An Empirical Study of Agent Programs
A Dynamic Blocks World Case Study in GOAL

M. Birna van Riemsdijk and Koen V. Hindriks

EEMCS, Delft University of Technology, Delft, The Netherlands
{m.b.vanriemsdijk,k.v.hindriks}@tudelft.nl

Abstract. Agent-oriented programming has been motivated in part by the conception that high-level programming constructs based on common sense notions such as beliefs and goals provide appropriate abstraction tools to develop autonomous software. Various agent programming languages and frameworks have been developed by now, but no systematic study has been done as to how the language constructs in these languages may and are in fact used in practice. Performing a study of these aspects may contribute to the design of best practices or programming guidelines for agent programming, and clarify the use of common sense notions in agent programs. In this paper, we analyze various agent programs for *dynamic blocks world*, written in the Goal agent programming language. We present several observations based on a quantitative and qualitative analysis that provide insight into more practical aspects of the development of agent programs. Finally, we identify important issues in three key areas related to agent-oriented programming that need further investigation.

1 Introduction

The concept of a *goal* lies at the basis of our understanding of why we perform actions. It is common sense to explain the things we do in terms of beliefs and goals. The *reasons* for performing actions are derived from our motivations and the notion of *rational behavior* is typically explained in terms of actions that are produced in order to further our goals. For example, a researcher who has a goal to have finished a paper but is going on a holiday instead is not considered to behave rationally because holidays do not further the goal of writing a paper.

Shoham was one of the first who proposed to use such common sense notions to build programs [14], coining a new programming paradigm called *agent-oriented programming*. Inspired by Shoham, a variety of agent-oriented programming languages and frameworks have been proposed since then [3, 4]. For several of them, interpreters and Integrated Development Environments (IDEs) are being developed. Some of them have been designed mainly with a focus on building practical applications (e.g., JACK [17] and Jadex [13]), while for others the focus has been also or mainly on the languages' theoretical underpinnings (e.g., Goal [10], 2APL [6], and Jason [5]).

In this paper, we take the language Goal as our object of study. Goal is a high-level programming language to program rational agents that derive their choice of action from their beliefs and goals. Although the language's theoretical basis is important, it *is* designed by taking a definite *engineering stance* and aims at providing useful programming constructs to develop agent programs.

However, although it has been used for developing several small-size applications, no systematic study has been done as to *how the language constructs are used* in practice to program agents, and *how easy it is to read* the resulting programs. Also, if the language is going to be used for building real-world applications, *efficiency* and a programmer's knowledge of this becomes an important issue. We believe it is important to get a better understanding of these issues in order to identify which aspects need to be addressed with respect to them, and to be able to design a set of best practices or *programming guidelines* that support Goal programmers. It is the purpose of this paper to contribute to this aim. We do this by analyzing three Goal programs for the dynamic blocks world domain. To the best of our knowledge, this is the first time such comparative analysis of agent programs programmed in a dedicated agent programming language has been done. This kind of empirical software engineering is expected to form over time a body of knowledge leading to widely accepted and well-formed theories [2].

The dynamic blocks world domain is explained in Section 2 and in Section 3 we explain the Goal language. We outline our approach in Section 4. In Section 5, we analyze the programs using various numeric measures on their code and execution. In Section 6, we look in more detail at how the programs were written and how the language constructs of Goal were used. We conclude in Section 7.

2 The Dynamic Blocks World

The blocks world is a simple environment that consists of a finite number of blocks that are stacked into *towers* on a table of *unlimited* size. It is assumed that each block has a unique label or name $a, b, c, ...$. Blocks need to obey the following "laws" of the blocks world: (i) a block is either on top of another block or it is located somewhere on the table; (ii) a block can be directly on top of at most one other block; and, (iii) there is at most one block directly on top of any other block.

A *blocks world problem* is the problem of which actions to perform to transform an initial state or configuration of towers into a goal configuration, where the exact positioning of towers on the table is irrelevant. A blocks world problem thus defines an action selection problem. The action of moving a block is called *constructive* (see, e.g., [15]) if in the resulting state that block is in position, meaning that the block is on top of a block or on the table and this corresponds with the goal state, and all blocks (if any) below it are also in position. Observe that a constructive move always increases the number of blocks that are in position.

We have used a specific variant of the blocks world, which we call the *dynamic blocks world*. In the dynamic blocks world, a user can move blocks around while the agent is moving blocks to obtain a goal configuration. It was introduced in [11], and comes with an implemented environment[1]. In that environment, there is a gripper that can be used to move blocks and the user can move blocks around by dragging and dropping blocks in the environment's graphical user interface. The agent can steer the gripper by sending two kinds of actions to the environment: the action `pickup(X)` to tell it to pick up a block `X`, and the action `putdown(X,Y)` to tell it to put down the block `X`, which should be the block the gripper is currently holding, onto the block `Y`. The gripper can hold at most one block. The environment has a maximum of 13 blocks, and these can all be on the table at the same time (thereby realizing a table that is always "clear"). The user can move the blocks around on the table, put a block inside the gripper, or take away a block from the gripper. In contrast with the gripper, which can only pick up a block if there is no block on top of it and move it onto a block that is clear, the user can move any block in any way he likes.

