P2,L), A) :-
          move(P1, P2, A), not_in(A, L).
```

where L is the list of the restricted actions.

- REL

$$\text{REL} \quad \frac{P \overset{\alpha}{\rightarrow} P'}{P[f] \overset{f(\alpha)}{\rightarrow} P'[f]}$$

$f : L \rightarrow L$ is a relabelling function that renames observable actions. This means that each instance of α in the scope of f will be substitued with $f(\alpha)$.

```
move(relabel(P1, L), relabel(P1_1, L), A) :-
          move(P1, P1_1, X), subst_act_member(A, X, L).
move(relabel(P1, L), relabel(P1_1, L), A) :-
          move(P1, P1_1, A), not_subst_act(A, L).
```

where *subst_act_member(A, X, L)* and *not_subst_act(A, L)* are auxiliary functions.

- *subst_act_member(A, X, L)* verifies if the substitution X/A is present in the substitution list L.

- *not_subst_act(A, L)* verifies if A is not present in the substitution list L.

- CON

$$\text{CON} \quad \frac{P \overset{\alpha}{\rightarrow} P'}{K \overset{\alpha}{\rightarrow} P'} \text{ where } K \overset{def}{=} P$$

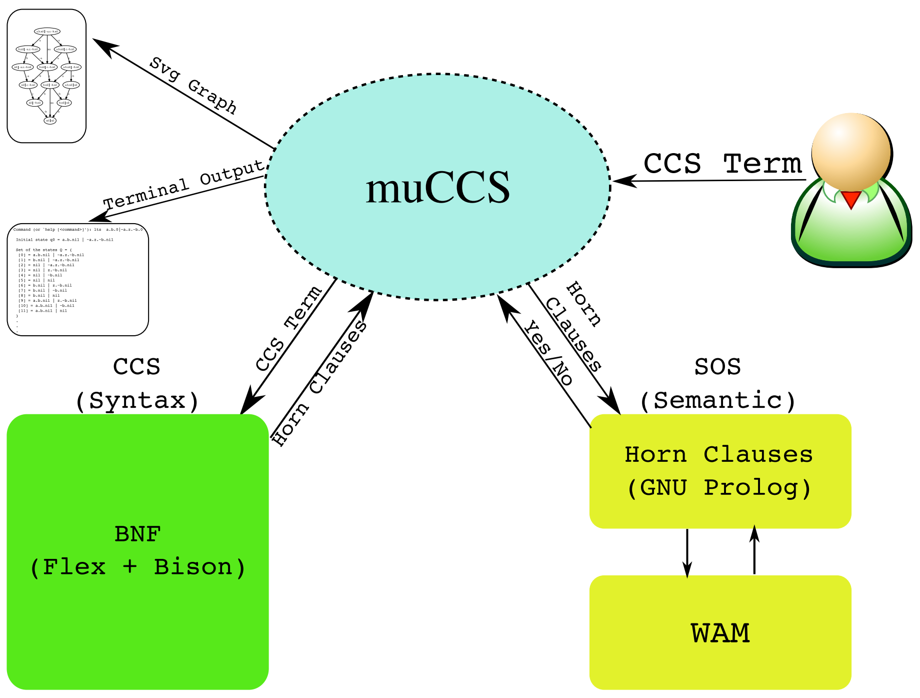
The Con rule defines agent constants, and it is the basic means for creating recursive agent definitions. This behaviour is obtained with the following rule:

```
move(proc(K), P1, A) :-
        proc_def(K, P), move(P, P1, A).
```

where *proc_def*(K, P) is defined using the GNU-Prolog Built-in Predicate **dynamic** that permits keeping track of process definition.

Chapter 4

Main project scheme



