Implementation of a Planning System Using GCLA

by

Johan Palmkvist
Implementation of a Planning System Using GCLA

by

Johan Palmkvist

Swedish Institute of Computer Science
Box 1263, S-164 28 Kista, Sweden
email: johanp@sics.se
phone: +46 8 752 15 00

Royal Institute of Technology
Stockholm, Sweden
email: johanp@nada.kth.se

Abstract

This report describes a planning system implemented in GCLA. A definition of planning in the house world, which is a superset of STRIPS-like planning, is given, as well as a short introduction to the programming language GCLA. The blocks world and STRIPS-like planning is described, followed by a comparison between STRIPS-like planning and planning in the house world.

Keywords

GCLA, Logic Programming, Planning.
1. Introduction

The report describes the implementation of a planning system for a construction site. The planning system is able to produce a plan, given a specified state of a house.

GCLA, a new programming language developed at SICS, especially suited for hypothetical reasoning, is used as implementation language. The language is also valued as an implementation language of a knowledge based system. GCLA is best regarded as a logic programming language, with some properties commonly found among functional languages.

The domain is described in section 3. Each method in the domain is usually divided into three documents. In the document production data, we found operation plans, which we implemented in our data base. E.g. the operation plan to build a house can be described as the rule:

\[
\begin{align*}
\text{house} &::= \text{pre(signed_contract)} \\
&\quad \text{act(floor, wall, roof)} \\
&\quad \text{time}(1) \\
&\quad \text{post(house)}
\end{align*}
\]

where \text{pre} is a precondition, \text{act} is the needed activities, \text{time} is the time needed to build a house, and \text{post} tells us what is true after the house is built. The above rule can be implemented in GCLA as the clause:

\[
\begin{align*}
\text{house :-} \\
&\quad \text{pre(signed_contract),} \\
&\quad \text{act([floor, wall, roof]),} \\
&\quad \text{time}(1), \\
&\quad \text{post(house)}. \\
\end{align*}
\]

In STRIPS-like planning, as described in section 6, there is a given instruction-set, and the instructions can only execute in sequence. When planning in the house world we must first find an instruction-set, and then we use it for planning. The instruction-set can also be more complex than in STRIPS-like planning, since the instructions can, or may be, executed in parallel, or in sequence, or a combination of both.

In the discussion in section 7, we state that STRIPS-like planning is a true subset of planning in the house world, i.e. planning in the house world is more general form of planning than STRIPS-like planning.

We implemented our planning system in GCLA, and we found the language well suited to our needs. The technique for converting a rule, as described in section 5.1, in a desired data base to the corresponding GCLA clause in a real data base, is straight forward and self explanatory.
The author would like to thank Martin Aronsson at SICS, and Carl-Gustav Jansson at Stockholm University.

2. The Problem

2.1. Task

The task is to implement a planning system for a construction site. The system should be able to produce a plan, given a specified state of a house.

The task is also to evaluate GCLA as an implementation language of a knowledge based system.

2.2. Background of the Choice of Implementation Language

A new programming language, GCLA, Generalized Horn Clauses, has been developed within the scope of the project "Foundations of Programming for Knowledge Based Systems". The language is based on the theory of partial inductive definitions, also developed at SICS. GCLA is best understood as belonging to the family of logic programming languages, but with some properties usually found among functional languages. The language is designed to be used for development of reasoning systems, and to handle problems which include hypothetical questions and non-monotonic reasoning. A number of small test programs have been run on the current interpreter. However, no larger applications have been written in GCLA.
3. The Language GCLA

GCLA [A1, A2] is a new programming language, especially suited for hypothetical reasoning. The language is best regarded as a logic programming language, with some properties commonly found among functional languages.

In GCLA, a program is not looked upon as a set of true facts and implications from which conclusions are drawn using a specific computation rule (e.g. SLD resolution as in Prolog), but rather as a definition of rules determining the possible inference. In a sense the program defines the "systems mind", meaning that the inference rules of the logic are given by the program. To execute a program, a query $c$ is posed, asking whether there is a substitution $\sigma$ such that $c \sigma$ holds according to the logic defined by the program.

The goal $c$ has the form $\Gamma \vdash c$, where $\Gamma$ is a list of assumptions, and $c$ is the conclusion drawn from the assumption $\Gamma$.