The fact that a user can move around blocks can give rise to various kinds of possibly problematic situations. For example, the agent may be executing the action `putdown(a,b)`, while the user moves some block on top of `b`. This means that `a` can no longer be put down onto `b`, since `b` is not clear anymore. It may also be the case that the agent is moving a block `a` from some other block onto the table, since `a` could not yet be moved in a constructive way. It may be the case that while the agent is doing that, the user moves blocks in such a way that now `a` *can* be moved into position, making the previous action superfluous. A comprehensive list of such cases where the agent has to deal with the dynamics of the environment, can be found in [16].

3 Explanation of GOAL

A Goal agent decides which action to perform next based on its beliefs and goals. Such a decision typically depends on the current state of the agent's environment as well as general knowledge about this environment. The former type of knowledge is typically dynamic and changes over time, whereas the latter typically is static and does not change over time. In line with this distinction, two types of knowledge of an agent are distinguished: conceptual or domain knowledge stored in a *knowledge base* and beliefs about the current state of the environment stored in a *belief base*. In the implementation of Goal, both knowledge base and belief base are Prolog programs. For example, the knowledge base may contain the definition of the predicate `clear(X)`, and the belief base may contain information about which blocks are present and how they are stacked, represented using the predicate `on(X,Y)`, as illustrated in Table 1.

A decision to act will usually also depend on the goals of the agent. Goals of an agent are stored in a *goal base*. The goal base consists of conjunctions of

[1] `http://www.robotics.stanford.edu/users/nilsson/trweb/TRTower/TRTower_links.html`

Prolog atoms. For example, the goal to build a tower where `a` is stacked on `b` can be represented as a conjunction of the corresponding `on` predicates (see Table 1). The goals of an agent may change over time, for example, when the agent adopts a new goal or drops one of its goals. As a rational agent should not pursue goals that it already believes to be achieved. GOAL provides a built-in mechanism for doing so based on a so-called *blind commitment strategy*. This strategy is implemented by removing conjunctions of atoms that represent goals from the goal base, as soon as they are believed to be achieved. These conjunctions will only be removed if they have been achieved *completely* in the current state. Together, the knowledge, beliefs and goals of an agent make up its *mental state*.

```
knowledge{
  clear(table).
  clear(X) :- block(X), not(on(Y,X)).
}
beliefs{
  block(a). block(b). block(c).
  on(a,table). on(c,b). on(b,table).
}
goals{
  on(a,b), on(b,table).
}
program{
  if bel(holding(X)), a-goal(on(X,table)) then putdown(X,table).
}
```

Table 1. Knowledge Base, Belief Base, Goal Base, Action Rules

To select an action, a GOAL has so-called *action rules*. The action rules consist of a condition on the mental state of the agent, and an action that should be executed if the mental state condition holds. In essence, writing such conditions means specifying a *strategy* for action selection that will be used by the GOAL agent. A mental state condition can consist of conditions on the agent's beliefs and on the agent's goals. Informally, `bel(`φ`)` can be read as "the agent believes that φ". `bel(`φ`)` holds whenever φ can be derived from the belief base *in combination with the knowledge base*. In the example of Table 1, it follows that `bel(clear(a))`, which expresses that the agent believes that block `a` is clear.

Similarly, `goal(`φ`)` can be read as "the agent has a goal that φ". `goal(`φ`)` holds whenever φ can be derived from *a single goal* in the goal base *in combination with the knowledge base*. In the example of Table 1, it follows, e.g., that `goal(on(b,table))`. In order to represent achievement goals, i.e., goals that are not believed to be achieved yet, the keyword `a-goal` can be used. This is defined as follows:

$$\texttt{a-goal}(\varphi) \overset{df}{=} \texttt{goal}(\varphi)\texttt{, not(bel(}\varphi\texttt{))}$$

In the example, `a-goal(on(a,b))` holds, but `a-goal(on(b,table))` does not. Similarly, `goal-a(`φ`)` represents that φ can be derived from a goal in the goal base, but φ *is* already believed to be achieved. A mental state condition is a conjunction of these mental atoms, or their negation. Table 1 shows an example of an action rule that specifies the following: if the agent believes it is holding block `X` and has the achievement goal of having `X` on the table, then the corresponding `putdown` action should be selected. If the conditions of multiple action rules hold at the same time, an applicable rule is selected at random.

