Grammars for Learning Control Knowledge with GP

Ricardo Aler
Universidad Carlos III de Madrid
Avda. de la Universidad, 30
28911 Leganés (Madrid). Spain
aler@inf.uc3m.es

Daniel Borrajo
Universidad Carlos III de Madrid
Avda. de la Universidad, 30
28911 Leganés (Madrid). Spain
dborrajo@ia.uc3m.es

Pedro Isasi
Universidad Carlos III de Madrid
Avda. de la Universidad, 30
28911 Leganés (Madrid). Spain
isasi@ia.uc3m.es

Abstract- In standard GP there are no constraints on the structure to evolve: any combination of functions and terminals is valid. However, sometimes GP is used to evolve structures that must respect some constraints. Instead of "ad-hoc" mechanisms, grammars can be used to guarantee that individuals comply with the language restrictions. In addition, grammars permit great flexibility to define the search space. EVOCK **(Evolution of Control Knowledge) is a GP based system that learns control rules for** PRODIGY**, an AI planning system.** EVOCK **uses a grammar to constrain individuals to** PRODIGY4.0 **control rule syntax. In this paper, we describe the grammar specific details of** EVOCK**. Also, the grammar approach flexibility has been used to extend the control rule language utilized by** EVOCK **in earlier work. Using this flexibility, tests were performed to determine whether using combinations of several types of control rules for planning was better than using only the standard select type. Experiments have been carried out in the blocksworld domain that show that using the combination of types of control rules does not get better individuals, but it produces good individuals more frequently.**

1 Introduction

In standard GP there are no constraints on the structure to evolve: any combination of functions and terminals is valid. All functions must be protected against anomalous arguments (as well known as *operational closure* [Koza, 1992]). However, sometimes GP is used to evolve structures that must respect some constraints. Instead of using "ad-hoc" mechanisms like in [Koza, 1994], grammars can be used to guarantee that individuals comply with the language restrictions. In addition, grammars permit great flexibility to define the search space [Whigham, 1997, Wong and Leung, 1997].

In earlier work, we have experimented with a GP based system (EVOCK: Evolution of Control Knowledge) [Aler *et al.*, 1998, Aler *et al.*, 2000a], that is able to learn control knowledge for an AI planner called PRODIGY4.0 . PRODIGY4.0 is a domain independent planner that can use domain dependent control knowledge to make planning more efficient [Veloso *et al.*, 1995]. GP can be used to evolve control knowledge. However, the PRODIGY4.0 control knowledge language is constrained and can not be protected via operational closure. Therefore we chose to use a grammar to constrain individuals.

From a grammar point of view, there are mainly two aspects that affect the learning process for problem solving: the concept language (what language we use for describing the learned concepts); and the domain language (how we model the domain knowledge). In previous work, we have explored the second aspect [Aler *et al.*, 2000b]. Here, we concentrate on exploring some hypotheses about the first aspect. In this respect, one of the main concerns when trying to learn control knowledge for a specific problem solver (as in general is the case for all learning tasks) consists on knowing which is the best language to define the learned concepts. In our prior research we fixed the concept language (as most researchers have done), hence constraining the hypothesis space. However, very little has been done in learning for problem solving on showing how this affects the results.

We used a subset of PRODIGY4.0 control knowledge language, the so-called `select-rules`, in earlier experiments. The reason was that EVOCK was used in combination with another learning system (HAMLET [Borrajo and Veloso, 1997]) that was able to use only `select-rules`. In this paper, we present empirical results in the blocksworld domain where EVOCK is used to explore a larger and flexible control knowledge language subset, including `prefer` and `reject` rules. We also describe for the first time the grammar related aspects of EVOCK.

2 The Planner PRODIGY4.0

PRODIGY4.0 is a nonlinear planning system that follows a means-ends heuristic. It performs a bidirectional search, progressively from the initial state toward the goals, and regressively from the goals to the initial state. The inputs to the problem solver algorithm are:

- Domain theory, $\mathcal{D}$ (or, for short, domain), that includes the set of operators (actions that can be applied to the environment) specifying the task

0-7803-6657-3/01/$10.00 ©2001 IEEE

knowledge and the object hierarchy;

- Problem, specified in terms of an initial configuration of the world (initial state, $\mathcal{S}$) and a set of goals to be achieved ($\mathcal{G}$); and
- Control knowledge, $\mathcal{C}$, described as a set of control rules, that guides the decision-making process.