The clauses in the program may be used both to form the introduction rules, applicable to the consequence of the goal (to the right of $\vdash$), and to form the elimination rules, applicable to the assumptions of a goal (to the left of $\vdash$). The introduction rules of the logic (defined by the GCLA program) are the clauses as they stand. The elimination rules are formed by the largest set of clauses in the program whose head is unifiable with the given assumption. Consider the program:

$$
\begin{align*}
p & \leftarrow q, \\
r & \leftarrow q, \\
q & \leftarrow s.
\end{align*}
$$

If we ask GCLA if $r$ follows from $p$

$$p \vdash r,$$

the interpreter will answer yes. The clause $r \leftarrow q$ is used as an introduction rule giving the goal $p \vdash q$, and the clause $p \leftarrow q$ is used as a elimination rule, giving the goal $q \vdash q$, which is trivially provable.

There is an essential conceptual difference between assumptions and definitional clauses. Consider the program:

$$w \leftarrow q.$$
The query $w \vdash q$ succeeds, because using the rule $w :\neg q$ as an elimination rule gives us the goal $q \vdash q$, which succeeds. But, if the program is changed to:

\[
\begin{align*}
w & :\neg \text{true}. \\
w & :\neg q.
\end{align*}
\]

(where $w :\neg \text{true}$ stands for a rule which is unconditional true), i.e. asserting the fact $w$, the goal $\vdash q$ fails since there is no introduction rule for $q$.

For a more detailed description of the language GCLA and its interpreter we refer to [A1, A2].

4. **Description of the Domain**

The orderer has a design of the house and invites building companies to submit offers.

The building company quantifies the design and writes a quantity catalogue, which includes prices for all parts and activities, and the needed number of man-hours. The total price is calculated from the quantity catalogue. The building company makes one main time plan and some detailed time plans from the quantity catalogue.

The orderer then decides which company who gets the order.

4.1. **Methods and Data**

To give a foundation for calculation, planning and preparation of construction work for houses, and to make work instructions easy, the Swedish Building Association has decided to publish Method and Data descriptions [M1].

The methods described in [M1] are uniformly designed and arranged to facilitate the purpose they serve. For these methods to be of any valuable use, the concepts used have to be defined clearly.
Usually each method is divided into three documents: production data, time formula and working description. Both the production data and time formula documents contain time data, but with different time resolution. In exceptional cases, i.e. when the production data has the same resolution as the time formulas, the production data and the time formula document can be merged into one document. Consequently, calculation of the time needed to carry out the method, can be done with data, either from the production data or time formula document. The working description document describes the method and its steps.

4.2. Production Data

This document contains time data concerning planning and calculation, and usually has ten headlines: material, equipment, method, operation plan, team size, method time, production time, operation description, time foundations and calculation example.

We will only address headlines that concerns our planning system.

• Method
The method is illustrated with representative pictures of the work, in some cases with an explanation text. Usually, if more than one method is illustrated, they are represented with the characters A, B and C. Methods which are equal from time aspect, are represented with the same character. A more detailed description of the method can be found in the working description document.

![Method A. Casting with 1 man.](image)

![Method B. Casting with 2 men.](image)

*Fig. 4.1. Example of two alternative methods.*

• Operation Plan
The succession of the operations are illustrated by the operation plan (fig. 4.2). The studied operations are represented with large arrows. The length of the arrows corresponds, when possible, to the time needed to carry out the different operations.
• Method Time
Method time is the total time, measured in man-hours, needed to carry out the operations and "events" depending on the chosen method. The method time is calculated from standard times, given in tables and diagrams. The standard time is given in man-hours.

4.3. Definitions from Method and Data

• Activity
An activity is something we do, e.g. to build a wall. Sometimes an activity can be divided into smaller activities, subactivities. For instance, to build a wall (fig. 4.3) may be divided into three subactivities: mouldfound wall, reinforcement wall, casting wall.

• Structure Plan
A structure plan (fig. 4.4) describes the succession of, and dependencies between, the activities. It should be evident in the structure plan if it is possible to perform two or more activities at the same time, i.e. in parallel.

<table>
<thead>
<tr>
<th>Mouldfound wall</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Reinforcement wall</td>
<td></td>
</tr>
<tr>
<td>Casting wall</td>
<td></td>
</tr>
</tbody>
</table>

Fig. 4.4. An example of a structure plan.
• Time Plan
A time plan (fig. 4.5) is a structure plan with a time scale. It is described in the time plan when an activity is planned to begin, and when to end. A time plan has a specified resolution, for example weeks.

Fig. 4.5. An example of a time plan.

5. Planning in the House World

In the blocks world, as described in section 6, we have a specified state and we are looking for a transition that will allow us to reach a desired state. Both states are rather "physical" states. On the contrary, in the house world we do not have a specified state initially. Instead we have a specification of the desired house, and are looking for a transition in order to turn the specification of the house into a real house. Thus, we are looking for the activities needed to build the specified house.