Actions that may be performed by a GOAL agent need to be specified by the programmer of that agent. GOAL does provide some special built-in actions but typically most actions that an agent may perform are derived from the environment that the agent acts in. Actions are specified by specifying the conditions when an action can be performed (preconditions) and the effects of performing the action (postconditions). Pre- and postconditions are conjunctions of literals. A precondition φ is evaluated by verifying whether (an instantiation of) φ can be derived from the belief base (as always, in combination with knowledge in the knowledge base). Any free variables in a precondition may be instantiated during this process just like executing a Prolog program returns instantiations of variables. In GOAL, the effect φ of an action is used to update the beliefs of the agent to ensure the agent believes φ after performing the action. The positive literals are added to the belief base, and negative literals are removed. In addition, actions that correspond to actions that can be executed in the agent's environment, are sent to that environment. In the dynamic blocks world, the `pickup` and `putdown` actions are specified as in [16]. They have a true postcondition, and therefore do not update the belief base.

In addition to the possibility of specifying user-defined actions, GOAL provides several built-in actions for changing the beliefs and goals of an agent, and for communicating with other agents. Here we only briefly discuss the two built-in actions `adopt(`φ`)` and `drop(`φ`)` which allow for modifying the goal base of an agent. The action `adopt(`φ`)` is an action to adopt a new goal φ. The precondition of this action is that the agent does not believe that φ is the case, i.e. in order to execute `adopt(`φ`)` we must have `not(bel(`φ`))`. The idea is that it would not be rational to adopt a goal that has already been achieved. The effect of the action is the addition of φ as a single, new goal to the goal base. The action `drop(`φ`)` is an action to drop goals from the goal base of the agent. The precondition of this action is always true and the action can always be performed. The effect of the action is that any goal in the goal base from which φ can be derived is removed from the goal base. For example, the action `drop(on(b,table))` would remove all goals in the goal base that entail `on(b,table)`; in the example agent of Table 1 the only goal present in the goal base would be removed by this action.

A final aspect of GOAL that we have to discuss, is sensing. Sensing is not represented as an explicit act of the agent but a perceptual interface is defined between the agent and the environment that specifies which percepts an agent will receive from the environment. Each time after a GOAL agent has performed an action, the agent processes any *percepts* it may have received through its perceptual interface. Incoming percepts are processed through percept rules. The percept rules for the dynamic blocks world are specified in [16]. Percept are of the form `block(X)`, representing that there is a block `X` in the environment, `holding(X)`, representing that the gripper is holding block `X`, and `on(X,Y)`, representing that block `X` is on `Y`. The percept rules specify that these atoms are added to the belief base as soon as they are perceived (indicated by the `percept` keyword), and they are removed from the belief base if they are not perceived.

4 Approach

In this section, we describe the research approach that we have followed. First, we have asked three programmers to program a Goal agent for the dynamic blocks world. We refer to the resulting programs as A, B, and C. The code of the programs can be found in the corresponding appendices. The person who programmed A had the least experience with Goal, while the programmer of C had the most. All programmers were somewhat familiar with the blocks world domain. As a starting point, they were given the action specification and percept rules of [16]. Another constraint was that the agent would only get one goal configuration to achieve. They were also given a set of test cases in order to test the functioning of their program. Some of these test cases are included in [16]. After the programs were handed in for analysis, they were not modified anymore.

We have performed three kinds of analyses on the programs. First, we have analyzed the code, both using quantitative metrics such as the number of action rules (Section 5.1), as well as performing a qualitative analysis, looking in more detail at the code itself (Section 6). Second, we have performed an experiment where we have asked six test subjects to look at the code of all three programs and comment on their readability (Section 6.2). The test subjects were somewhat familiar with the Goal language, but did not have extensive experience in programming with it. All comments were removed from the programs before they were given to the test subjects, but white space was preserved. Third, we have performed limited testing of the programs to obtain quantitative metrics on the action sequences that were executed (Section 5.2).

5 Quantitative Analysis

In this section, we compare the three Goal agent programs for the dynamic blocks world based on numeric measures of their code (Section 5.1) and of behavior shown during execution (Section 5.2). In this and the next section, we summarize our main findings by listing observations. It is important to note that these observations are based only on the three blocks world programs that we used for our analysis.

5.1 Code

We provide numeric measures for each of the sections of a Goal program, except for the action specification and percept rules sections, since these formed the starting point for all three programs (see Section 4) and are the same (with a slight exception for A, see [16]). The results are summarized in Table 2.