PRODIGY4.0 planning/reasoning cycle, involves several decision points, namely:

- *Apply* an instantiated operator whose preconditions are satisfied in the current state or continue *subgoaling* on another unsolved goal. This actually determines whether the planner should work progressively or regressively.
- If the planner works regressively:
  - *select a goal* from the set of pending goals and subgoals;
  - *choose an operator* to achieve a particular goal;
  - *choose the bindings* to instantiate the chosen operator;

We refer the reader to [Veloso *et al.*, 1995] for more details about PRODIGY. For this work, we can see the planner as a program with several decision points that can be guided by control knowledge. If no control knowledge is given, PRODIGY4.0 might take the wrong decisions at some points, requiring backtracking and reducing planning efficiency. Table 1 shows two examples of control knowledge rules for the blocksworld domain. The rule in Table 1 (a) determines when the operator `unstack` must be selected: if it is desired to hold an object that is on another object in the current situation, then `unstack` should be selected (instead of `pick-up`, which only takes blocks from the table.).[1] The rule in Table 1 (b) selects the right bindings for `unstack` arguments: if it has been selected the operator `unstack` to clear a block <y>, and that block is currently under another block <x>, then the variables of `unstack` are <x> and <y> (unstack x from y). Several types of control rules are allowed. It is possible to select, prefer, or reject either operators, goals, or bindings. Control knowledge can be handed down by a programmer or learned automatically, as we do in this paper.

2.1 EVOCK

Here we only intend to provide a summary of EVOCK and refer to [Aler *et al.*, 1998] for details. EVOCK is a machine learning system for learning control rules based on Genetic Programming (GP) [Koza, 1992]. In EVOCK, each individual is a set of control rules that is manipulated by EVOCK's genetic operators. EVOCK individuals are generated and modified according to a grammar that represents the language provided by PRODIGY4.0 for writing correct control rules. EVOCK genetic operators can grow (components of) rules, remove (components of) rules and cross (parts of) rules with (parts of) other rules, just like the GP crossover operator does. EVOCK also includes some tailor made operators for modifying control rules. EVOCK guiding heuristic -the fitness function- measures individuals according to the number of planning problems from the learning set they are able to solve, the number of nodes expanded and the size of the individual (smaller individuals are preferred because they run faster and are more general).

[1] We refer those readers unfamiliar with the standard blocksworld domain to Table 6,

Table 1: Examples of control rules: (a) for selecting the unstack operator, (b) for selecting the right bindings for the unstack operator.

```
(control-rule select-operator-unstack
  (if
    (and
      (current-goal
        (holding <object1>))
      (true-in-state
        (on <object1> <object2>))))
    (then select operator unstack))
```

(a)

```
(control-rule select-arguments-for-unstack
  (if
    (and
      (current-goal (clear <y>))
      (current-operator (UNSTACK))
      (true-in-state
        (on <x> <y>))))
  (then select bindings
    ((<ob> . <x>) (<underob> . <y>))))
```

(b)

3 EVOCK Grammar

EVOCK uses two grammars to constrain individuals: domain independent and domain dependent. They are displayed in Tables 2 and 3, respectively. Terminal symbols are in lowercase, whereas non-terminal generative symbols are in uppercase.

The domain independent grammar describes the general structure of control rules (see Table 2). In particular, the top level non-terminal symbol `LIST-ROOT-T` says that an individual is a list of one or more control rules. An EVOCK individual is made of one or more control rules (`RULE-T`), which in turn are composed of a condition (`AND-T`) and an action (`ACTION-T`). A condition is a list of metapredicates (`METAPRED-T`), which in turn are functions that access the internal state of PRODIGY4.0 and return a value. EVOCK uses the following metapredicates:

- `true-in-state`: it tests whether a particular condition is true in the current planning situation.
- `current-goal`: it checks the current goal the planner is working on.

Table 2: Domain independent grammar for generating syntactically correct sets of control rules.