5.1. The Database

Consider an activity house, we can define house as the three subactivities: floor, wall and roof. This can be formulated in the rule:

    house ::= floor, wall, roof

This rule does not say anything about the succession of, or dependency between, the subactivities.

In a similar way as the first definition of house, we can define the subactivity roof, either as the activity tie beams, or the activity roof truss.

    roof ::= tie beams
    roof ::= roof truss
For each activity, which can be divided into one or more subactivities, there should be such a rule defining the activities subactivities. For those activities which cannot be divided into subactivities, there should be a rule defining the atomic activity.

\[
\text{house-painting ::= house-painting}
\]

The activity may have preconditions, which must be true before we execute the activity. For example, a contract have to be signed before we start to build the house:

\[
\text{house ::= pre(signed_contract)}
\begin{align*}
& \text{act(floor, wall, roof)} \\
& \text{post(house)}
\end{align*}
\]

where \text{pre} is the precondition, and \text{post} tells us what is true after the activity \text{house} has been executed.

Each activity consumes from one or more resources. The only resource in our model is time. The activity \text{house} is now defined as:

\[
\text{house ::= pre(signed_contract)}
\begin{align*}
& \text{act(floor, wall, roof)} \\
& \text{time(1)} \\
& \text{post(house)}
\end{align*}
\]

The general form of a rule in our database will be:

\[
A ::= \text{pre}(P) \\
& \text{act}(A') \\
& \text{time}(T) \\
& \text{post}(A)
\]

where the activity \text{A} is defined as the activity \text{A'}, with the precondition \text{P}, the postcondition \text{A} and consuming the time \text{T}. The precondition \text{P} may be empty, i.e. always true.

The technique to convert the rule to GCLA code is straightforward and self-explanatory. The GCLA clause corresponding to the previous rule is:

\[
\text{house :-}
\begin{align*}
& \text{pre(signed_contract),} \\
& \text{act(floor, wall, roof)}, \\
& \text{time(1)}, \\
& \text{post(house)}.
\end{align*}
\]

5.2. Specifying The House

Somehow we want to be able to specify the house which we are planning. One solution is to use parameters in the GCLA clauses.
house(floor(F), wall(W), roof(R)) :-
  pre(signed_contract),
  act([floor(F), wall(W), roof(R)]),
  time(1),
  post(house).

F, W, R are variables specifying the floor, the wall and the roof.

In order to get the planning system working as intended, we must have a substitution schema.

If we have a plan P:

\[ P = [\text{Pre}, \text{act}(A), T, \text{Post}] \]

we want to be able to substitute the activity A with the plan P', which is the plan for the activity A. After the first substitution we want the plan P to be:

\[ P = [\text{Pre}, \text{act}(\text{plan}(P'), \text{act}(A'), T', \text{Post'})), T, \text{Post}] \]

The GCLA clause for which the substitution schema behaves like above, can be written as:

\[ \text{plan}([\text{Pre}, \text{act}(A), T, \text{Post}]) :- \\
  (\text{act}(A) \rightarrow \text{plan}(\text{Plan})) \rightarrow \\
  '\&\text{axiom}'(\text{plan}([\text{Pre}, \text{act}(\text{Plan}), T, \text{Post}])). \]

The GCLA clause defining house is now written as:

\[ \text{house(floor}(F), \text{wall}(W), \text{roof}(R)) :- \\
  \text{plan}([\text{pre}(\text{signed}\_\text{contract}), \\
  \text{act}([\text{floor}(F), \text{wall}(W), \text{roof}(R)]), \\
  \text{time}(1), \\
  \text{post}(\text{house})]). \]

### 5.3. Finding Activities

In order to find the necessary activities to build a specified house, we pose the question:

\[ \text{house}(F, W, R) \rightarrow \text{plan}(P). \]

to the planning system. In this goal, the house is not specified in detail. We leave it to the system to propose a house and a list of activities satisfying the given specification.
Because GCLA tries to use the introduction rules first [see rule program-right [A1]], the goal will not work as intended. The solution is to guide GCLA with the primitive no п_right, which means that GCLA does not try to use the introduction rule during execution of the goal. The systems answer to the goal:

\[
\text{house}(F, W, R) \mid \text{plan}(P)) : \text{no п_right.}
\]

will be:

\[
\begin{align*}
F &= \text{floor}(_{522}) \\
W &= \text{wall}(_{523}) \\
R &= \text{roof}(_{524}) \\
P &= \text{pre}(_{\text{signed_contract}}) : \text{p_right,} \\
    &\quad \text{act}((\text{floor}(_{522}),) \\
    &\quad\quad \text{wall}(_{523}), \\
    &\quad\quad \text{roof}(_{524})), \\
    &\quad \text{time}(1), \\
    &\quad \text{post}(_{\text{house}}) : \text{p_right}
\end{align*}
\]

where _522 etc. are uninstantiated variables.