Before we discuss the table, we provide some additional information on how to interpret our measures for action rules. The other measures speak for themselves. The total number of action rules is counted for each of the programs, and is also split into environment actions (`pickup` or `putdown`), and actions for adopting or dropping a goal. We also counted the number of belief and goal conditions in action rules. We provide the average (avr) number of conditions per rule, and the minimum (min) and maximum (max) number of conditions that have been used in one rule. The average number of conditions is obtained by dividing the total number of conditions by the total number of rules. The conditions have been counted such that each atom inside belief or goal keyword was counted as one condition. For example, a conjunctive belief condition with n conjuncts, `bel(cond1, ..., condn)`), is counted as n conditions, and similarly for a disjunctive belief condition. If the number of belief or goal conditions that are used is 0, we do not split this into avr/min/max, but simply write 0.

Table 2. Numeric Measures of Code

Numeric measure	**A**	**B**	**C**
clauses knowledge base	16	4	8
defined Prolog predicates in knowledge base	11	2	3
clauses (initial) belief base	0	0	1
goals (initial) goal base	0	1	1
action rules [env. action/adopt/drop]	3 [3/0/0]	14 [5/6/3]	12 [3/3/6]
`bel` conditions in action rules (avr/min/max)	1.3/1/2	1.8/0/4	1.7/0/6
`a-goal` conditions in action rules (avr/min/max)	0	1.6/1/2	0.8/0/1
`goal` conditions in action rules (avr/min/max)	0	0	0.8/0/2
`goal-a` conditions in action rules (avr/min/max)	0	0	0.08/0/1

As can be seen in Table 2, the extent to which the knowledge base is used differs considerably across the three programs. Where A has 16 clauses and 11 defined predicates in the knowledge base, thereby making heavy use of Prolog, program B has only 4 clauses and 2 defined predicates. The belief base initially has very little information in all three programs, which suggests that it is used mostly through updates that are performed during execution by means of the percept rules. Both B and C initially have one goal in the goal base, which reflects the fact that in our setting we consider only one goal configuration of the blocks. Program A does not use the goal base for representing the goal configuration.

The number of action rules is very small for program A (only 3), while B and C use considerably more rules (14 and 12, respectively). Also, program A only uses action rules for selecting environment actions, while the majority of the rules of B and C (9 in each case) concern the adoption or dropping of goals. Moreover, A only uses a small number of belief conditions in the rules (maximum of 2), and does not make use of goal conditions. The latter corresponds with the fact that in A, no goals are inserted into the goal base (neither initially, nor through the use of action rules). The number of belief conditions in B and C are comparable, ranging from 0 to 4 or 6 conditions per rule, respectively. The number of conditions on goals in B and C is rather similar (1.6 on average), and is typically smaller than the number of belief conditions (maximum of 2). The use of conditions on goals differs for B and C in that B uses only `a-goal` conditions, while in C there is an equal number of `a-goal` and `goal` conditions, and one `goal-a` condition. Program C thus makes the most use of the various constructs offered by GOAL.

What we did not include in the table is that almost all rules in B and C have at least one positive, i.e., non-negated, condition on goals (only one exception in B). This corresponds with the idea that actions are selected because an agent wants to reach certain goals. None of the programs use the action rules to select actions for updating the belief base.

We summarize our findings through a number of main observations. The first concerns the relation between the experience that programmers have with the GOAL language, and how this relates to their use of the constructs.

Observation 1 (Experience with GOAL) *For our programs it is the case that the more experienced the programmer is with* GOAL*, the more of the language constructs offered by* GOAL *are used.*

This suggests that programmers have a tendency to stick to what they know best, rather than try out constructs they are less familiar with. This means that education and training is essential if programmers are to make full use of the features offered by GOAL. The observation is also in line with the following observation, which addresses the use of the knowledge base in comparison with the action rules.

Observation 2 (Focus on Knowledge Base or Action Rules) *Two ways in which the* GOAL *language can be used, are by focusing on the knowledge base and keeping the number of action rules small, or by focusing on the action rules and keeping the knowledge base small.*

A final observation of this section concerns the use of action rules for adopting and dropping goals, in comparison with rules for selecting environment actions.

Observation 3 (Many Action Rules for Adopt or Drop) *In both programs that use goals, the number of action rules for adopting or dropping goals is considerably larger than the number of rules for selecting environment actions.*

5.2 Execution

Besides looking at the code of the programs, we have also analyzed their behavior during execution. We have run the programs using four test cases (see [16]), of which two included dynamics (one block was moved by the user while the agent was executing). The number of blocks ranged from 3 to 13. We have recorded the number of actions (both environment actions as well as adopt and drop actions) that were executed by the agent in one run of each test case (see Table 3).[2] The number of executed actions to solve a certain problem can be taken as a measure for the *efficiency* of an agent program.