LIST-ROOT-T	→	(list RULE-T) | (list RULE-T RULE-T) | ...
RULE-T	→	(rule AND-T ACTION-T)
AND-T	→	(and METAPRED-T) | (and METAPRED-T METAPRED-T) | ...
METAPRED-T	→	(true-in-state GOAL-T) | (target-goal GOAL-T) | (current-goal GOAL-T) | (some-candidate-goals LIST-OF-GOALS-T)
LIST-OF-GOALS-T	→	(list-goal GOAL-T) | (list-goal GOAL-T GOAL-T) | ...
ACTION-T	→	(select-goal GOAL-T) | (select-operator OP-T) | (select-bindings BINDINGS-T) | sub-goal | apply

Table 3: Domain dependent grammar for generating syntactically correct sets of control rules in the blocksworld domain.

OBJECT	→	$< object-1 > \mid < object-2 > \mid \ldots$
OP-T	→	pickup | put-down | stack | unstack
BINDINGS-T	→	(pick-up-b OBJECT) | (put-down-b OBJECT) | (stack-b OBJECT OBJECT) | (unstack-b OBJECT OBJECT)
GOAL-T	→	(clear OBJECT) | (on-table OBJECT) | (arm-empty) | (holding OBJECT) | (on OBJECT OBJECT)

- target-goal: it tests whether a goal is one of the unachieved goals.
- some-candidate-goals: it checks whether any of a list of goals is a pending goal.

Metapredicates have typed arguments. Typing is described also in the grammar. For instance, the current-goal metapredicate requires that its argument is a goal (GOAL-T).

The action of a rule can select[2] or reject a goal, an operator, or a binding. It can also prefer a goal to another, and similarly for an operator or a binding. Finally, it can also decide to work progressively (apply) or regressively (subgoal). In some parts of a control rule, a list of goals can appear. This is what the LIST-OF-GOALS-T symbol is provided for.

Table 3 describes the domain dependent grammar. This grammar is automatically generated on the fly for every planning domain (which is an argument for EVOCK). A PRODIGY4.0 domain contains a type hierarchy and a set of planning operators. For instance, in a variant of the blocksworld domain, there could be two generic types block and robot-arm, and two subtypes of block large-block and small-block. For each type, a production rule is added to the grammar, so that variables of that type can be generated. In the standard blocksworld there is only one type: OBJECT. This gives birth to the OBJECT grammar rule of Table 3. Note that the type of variables is coded in its name (e.g. variable <object-1> is of type object).

The rest of the grammar rules come from the planning operators in the domain. In the blocksworld there are four operators pickup, put-down, stack, and unstack. They originate the OP-T and BINDINGS-T rules. Finally, planning states are represented by means of logic predicates. They (and associated information about predicate arguments) can also be extracted from operator descriptions and give rise to the GOAL-T rule.

The previous grammars are used for two main purposes:

- to generate correct individuals for the initial population. This is easily achieved by starting on the LIST-ROOT-T non-terminal symbol and randomly applying grammar rules until an individual has been built up.
- to force the genetic operators to produce correct individuals from correct individuals:
  - Mutation: it is similar to creating a whole individual. First, the mutation point is randomly selected (as in standard mutation). Then, EVOCK determines which non-terminal symbol NTS generated the element at the mutation point, cuts the subtree that hangs from that point, and grows a new subtree from that non-terminal.[3]
  - Crossover: it works likewise. First, a crossover point is selected randomly in the mother individual. Then, the non-terminal symbol that generated it is determined. Next, a crossover point that was generated with the same symbol is randomly selected in the father individual. Finally, both subtrees are swapped, as in standard crossover, to produce

[2]

[3]This requires that terminal symbols indicate non-ambiguously the non-terminal that generated it, or that ambiguity does not result in generating incorrect individuals.

the offspring.

- Besides replacing subtrees (mutation) and swapping them (crossover), EVOCK has genetic operators for adding subtrees and removing (pruning) them. Subtrees can only be added or removed at those points that have been generated by the `LIST-ROOT-T` and `LIST-OF-GOALS-T` non-terminals, which define lists of elements. These non-standard genetic operators work in a similar way to Koza's architecture altering operators [Koza, 1994] (they can add and remove branches to the tree).