In this way the plan $P$, and the house, gets more and more specified in a "lazy-evaluation" fashion.
5.4. Turning Activities to a Plan

The activity-list \( P \) which is described above, is not guaranteed to be a sound solution. All the activities preconditions and postconditions must be checked in order to find all dependencies.

First the preconditions of the activity is tested. If they are true, then we try to do the activity. The activity may consist of subactivities, and if so, then recursively test the subactivities preconditions. On each level, when the preconditions is fulfilled then the activity and the postconditions can be executed. If one or more of the activities preconditions or the subactivities preconditions should fail, the activity is suspended and tried later.

When we have tested all the preconditions in the activity-list \( P \) and if they all are true, the activity-list \( P \) is also a plan. The plan \( P \) is then used for planning, as described in section 7.2.
6. The Blocks World and STRIPS-Like Planning

The blocks world is a very simple example, which is often used in planning problems. It is very easy to understand, yet involves many problems that must be dealt with in a proper way in order to derive sound solutions.

6.1. Description of the Domain

The blocks world consists of a number of blocks, a table and a robot arm. The blocks can be stacked on each other, or put down on the table. The table is considered to have infinite many places where the blocks can be put down. The problem is to find a number of actions, which reorders the blocks in order to reach one state from another. For example, the problem could be to find a list of actions to reorder the pile of blocks in fig. 6.1 to the state showed in fig. 6.2.

![Fig. 6.1. The specified state.](image)

![Fig. 6.2. The desired state.](image)

A state (configuration) in the blocks world could be described with the predicates `table(X)`, `on(X,Y)` and `clear(X)`. For example, the configuration in fig. 1 could be described by the four statements:

```
clear(a), on(a,b), on(b,c), table(c)
```

There are a lot of possible actions that a robot can perform with a pile of blocks. We will stick to three possible actions: `stack`, `unstack`, and `move`. The action `stack` picks a block from the table and puts it onto another block. The action `unstack` performs the opposite, i.e. picks the topmost block from a pile of blocks and puts it down on the table. The action `move` moves the topmost block from one pile to another.
This can be formulated as a set of if-then rules. We assume that \( x \) and \( y \) are distinct variables:

\[
\text{stack}(X,Y):
\begin{align*}
&\text{if on table}(X) \text{ and clear}(X) \text{ and clear}(Y) \\
&\text{then not on table}(X) \text{ and not clear}(Y) \text{ and on}(X,Y)
\end{align*}
\]

\[
\text{unstack}(X,Y):
\begin{align*}
&\text{if on}(X,Y) \text{ and clear}(X) \\
&\text{then on table}(X) \text{ and clear}(Y) \text{ and not on}(X,Y)
\end{align*}
\]

\[
\text{move}(X,Y,Z):
\begin{align*}
&\text{if clear}(X) \text{ and clear}(Z) \text{ and on}(X,Y) \\
&\text{then on}(X,Z) \text{ and clear}(Y) \text{ and not clear}(Z)
\end{align*}
\]

These robot actions changes the system from one state into another, by changing the state destructively. This should be contrasted with logical implication, where a global database is updated with new information from a true implication, but nothing is ever removed from the database.

6.2. STRIPS

STRIPS was invented by Fikes and Nilsson in 1971, and is described in a number of places in the literature, for example [N1]. STRIPS's rules are divided into three parts: a precondition list, an adding list and a deleting list. The precondition list \( P \) contains the facts that must hold in order for this action to be performable. The adding list \( A \) contains the facts that becomes true when applying the rule, and the deleting list \( D \) contains facts that becomes false when applying the rule. The general format for a STRIPS rule is if \( P \) then delete \( D \) from the current state and add \( A \) to the current state. The rules mentioned above can then be formulated as:

\[
\text{stack}(X,Y):
\begin{align*}
&P: \text{clear}(X), \text{clear}(Y), \text{table}(X) \\
&D: \text{clear}(Y), \text{table}(X) \\
&A: \text{on}(X,Y)
\end{align*}
\]

\[
\text{unstack}(X,Y):
\begin{align*}
&P: \text{clear}(X), \text{on}(X,Y) \\
&D: \text{on}(X,Y) \\
&A: \text{clear}(Y), \text{table}(X)
\end{align*}
\]

\[
\text{move}(X,Y,Z):
\begin{align*}
&P: \text{clear}(X), \text{clear}(Z), \text{on}(X,Y) \\
&D: \text{on}(X,Y), \text{clear}(Z) \\
&A: \text{on}(X,Z), \text{clear}(Y)
\end{align*}
\]