Table 3. Executed Actions

Test Case	**Action Type**	**A**	**B**	**C**
1	adopt or drop	0	2	3
	env. action	4	4	4
2	adopt or drop	0	36	18
	env. action	28	36	30
3	adopt or drop	0	10	8
	env. action	11	12	10
4	adopt or drop	0	8	12
	env. action	8	9	8

All three programs achieved the goal in all four test cases. The number of executed environment actions was comparable for all three programs throughout the test cases, although program B always executed a little more than A and C (up to 20% more). Since A does not use adopt or drop actions in the program, the only actions executed by A were environment actions. By contrast, B and C execute a considerable amount of adopt and drop actions. The average portion of adopt or drop actions compared to the total number of actions was 0.44 and 0.46 across the four test cases for B and C, respectively. It ranged between 0.33 and 0.50 for B, and between 0.38 and 0.60 for C. We thus make the following observation, which seems in line with Observation 3.

Observation 4 (Number of Executed Adopt or Drop Actions) *In the programs that use goals, the adopt and drop actions form a considerable portion of the total number of executed actions.*

When taking the total number of executed actions as a measure for efficiency, it is clear that A is much more efficient than B and C. However, when looking only at environment actions, the programs' efficiency is comparable. Whether to take the number of executed environment actions as a measure for efficiency or whether to also take into account (internal) adopt and drop actions, depends on how much time it takes to execute them. Assuming that the selection and execution of adopt or drop actions takes comparably little time compared with the execution of environment actions, one can take only the executed environment actions as a measure for the efficiency of the program. However, if this is not the

[2] If the same action was sent to the environment multiple times in a row, this was counted as one action. Sending an action to the environment multiple times in a row can happen if the action rule for selecting that action keeps being applicable while the action is being executed.

case then the selection of adopt and drop actions should be taken into account. In that case it should be carefully considered by the programmer whether the reasoning overhead is necessary and how it may be reduced. In our case study, however, environment actions took considerably more time than adopt or drop actions.

This data is based on one run per program per test case. However, we did do several more runs for some of the test cases. Those runs showed that there was non-determinism in the programs, since the execution traces were not identical in all cases. Non-determinism can, e.g., occur if the goal of the agent is to put block a on b on c on d on the table, and initially block a is on c and b is on d. In this case, the agent has no choice but to move both a and b onto the table. However, the order in which this is done may differ.

The fact that the programs show non-determinism, means that they *under-specify* the behavior of the agent. The programs are thus a high-level specification of agent behavior, in the sense that they do not specify the behavior to the full detail. This leaves room for *refinement* of the programs in several was (see, e.g., [8, 1, 9] for approaches that take advantage of this).

Observation 5 (Underspecification) GOAL *naturally induces the specification of non-deterministic agents, i.e., agents of which the behavior is underspecified.*

6 Analysis of Programming Aspects

In this section, we discuss the code of the GOAL programs in more detail. We describe their structure (Section 6.1), and discuss similarities and differences (Section 6.2). We do not discuss the process of programming GOAL agents. The reason is that we did not collect sufficient data on how the programmers went about programming the agents. Also, where usually one would start by thinking about percepts (and percept rules) and action specifications, these were given in our setting. Therefore, the focus was on programming the strategy of how to solve the dynamic blocks world problem.

6.1 Structure of Programs

Program A The knowledge base is used to determine where blocks should be moved. This is done by defining a predicate `goodMove(X,Y)` on the basis of several other predicates. A distinction is made between a constructive move, which moves a block to construct a goal tower, and an unpile move, which moves a block to the table in order to clear blocks such that eventually a constructive move can be made. If possible, a constructive move is selected. The goal configuration of the blocks is specified in the knowledge base, rather than in the goal base. The predicate `isgoal(tower(T))` is used for this, where `T` is a list of blocks specifying a goal tower. In order to derive which towers are currently built, the predicate `tower(T)` is defined, which specifies that the list `T` is a tower if the blocks in the list are stacked on top of each other, such that the head of the list is the top block and this top block is clear (defined using the predicate `clear(X)`).

Three action rules are defined. The first two specify that `pickup(X)` or `putdown(X,Y)` can be executed if `goodMove(X,Y)`. The third rule specifies that if the agent is holding a block for which no good move can be derived, the block should be put onto the table. Most of the dynamics cases specified in [16] are handled by the knowledge base, and some are handled by one of the action rules (see [16]).

Program B The knowledge base defines the predicates `clear(X)` and `tower(T)`, where `T` is a list of blocks that are stacked on top of each other. In contrast with the definition of this predicate in A, here the top block of the tower does not have to be clear (i.e., a bottom part of a tower is also a tower). The belief base is empty. The goal base initially contains the goal configuration as a conjunction of `on(X,Y)` atoms. During execution, goals of the form `clear(X)` and `holding(X)` are adopted.