4 Empirical Results

In earlier work we have experimented with a subset of PRODIGY4.0 control rule language. It used just the `select` rules.[4] This kind of rule prunes PRODIGY4.0 search very eagerly because, at any search node, it selects a single branch and rejects the rest. They are very effective if the control rule is correct, but if it is not, it might restrain PRODIGY4.0 from solving a problem. `prefer` rules are less aggressive, because they give priority to some branches on others, but no branch is ever pruned. Although results with `select` were quite good, we wanted to know if they could be improved upon by using other kind of rules as well.

In this work, we take advantage of EVOCK flexibility to define the search space via the grammar. We will extend the search space explored by EVOCK to include `prefer` and `reject` rules. In contrast to other more specialized machine learning algorithms, we only need to redefine EVOCK grammar.

EVOCK was tested in a well-known planning domain: the blocksworld. The experimental setup is as follows:

- **Fitness cases:** 192 fitness cases were randomly generated by a blocksworld problem generator. Their difficulty ranges from 1 to 5 goals.[5]
- **Population size:** two population sizes were tried: 2 and 300. The 2-population is actually a kind of hill-climbing and uses no crossover.
- **Selection method:** tournament (size of 5 in 300 size populations)
- **Fitness function:** it is a hierarchical function that measures individuals according to the number of fitness cases solved (to maximize), the number of nodes expanded (to minimize) and the size of the individual (to minimize). This function is not trivial and has been described in detail elsewhere [Aler *et al.*, 1998].
- **Running time:** EVOCK was run for 100000 evaluations. Every time a fitness case is run, it counts as an evaluation.

Table 4 displays the number of rules of each type that are used in average by best-of-run individuals. We used two grammar configurations of EVOCK: one with a `select`-only grammar and another with a `select/reject/prefer` grammar. The latter has been named `whole`.[6] We also used two different population sizes: 2 and 300 (the number in parentheses is the size of the population). In this table we wanted to explore on one side how many rules were learned of each type by best-of-run individuals in different configurations. Each EVOCK configuration was run about 50 times. Table 4 shows clearly that both `reject` and `prefer` rules are used by best-of-run individuals of the `whole` configurations, although they tend to learn `select` rules. It also shows that the `whole(2)` configuration tends to use a larger number of `select` and `reject` rules than `whole(300)` and fewer `prefer` rules.

Table 4: Average number of rules of each type used by best-of-run individuals.

Configuration	`Select`	`Reject`	`Prefer`
select(2)	2.97	0.00	0.00
select(300)	2.97	0.00	0.00
whole(2)	1.91	1.09	0.25
whole(300)	1.76	0.62	0.74

However, are `reject/prefer` rules useful? Figure 1 summarizes experimental results to answer this question. To obtain those results, all the best-of-run individuals were tested with a testing set consisting of 416 hard planning problems, to check if the control knowledge learned escalates well to more difficult problems. The timeout given for solving each test problem is $t_{test} = \frac{150}{16}(1 + \text{floor}(\frac{\#\text{goals}}{10}))$ seconds, where #goals is the number of goals in the testing problem. This number ranges from 5 to 50.[7] Figure 1 shows, for every configuration, the frequency of experiments (y-axis) that are able to solve a proportion x of problems (x-axis) or more. The vertical line represents PRODIGY4.0 with no control knowledge.

All configurations obtain a similar maximum (80% of problems solved). Therefore, it seems that extending the grammar does not produce more clever individ-

[4] Actually, our `select` configuration includes `apply` and `sub-goal` control rules.

[5] A planning problem can have several goals to achieve. Usually, the more goals it has, the more difficult the problem is.

[6] Extending the grammar increases the number of degrees of freedom of the 'whole' configuration, hence there is much greater potential for overlearning. On the other hand, using a richer language might offer new opportunities to the learning system. Which tendency will dominate can only be determined with an empirical comparison.

[7] 50 goal problems are very hard problems which are unsolvable by many current planners.