For example, one possible sequence of STRIPS actions leads from the state in fig. 6.1 to the state in fig. 6.2 is:

\[
\text{unstack}(a,b), \text{move}(b,c,a), \text{stack}(c,b).
\]
6.3. STRIPS-Like Planning

Planning is accomplished by asking what should be assumed in order to derive the desired state. For example, to reach a state where the block a is on the table, the query:

\[
\text{action}(X, \text{sit}(0)) \leftarrow \text{table}(a)
\]

is asked to the program, and the system replies with the solution:

\[
X = \text{unstack}(a,b).
\]

What happens is that we are looking for what assumption that should be made in order to achieve the desired state \text{table}(a).

Another way of looking on this query is to say that \text{action}(X, \text{sit}(0)) is a function call, which should give as result \text{table}(a). Thus the call:

\[
\text{action}(X, \text{sit}(0)) \leftarrow \text{table}(a)
\]

is a form of equation solving: find an x such that the evaluation of \text{action}(X, \text{sit}(0)) gives the result \text{table}(a). (Although in this case we are using the program to pass the result \text{table}(a).)

Of course one can ask more complicated questions, like "find three actions that give as a result that block c stands on block b":

\[
\text{action}(X, \text{action}(Y, \text{action}(Z, \text{sit}(0)))) \leftarrow \text{on}(c,b).
\]

There are two solutions to this query, namely:

\[
Z = \text{unstack}(a,b), \ Y = \text{unstack}(b,c), \ X = \text{stack}(c,b)
\]

and:

\[
Z = \text{unstack}(a,b), \ Y = \text{move}(b,c,a), \ X = \text{stack}(c,b).
\]
7. STRIPS-Like Planning vs Planning in the House World

In this section we will compare STRIPS-like planning and planning in the house world. We will also explain why we think STRIPS-like planning is a subset of planning in the house world.

7.1. STRIPS-Like planning

STRIPS-like planning, like planning in the blocks world, as described in section 6 and [N1] has a given instruction-set. The problem is to find a sequence of instructions to reach a desired state from a specified state. We can say that the given instruction-set is a foundation for the solution to the planning problem (fig. 7.1).

![Fig. 7.1. The instruction-set is a foundation for the solution.]

7.2. Planning in the House World

In planning in the house world there is no given instruction-set. The first problem is to define a high-order function to find one or more possible instruction-sets from the given database (fig. 7.2).

![Fig. 7.2. Given a specification, the function finds possible instruction-sets.]

The next problem is to select one of the instruction-sets and to use it for planning (fig. 7.3). One difference, compared to STRIPS-like planning, is that the instruction-set can, or may be, executed in parallel, or in sequence, or a combination of both.
Fig. 7.3. One of the instruction-sets is selected to be the foundation for the solution.

7.3. Comparison

The form of rules in our database, as described in section 5.1, can be adopted to instructions. The general form of a instruction is then:

\[
\text{Instruction ::= Precondition} \\
\text{Instruction'} \\
\text{Consumption} \\
\text{Postcondition}
\]

According to our definition of planning in the house world, we can say that STRIPS-like planning is a subset of planning in the house world. A subset where the instruction-set is given, and all the instructions in the instruction-set are to be executed in sequence.
8. Summary

8.1. Planning

We adopted the definition of planning as described by [N1], and we called it STRIPS-like planning.

We defined a more general form of planning, which we called planning in the house world, and stated that STRIPS-like planning is a true subset of planning in the house world.

We hope our definition of planning in the house world will give a deeper understanding about general planning.

8.2. GCLA as an Implementation Language

We implemented our planning system in GCLA, and we found the language well suited to our needs. The technique for converting a rule, as described in section 5.1, in a desired database to the corresponding GCLA clause in a real database, is straightforward and self-explanatory.

GCLA is more suited to our needs than Prolog, because of its ability of hypothetical reasoning, and because of that GCLA is a superset of pure Prolog.

The experimental version of GCLA, the lack of a GCLA debugger and the lack of GCLA programming experience, have been our major problems during the development of the system.

A GCLA compiler is being developed and a SICS research report named Programming in GCLA is being written, so the future of GCLA is promising.

8.3. Future Work

An interesting future work is to integrate this sort of planning system in SICS's MDA project. The input to the planning system could be produced by a tool, which allows the user to select methods among the available methods. The output from our planning system could be used as input to a tool, also developed at SICS, which is used by the local manager on the building place.
References