The action rules are divided into three parts: rules for clearing blocks, rules for moving blocks to construct towers, and rules for dealing with dynamics. In addition, there is one rule for selecting the `pickup` action, which can be used either for clearing blocks or for moving blocks to construct towers. The rules for clearing blocks mainly adopt goals: the goal to make a block clear, and on the basis of this goal the agent adopts the goal to hold a particular block and then to put the block on the table. The rules for dealing with dynamics mainly drop goals, or select the action of putting a block down onto the table. These rules explicitly address dynamics cases (2a), (2b), (2c), (2e), and (3a) (see [16]). Case (2d) is handled automatically by adopting a new goal of holding a block, and cases (3b-d) are not handled.

Program C The knowledge base defines the predicates `clear(X)`, `tower(T)` and `above(X,Y)`. The definition of `tower(T)` is the same as in B, and `above(X,Y)` expresses that block `X` is somewhere above block `Y`, but not necessarily on `Y`. As in B, the goal base initially contains the goal configuration as a conjunction of `on(X,Y)` atoms. During execution, goals of the form `do(move((X,Y))` are adopted, to express that block `X` should be moved onto `Y`. The belief base contains one clause for specifying when such a goal is reached (namely when `on(X,Y)` holds).

The action rules are divided into three parts: rules for adopting goals of the form `do(move((X,Y))`, rules for selecting `pickup` and `putdown` actions, and rules for dropping goals of the form `do(move((X,Y))`. The rules for adopting goals both adopt goals in order to construct towers, as well as to move blocks to the table in order to clear other blocks. For each of the actions `pickup` and `putdown` there is one regular rule, and in addition there is a rule for `putdown` for dealing with dynamics cases (2b) and (2e) (see [16]). The rules for dropping goals explicitly address dynamics cases (2a), (2c), and (3a-d). Case (2d) is handled automatically by adopting a new goal of moving a block.

6.2 Similarities and Differences in Structure

In this section, we discuss similarities and differences with respect to the structure of the GOAL programs, as can be found when looking in more detail at the code.

Similarities Our first observation concerns the knowledge base.

Observation 6 (Basic Domain Predicates) *All programs define basic domain predicates in the knowledge base.*

In all three programs, the predicates `clear(X)` and `tower(T)` were defined in the knowledge base, although their definitions vary slightly. The definition of `clear` is needed, since this predicate is used in the action specifications. The `tower` predicate is needed in order to select constructive moves: a constructive move moves a block on top of a partial goal tower. One could view these two predicates as the most basic and essential for the domain, which explains why all programs define them.

An aspect where programs B and C are similar, is the use of dropping of goals.

Observation 7 (Dynamics of Environment) *In both programs that use goals, dropping of goals is used for dealing with the dynamics of the environment.*

If the user moves blocks around, it can be the case that a goal that was adopted in a certain situation, is no longer achievable or should no longer be achieved in the changed situation. This means that the goal should be dropped again. Since A does not adopt goals, it does not have to consider dropping them again if the environment is changed. Another more domain specific way in which the programs handle dynamics, is that they select the action `putdown(X,table)`, e.g., if the agent is holding a block that it does not want to move in that situation.

Another aspect where B and C are similar, is the frequent use of negative goal conditions in the action rules for adopting goals, through which it is checked whether the goal that is to be adopted did not already have an instance in the goal base. In particular, in B there should not be more than one goal of the form `holding(X)` in the goal base, because this goal specifies which block the agent should pick up next. This is achieved by checking whether `not(a-goal(holding(S)))` is the case (where `S` is unbound), before adopting a goal `holding(X)`. Similarly, in C there should not be more than one goal of the form `do(move(X,Y))`, and corresponding negative goal conditions are included in the action rules.

Observation 8 (Single Instance Goals) *In both programs that use goals, there are goals of which at most one instance can occur in the goal base at any one time. This is achieved through the use of negative goal conditions.*

Interestingly, in the Jadex framework the notion of cardinality is introduced [12] to restrict the number of instances of a goal that can be active at the same time. A similar feature may be added to GOAL in order to help the programmer specify these kinds of uses of goals.

Differences One of the main differences between B and C are the goal predicates that are added during execution. B uses `clear(X)` and `holding(X)`, while C uses `do(move((X,Y))`. The goals of B correspond to the most basic predicates of the domain. They do not allow to specify as part of the goal where a block that is or should be picked up, has to be moved. Instead, this is determined on the basis of the goal in the goal base that specifies the goal configuration of the blocks: if the block that an agent is holding can be moved onto a partially build tower, this should be done; otherwise, it should be moved onto the table. In C, a single goal `do(move((X,Y))` represents that the agent should hold `X` if this is not yet the case, and move it onto `Y` if it holds `X`. Since predicates of the form `do(move((X,Y))` are not added to the belief base through percepts, the programmer in this case has to define when such a goal is reached. This is done by adding a corresponding clause to the belief base. Moreover, since B uses action rules to derive the goal to clear blocks as an intermediate step for selecting a goal to hold a block, B uses more rules than C (6, where C uses 2) for specifying that blocks should be moved onto the table in order to clear blocks.