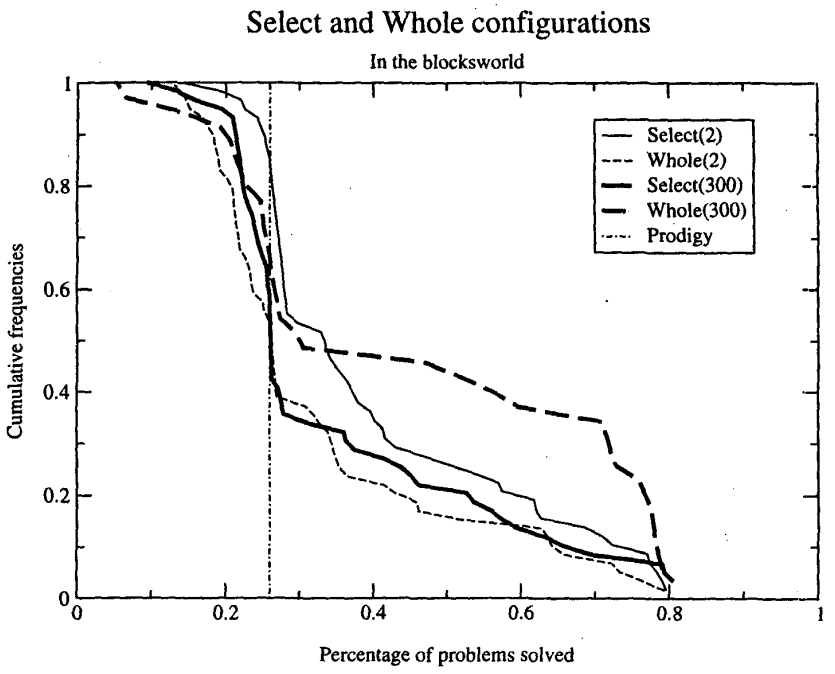

Figure 1: Frequency of experiments (y-axis) that are able to solve a proportion x of problems (x-axis) or more, in the blocksworld planning domain.

Table 5: Percentage of best-of-run individuals that solve at least 60%, 70%, and 78% testing problems (respectively) and that use certain kind of rules, in the 2 and 300 `whole` configurations.

	`whole(2)`			`whole(300)`		
	60%	**70%**	**78%**	**60%**	**70%**	**78%**
Type						
select	100%	100%	100%	75%	75%	100%
reject	50%	50%	0%	58%	58%	100%
prefer	25%	0%	0%	75%	75%	75%

uals. However, the `whole(300)` configuration obtains good individuals more frequently than the rest of configurations. Thus, it seems that the combination of exploration (a 300-population) and the extended grammar is positive. Results in Table 5 seem to confirm this. It displays the frequency of the best best-of-run individuals that contain `select`, `reject`, and `prefer` rules, respectively. The first three columns refer to `whole(2)` and the three last ones, to `whole(300)`. Each column considers only the best-of-run individuals that are able to solve, at least, 60%, 70%, and 78% of the testing problems, respectively. For instance, the 100% in the first line and first column means that all the best-of-run individuals that solve 60% of problems (or more) use `select` rules. However, only 50% of them use `reject` rules. There are three interesting results:

- First, `select` rules seem to be required by individuals in order to obtain good results (`whole(2)`, best-of-run individuals always have `select` rules. `whole(300)` is also very close to this.
- Second, only 50% of the `whole(2)` best-of-run individuals use `reject` rules. `whole(300)` makes better use of them (from 58% to 100%).
- Third, the `whole(2)` configuration does not use `prefer` rules in order to obtain good results, whereas the `whole(300)` uses them very frequently (75%). Therefore, this seem to confirm that `prefer` rules are the reason behind `whole(300)` obtaining good individuals more frequently (the best of all the best-of-run individuals does not use any `prefer` rule). It is remarkable that only the more exploratory configuration can make good use of `prefer` rules. The right hand side of `prefer` rules have two arguments (`select` and `reject` rules have only one). Perhaps it is difficult for the `whole(2)` configuration to guess the two arguments correctly because of its strong bias towards simplicity: if a mutation generates a `prefer` right hand side, and its two arguments contain the wrong combination, the mutated individual will not survive the next generation. `whole(300)`