```
Command (or `help [<command>]'): lts  a.b.0|-a.z.-b.0

 Initial state q0 = a.b.nil | ~a.z.~b.nil

 Set of the states Q = {
  [0] = a.b.nil | -a.z.-b.nil
  [1] = b.nil | -a.z.-b.nil
  [2] = nil | -a.z.-b.nil
  [3] = nil | z.-b.nil
  [4] = nil | ~b.nil
  [5] = nil | nil
  [6] = b.nil | z.~b.nil
  [7] = b.nil | ~b.nil
  [8] = b.nil | nil
  [9] = a.b.nil | z.-b.nil
  [10] = a.b.nil | -b.nil
  [11] = a.b.nil | nil
 }
 .
 .
 .
```

The picture above is an illustration of the main project scheme. There are two basilar concepts that stand in this scheme: Syntax and Semantic. As far as Syntax is concerned, we use Flex and Bison in order to generate BNF grammar. On the other hand, CCS semantics can also be introduced through an implementation of Horn Clauses with the inferred engine GNU-Prolog. The use of C programming language helped us implement the queries and generate the final output.

Some examples of muccs computation are presented below:

Given the following BNF grammar:

$\text{Proc} ::= \text{Act.Proc} \;\Big|\; \text{Proc} + \text{Proc} \;\Big|\; \text{Proc} \mid \text{Proc} \;\Big|\; \ldots$
$\text{Proc} ::= 0$

a.b.0 will be transformed in the CCS syntax in this way:

- **CCS Syntax (muCCS input):** *a.b.0*
- **Transitions Horn-Clauses (Parser's output):** *pre(a, pre(b,0)).*
- **First Order Logic (SOS rule):** $|{-}\alpha.P \overset{\alpha}{\to} P$
- **GNU-Prolog (SOS rule):**

```
move(pre(A,X),X,A).
```

- **Prolog Query (Output of the inference engine):**

```
?- move(pre(a, pre(b,0)), X, A).
X = pre(b,0).
A = a.
```

Here we present another example:

- **CCS Syntax (muCCS output):** *a.0 + b.0*
- **Transitions Horn-Clauses (Parser output):** *sum(pre(a,0), pre(b,0)).*
- **First Order Logic (SOS rule):** $\dfrac{P_1 \overset{\alpha}{\to} P_1'}{P_1 + P_2 \overset{\alpha}{\to} P_1'}$
- **GProlog (SOS rule):**

```
move(sum(P1,_), P1_1, A) := move(P1, P1_1, A)).
```

- **Prolog Query (Output of the inference engine):**