Observation 9 (Abstraction Levels of Goals) *Goals can be used on different abstraction levels. They can correspond to the most basic predicates of the domain, or higher-level goals can be defined. In the latter case, clauses can be added to the belief base in order to define when the higher-level goals are reached.*

In Section 5.2 we have shown that for programs B and C, the portion of adopt and drop actions compared to the total number of actions generated during execution is comparable. However, the number of adopt actions generated by B will most likely grow with the size and complexity of the initial configuration, since in that case many intermediate `clear` goals have to be generated, followed by the goal of holding a block. In C, by contrast, one `move` goal is created for each block that the agent choses to move.

Another difference between B and C is the way in which action rules are clustered. In B, they are clustered according to their function in the strategy (clearing blocks, making constructive moves, and dealing with dynamics), while in C they are clustered according to the type of their consequent (adopt, environment action, and drop). This points to a different style of programming, which may be related to the different levels of goals used in B and C. In C, there is only one type of goal that drives the selection of actions. It is then the job of the programmer to specify when this goal should be adopted, what should be done if the goal is adopted, and when it should be dropped. In B, by contrast, the goals for clearing blocks are selected as intermediate goals, on the basis of which the agent selects the goal of holding a block. Since the goals to clear blocks are thus closely related to the corresponding goal of holding a block, it seems more natural to cluster these rules. The way the rules are clustered in B may also be related to the fact that B was programmed in an incremental way. First a highly non-deterministic GOAL agent was programmed that solved the blocks world problem in a non-efficient way and that did not deal with the dynamics of the domain. Then, rules for dealing with dynamics were added and finally efficiency was addressed. The `tower` predicate was introduced in this last step, and was only used to modify the conditions of the rules for making constructive moves. B was thus developed through refinement (see also Observation 5) of a initial program, where refinement was done both through the addition of rules, as well as by modifying conditions of rules.

Observation 10 (Clustering of Action Rules) *Two ways in which action rules can be clustered are according to the type of their consequent, and according to their function in the action selection strategy.*

One of our motivations for doing this research is to find *code patterns* or typical ways in which the GOAL constructs are used, which can serve as guidelines for programming GOAL agents in various applications domains. Although no general conclusions can be drawn on the basis of our three blocks world programs, the observations that have been made so far in this section already suggest some possible code patterns or typical usages of the GOAL constructs.

A difference between A, compared to B and C, is that A does not use goals in the goal base. Instead, a predicate `isgoal(tower(T))` is used in the knowledge base. Since goals are not explicitly adopted in A, they do not have to be dropped again in case the user moves blocks around. On the other hand, if the goal base is not used, it is the job of the programmer to check which parts of the goal configuration have already been reached, and which have not. If the `a-goal` operator is used in action rules, the semantics of GOAL takes care of this. Also, the GOAL IDE provides a means to inspect the agent's mental state while it is executing, but if goals are used only as part of the knowledge base, one cannot see them when running the agent. Being able to see the agent's goals while it is executing helps in debugging: one can see why the agent is executing a certain action.

A difference between B, compared to A and C, is that B does not deal with dynamics cases (3b-d). This corresponds to the results presented in Section 5.2, which show that B is slightly less efficient than A and C. More action rules would probably have to be added to deal with these cases.

Another difference between the programs concerns their readability. We have asked several subjects somewhat familiar with the GOAL language to comment on the readability of the programs. Program A was found to be the easiest to understand, while the readability of B and C varied across subjects. This seems to be related to Observation 1, which suggests that more of the GOAL constructs are used as more experience is gained. Since all subjects had relatively little experience with programming in GOAL, it seems natural that they find the program making the least use of GOAL constructs easiest to understand. This also suggests that sufficient training is necessary to familiarize programmers with the various Goal— constructs. Another reason why action rules may be difficult to understand, as suggested by some of the subjects, is the relatively high number of belief and goal conditions that each have to be read and interpreted, in order to understand what the action rule is aimed at.

7 Conclusion and Future Work

In this paper, we have analyzed three Goal programs for the dynamic blocks world. We have made several observations based on a quantitative and qualitative analysis. The observations concern the use of Goal constructs, the behavior of Goal programs during execution, and the readability of Goal programs. Based on this analysis of three Goal programs in a specific domain we cannot draw very general conclusions on how Goal programs should be written. However, we do identify important issues in three key areas that need further investigation.

With respect to *use of the* Goal *constructs*, we have identified several similarities and differences across the three programs. Concerning the similarities, it needs to be further investigated whether these can be used as guidelines for programming Goal agents. Where we have identified different possible usages of Goal, it needs to be investigated what the advantages and disadvantages of these approaches are, in order to be able to determine whether and if so, when, they can be used as guidelines. These investigations should lead to a *programming methodology* for Goal.