Table 6: Definition of the blocksworld domain for PRODIGY.

```
PICK-UP (<ob>)          PUT-DOWN (<ob>)        STACK (<ob> <underob>)  UNSTACK (<ob> <underob>)
 (clear <ob>)            (holding <ob>)         (clear <underob>)       (on <ob> <underob>)
 (on-table <ob>)                                (holding <ob>)          (clear <ob>)
 (arm-empty)                                                            (arm-empty)
 =>                      =>                     =>                      =>
 del (on-table <ob>)     del (holding <ob>)     del (holding <ob>)      del (on <ob> <underob>)
 del (clear <ob>)        add (clear <ob>)       del (clear <underob>)   del (clear <ob>)
 del (arm-empty)         add (arm-empty))       add (arm-empty)         del (arm-empty)
 add (holding <ob>       add (on-table <ob>)    add (clear <ob>)        add (holding <ob>)
                                                add (on <ob> <underob>) add (clear <underob>)
```

can keep incorrect individuals for longer, and they stand a chance of being corrected.

This result could only be obtained by exploiting grammar EVOCK richness and declarativeness. Other control knowledge learning methods, such as HAMLET [Borrajo and Veloso, 1997], use "ad-hoc" grammars in the sense that they have them programmed. Therefore, exploring this type of hypothesis requires a heavy programming effort. For instance, incorporating new types of rules or metapredicates implies having to re-program their code.

5 Examples of individuals

The aim of this section is to present some examples of individuals obtained by EVOCK using the two different grammars and explain how they work. Although individuals are usually simple because of the strong bias towards this end, it is often difficult to understand what they do because they do not contain the whole planning algorithm, but only the control knowledge to guide the planner. Their interactions with the base planner are not always easy to follow. Here, we have chosen to explain not whole individuals, but isolated control rules from evolved individuals whose meaning seems to be independent from the rest of the control rules that make up the individuals (this is not always the case). One of the control rules comes from the best individual of `select(300)` (80% of testing problems solved). The second control rule comes from a very good individual (78%) of `select(300)`, which actually contains a `prefer` rule. They are shown in Figures 3 and 4.[8]

The control rule of Figure 3 says that if PRODIGY4.0 is trying to solve a `(clear <a>)` subgoal, then the `unstack` operator should be chosen. In a similar way, the control-rule of Figure 3 says that if `(clear <a>)` has to be achieved and there is at least one `clear` block in the world, then `unstack` should be preferred to `put-down`. Note that although this strategy works for the problem of Figure 2, where the blocks to clear are under other blocks, it would not work when the block is not clear because it is held by the robot arm. In that case, either `put-down` or `stack` should be used to clear the block.

Actually, the control rule of Figure 4 is more correct than the one in Figure 3 because it detects a particular situation where `unstack` should not be preferred. The extra condition `(clear <b>)` achieves this. This condition is always true, except in the case when there is only one block in the world, which is being held by the robot arm. But in that case, the only way to `clear` the block is by putting it down on the table, or on another block (hence unstack should not be chosen in that case).

Basically, both control rules seem to be solving a related subproblem. This should not be surprising. After all, we have found that individuals evolved by the two grammars get to the same maximum, therefore individuals should be equivalent, even if they use different grammars. Both control rules have learned how to use the `unstack` operator. In order to understand how they work, at least in some cases, the following simple problem will be traced: we consider two blocks `A` and `B`, which are on the table, and two other blocks `C1` and `E1`, which are on `A` and `B`, respectively. The goal is to put `A` on `B` (i.e. `(on A B)`). In order to achieve it, both `C1` and `E1` have to be `unstacked` from `A` and `B`, respectively. Figure 2 shows how PRODIGY4.0 solves this problem. Table 6 contains the standard blocksworld definition we have used, so that the trace can be easily followed.

```
(current-goal (clear <a>))
=>
(select-operator unstack)
```

Figure 3: Control rule obtained from the best individual of the `select(300)` configuration.

```
(current-goal (clear <a>))
(true-in-state (clear <b>))
=>
(then prefer-operator unstack put-down)
```

Figure 4: Control rule obtained from a good individual of the `whole(300)` configuration.

[8] Those control rules have also been slightly simplified, so that they are more understandable.