```
?- move(sum(pre(a,0), pre(b,0)), X, A).
X = 0.
A = a.
```

The C programming language interacts with the inference engine with queries like those in the previous examples and manipulates the results in order to obtain the user's final output.

Chapter 5

Conclusions

The aim of this project was the implementation of SOS rules in a specific desired programming language. The implementation of this project was based on pre-existing research work on Structural Operational Semantics and the application of such research in combination with different concepts like the combination of Flex and Bison to provide CCS syntax, the use of GNU-Prolog for the translation of SOS rules and as an inferred engine and even the use of C programming language to implement the queries.

Above all, this project involved the use of C programming language to unify all of the steps above to obtain the desired final result.

This research paper involved both research work and practical application of various theories and programming languages that in the end produced interesting results and generated the realization of all required items set forth at the beginning of this academic work.

Appendix A

muccs-2.0 Folder

A.1 Compilation and execution of the project

In order to view the execution of this project some requirements are needed:

1. Have GNU-Prolog $\geq$ 1.3.0 installed in your machine.
   (It can be downloaded at: http://www.gprolog.org/#download).

2. Have graphviz installed in your machine.
   (It can be download at: http://www.graphviz.org/Download.php).

3. After downloading GNU-Prolog, the following steps are necessary:

- cd muccs-2.0

- make

- ./muccs

When the prompt is visualized, one can execute different examples based on the operation that is desired to be executed. This project was tested in Ercolani Linux Laboratory and it compiles and executes perfectly.

A.2 Content and description of the muccs-2.0 folder

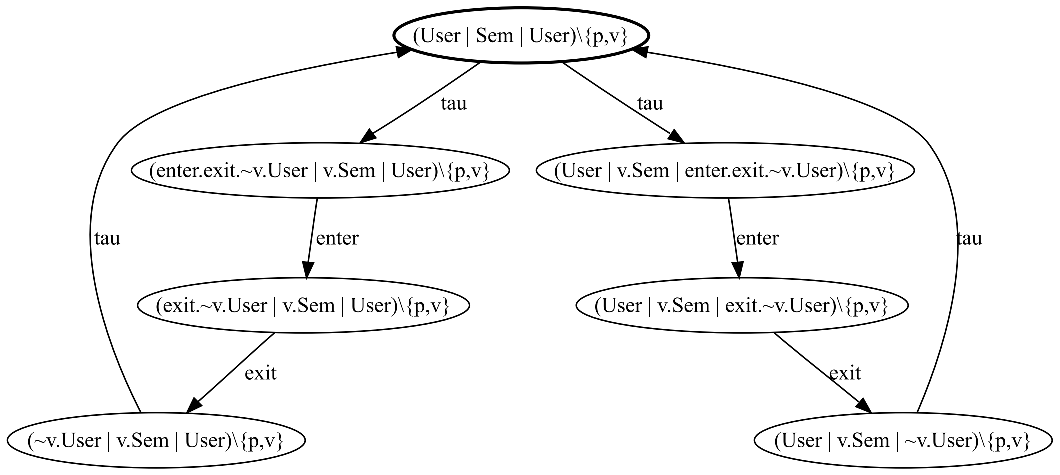
The following files are included in the muccs-2.0 folder. A brief description of each file is given below.

- **ccs.y/ccs.l** - Input of Bison and Flex for the CCS syntax; creates a parser that produces Horn clauses guided syntax directed.

- **sos.pl** - Defines a proper rappresentation of the SOS rules in Prolog.
- **muccs.c** - Implements the main function, prints the prompt of command lines, and other main functionalities.
- **libccs.c** - Implements the features of the program interfacing with the parser and with the inferred machine.
- **test** - Tests different examples and also includes LTS graphs generated with the package graphviz.

A.3 LTS graph

One of the most interesting points that we noticed during the work was that of the implementation of an LTS. We also created tests in which we generated the output of the LTS command and also presented the LTS graph with the help of the graphviz package. Below, we present one of the examples of an LTS graph:


(User | Sem | User)\{p,v}
tau
tau
(enter.exit.~v.User | v.Sem | User)\{p,v}
(User | v.Sem | enter.exit.~v.User)\{p,v}
tau
enter
enter
tau
(exit.~v.User | v.Sem | User)\{p,v}
(User | v.Sem | exit.~v.User)\{p,v}
exit
exit
(~v.User | v.Sem | User)\{p,v}
(User | v.Sem | ~v.User)\{p,v}

where

```
agent User = ~p.enter.exit.~v.User
agent Sem = p.v.Sem
```

Bibliography

[AIL07] Aceto L., Ingólfsdóttir A., Larsen G.K., Srba J., *Reactive Systems*, Cambridge University Press, August 2007.

[BBS01] Blackburn P., Bos J., Striegnitz K., *Learn Prolog now*, 2001, website: http://www.coli.uni-sb.de/kris/learn-prolog-now.

[Bur69] Burstall M.R., *Providing Properties of Programs by Structural Induction*, The Computer Journal, Vol.12, No. 1, pp. 41-48, 1969.

[Dia07] Diaz D., *GNU-Prolog Manual*, Edition 1.8, for GNU Prolog version 1.3.0, January 2007, website: http://www.gprolog.org/manual/html_node/.

[KLG84] Kahn G., Lang B., Gouge V., *The Mentor experience* Interactive Programming Environment, pp.128-140, New York: McGraw-Hill, 1984.

[Llo84] Lloyd J.W., *Foundations of logic programming*, Springer-Verlag, Berlin 1984.

[Plo98] Plotkin D.G., *The Origins of Structural Operational Semantics*, Laboratory for Foundations of Computer Science, School of Informatics, University of Edinburgh, King's Buildings, Edinburgh EH9 3JZ, Scotland, website: http://citeseer.ist.psu.edu/old/plotkin03origins.html.

[War83] Warren D.H.D., *An abstract Prolog instruction set*, Technical Note 309, SRI International, Menlo Park, CA, October 1983.