Regarding *readability*, we have observed that the relatively high number of belief and goal conditions in action rules makes them difficult to understand. We believe it is essential to address this issue in order to make Goal easier to use, and clear code is often argued to be less error-prone. We suggest that the use of macros and a notion of modules to structure action rules such as proposed in [7] may improve readability. In addition, we believe that proper training is essential to allow programmers to make full use of the features offered by Goal.

A third area where more research is needed, is *efficiency*. We have observed that if goals are used, on average a little less than half of the executed actions were actions for adopting or dropping goals. The use of goals thus creates considerable overhead. In the blocks world domain, this is not problematic since the time it takes to execute environment actions is considerably larger than the time it takes to select and execute adopt and drop actions. However, in other domains this may not be the case. Then, it might be necessary to design the program such that a limited number of adopt and drop actions is executed.

In future work, we also aim at refining the approach that we have used for analyzing the programs. For example, it may be useful to obtain data on the level of familiarity of the test subjects with Goal. One way to do this, is to give them a test program and let them suggest which actions the agent might execute next. Also, several subjects indicated that the assignment was quite hard to do. We may have to reconsider our decision to remove all comments from the programs before giving them to the test subjects. Moreover, in a future study we would also like to investigate the use of the debugging facilities of the GOAL IDE.

References

1. L. Astefanoaei and F. S. de Boer. Model-checking agent refinement. In *AAMAS*, pages 705–712, 2008.
2. V. R. Basili and L. C. Briand, editors. *Empirical Software Engineering: An International Journal*. Springer, 2009. `http://www.springer.com/computer/programming/journal/10664`.
3. R. H. Bordini, M. Dastani, J. Dix, and A. El Fallah Seghrouchni. *Multi-Agent Programming: Languages, Platforms and Applications*. Springer, Berlin, 2005.
4. R. H. Bordini, M. Dastani, J. Dix, and A. El Fallah Seghrouchni. *Multi-Agent Programming: Languages, Tools and Applications*. Springer, Berlin, 2009.
5. R. H. Bordini, J. F. Hübner, and M. Wooldridge. *Programming Multi-agent Systems in AgentSpeak using Jason*. Wiley, 2007.
6. M. Dastani. 2APL: a practical agent programming language. *Autonomous Agents and Multi-Agent Systems*, 16(3):214–248, 2008.
7. K. Hindriks. Modules as policy-based intentions: Modular agent programming in goal. In *Proceedings of the International Workshop on Programming Multi-Agent Systems (ProMAS'07)*, volume 4908, 2008.
8. K. Hindriks, C. Jonker, and W. Pasman. Exploring heuristic action selection in agent programming. In *Proceedings of the International Workshop on Programming Multi-Agent Systems (ProMAS'08)*, 2008.
9. K. Hindriks and M. B. van Riemsdijk. Using temporal logic to integrate goals and qualitative preferences into agent programming. In *Declarative Agent Languages and Technologies VI (DALT'08)*, volume 5397 of *LNAI*, pages 215–232. Springer, 2009.
10. K. V. Hindriks. Programming rational agents in GOAL. In R. H. Bordini, M. Dastani, J. Dix, and A. El Fallah Seghrouchni, editors, *Multi-Agent Programming: Languages, Tools and Applications*. Springer, Berlin, 2009.
11. N. J. Nilsson. Teleo-reactive programs and the triple-tower architecture. *Electronic Transactions on Artificial Intelligence*, 5:99–110, 2001.
12. A. Pokahr, L. Braubach, and W. Lamersdorf. A goal deliberation strategy for BDI agent systems. In *MATES 2005*, volume 3550 of *LNAI*, pages 82–93. Springer-Verlag, 2005.
13. A. Pokahr, L. Braubach, and W. Lamersdorf. Jadex: a BDI reasoning engine. In R. H. Bordini, M. Dastani, J. Dix, and A. El Fallah Seghrouchni, editors, *Multi-Agent Programming: Languages, Platforms and Applications*. Springer, Berlin, 2005.
14. Y. Shoham. Agent-oriented programming. *Artificial Intelligence*, 60:51–92, 1993.
15. J. Slaney and S. Thiébaux. Blocks World revisited. *Artificial Intelligence*, 125:119–153, 2001.
16. M. B. van Riemsdijk and K. Hindriks. An empirical study of agent programs: A dynamic blocks world case study in goal [extended version], 2009. `http://mmi.tudelft.nl/~koen/prima09extended.pdf`.
17. M. Winikoff. JACK^{TM} intelligent agents: an industrial strength platform. In R. H. Bordini, M. Dastani, J. Dix, and A. El Fallah Seghrouchni, editors, *Multi-Agent Programming: Languages, Platforms and Applications*. Springer, Berlin, 2005.