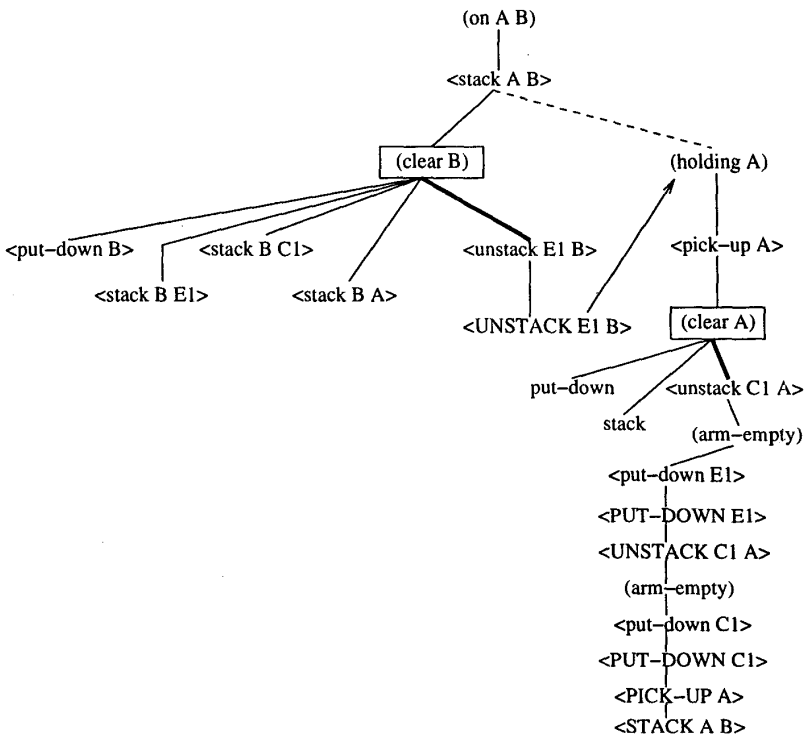

Figure 2: PRODIGY4.0 trace of a planning problem whose goal is to stack A on B, starting from an initial state where A and B are on the table and C1 and **E1** are on **A** and B, respectively. The effect of control rules is highlighted with a thick line, ath the `(clear B)` and `(clear A)` points.

`(on A B)` can only by achieved by `<stack A B>`. This operator requires that A is clear and that A is being held. PRODIGY4.0 solves the first goal first. `(clear B)` can be achieved by three operators: `<put-down>`, `<stack>`, and `<unstack>`. Obviously, the latter is the right one. However, PRODIGY4.0 tries the operators in the order they appear in the domain file (`pick-up`, `put-down`, `stack`, and `unstack`, in our case). Actually, PRODIGY4.0 tries all possible instantiations of these operators, which in this case are: `<put-down B>`, `<stack B E1>`, `<stack B C1>`, and `<stack B A>`, until it gets to the right one `<unstack E1 B>`.[9] In this case, there are only a few blocks, but the more blocks there are in the problem, the more instantiations PRODIGY4.0 will try. Hence, it is very important that PRODIGY4.0 guesses the right operator at this point. This is what the control rules of Figures 3 and 4 do: they prune the wrong operators when this is required. Their effects have been highlighted with a thick line in Figure 2 at the two points where they can be executed (`(clear B)` and `(clear A)`).

[9]Actually, PRODIGY4.0 should also try <unstack A B> and <unstack C1 B> but a domain independent heuristic forces PRODIGY4.0 to use <unstack E1 B> first.

Once `<unstack E1 B>` has been selected, PRODIGY4.0 decides to apply it to the current situation (which is still the initial situation) to change the world. This is represented by `<UNSTACK E1 B>` (uppercase means operator application). Now the robot arm is holding block E1 and B is clear. Then, PRODIGY4.0 tries to achieve the second precondition of `<stack A B>`: `(holding A)`. `pick-up A` is tried first, which requires that block A is `clear`. Again, both control rules would prune the search tree to select `unstack C1 A`, reducing the search effort. The rest of the problem is unaffected by the learned control knowledge (PRODIGY4.0 finds the solution without backtracking anyway).

6 Conclusions

EVOCK is a GP based system that learns control knowledge for an AI planning system (PRODIGY). Earlier work has shown that EVOCK is quite successful in several planning domains. Up to now, EVOCK has used only a subset of PRODIGY4.0 control knowledge language, the so-called `select` control rules. However, PRODIGY4.0 allows a larger subset, that includes `reject` and `prefer` rules. EVOCK is a GP system that uses a grammar to

constrain individuals. Operational closure cannot be used for that purpose, because individuals are not executable programs as in standard GP, but structures that must comply with a syntax that cannot be changed. As EVOCK uses a grammar, modifying it to use new kinds of rules is both simple and declarative.

In this paper we have extended EVOCK grammar and compared the results with a `select`-only grammar. Experimental results show that the new defined language through the extended grammar produces a best-of-all-runs individual that obtains similar results to the previously used language. However, the new language produces good individuals more frequently than the old one, because of the use of `prefer` rules.

In summary, grammar-based GP can be used to rapidly and flexibly test language based learnability hypotheses. Future research will try to test other language hypothesis (like negated metapredicates) to define the best language to be used for this learning task.

7 Future Work

- Currently, our grammar approach requires that non-terminal symbols can be identified non-ambiguously from terminals in the individual. Although we have not found this particularly limiting for our current application, we intend to let individuals be the actual parse trees, as it is done in most of the research in this field [Whigham, 1997, Wong and Leung, 1997].
- In this paper, we use a hierarchical fitness function, which trades off three objectives: number of solved cases, number of nodes expanded, and size of individual. Although we have found this kind of function useful to express our preferences, it may be worth looking at multi-objective pareto optimisation approaches (see for instance [Langdon, 1995]).

Acknowledgements

The research reported here was carried out as part of the research project funded by CICYT TAP-99-0535-C02 (http://decsai.ugr.es/~lcv/SEPIA/tap99-0535-c02-01.html).

We thank the anonymous reviewers for their suggestions. They helped considerably to improve this article.

Bibliography

[Aler *et al.*, 1998] Ricardo Aler, Daniel Borrajo, and Pedro Isasi. Genetic programming and deductive-inductive learning: A multistrategy approach. In Jude Shavlik, editor, *Proceedings of the Fifteenth International Conference on Machine Learning, ICML'98*, pages 10–18, Madison, Wisconsin, July 1998.

[Aler *et al.*, 2000a] Ricardo Aler, Daniel Borrajo, and Pedro Isasi. GP fitness functions to evolve heuristics for planning. In Martin Middendorf, editor, *Evolutionary Methods for AI Planning*, pages 189–195, Las Vegas, Nevada, USA, 8 July 2000.

[Aler *et al.*, 2000b] Ricardo Aler, Daniel Borrajo, and Pedro Isasi. Knowledge representation issues in control knowledge learning. In *Proc. 17th International Conf. on Machine Learning*, pages 1–8. Morgan Kaufmann, San Francisco, CA, 2000.

[Borrajo and Veloso, 1997] Daniel Borrajo and Manuela Veloso. Lazy incremental learning of control knowledge for efficiently obtaining quality plans. *AI Review Journal. Special Issue on Lazy Learning*, 11(1-5):371–405, February 1997.

[Koza, 1992] John R. Koza. *Genetic Programming: On the Programming of Computers by Means of Natural Selection.* MIT Press, Cambridge, MA, USA, 1992.

[Koza, 1994] John R. Koza. *Genetic Programming II: Automatic Discovery of Reusable Programs.* MIT Press, Cambridge Massachusetts, May 1994.

[Langdon, 1995] W. B. Langdon. Pareto, population partitioning, price and genetic programming. Research Note RN/95/29, University College London, Gower Street, London WC1E 6BT, UK, April 1995.

[Veloso *et al.*, 1995] Manuela Veloso, Jaime Carbonell, Alicia Prez, Daniel Borrajo, Eugene Fink, and Jim Blythe. Integrating planning and learning: The PRODIGY architecture. *Journal of Experimental and Theoretical AI*, 7:81–120, 1995.

[Whigham, 1997] P. A. Whigham. Evolving a program defined by a formal grammar. In *Fourth International Conference on Neural Information Processing - The Annual Conference of the Asian Pacific Neural Network Assembly (ICONIP'97)*, pages 456–459, Dunedin, New Zealand, 1997.

[Wong and Leung, 1997] Man Leung Wong and Kwong Sak Leung. Evolutionary program induction directed by logic grammars. *Evolutionary Computation*, 5(2):143–180, 1997.