An Empirical Study of Patterns in Agent Programs

Koen V. Hindriks[1], M. Birna van Riemsdijk[1], and Catholijn M. Jonker[1]

Delft University of Technology, P.O. Box 5031, 2600 GA, Delft, The Netherlands,
{k.v.hindriks, m.b.vanriemsdijk, c.m.jonker}@tudelft.nl

Abstract. Various agent programming languages and frameworks have been developed by now, but very few systematic studies have 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 contributes to the design of best practices or programming guidelines for agent programming. Following a first empirical study of agent programs written in the GOAL agent programming language for the dynamic blocks world, in this paper we perform a considerably more extensive analysis of agent programs for the first-person shooter game UNREAL TOURNAMENT 2004. We identify and discuss several structural code patterns based on a qualitative analysis of the code, and analyze for which purposes the constructs of GOAL are typically used. This provides insight into more practical aspects of the development of agent programs, and forms the basis for development of programming guidelines and language improvements.

1 Introduction

Shoham was one of the first who proposed to use common sense notions such as beliefs and goals to build rational agents [15], 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]. 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 [18] and Jadex [14]), while for others the focus has been also or mainly on the languages' theoretical underpinnings (e.g., 2APL [6], GOAL [8], and Jason [4]).

In this paper, we take the language GOAL as 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. Starting with small-size applications such as (dynamic) blocks world , the language is being applied more and more in larger domains where agents have to function in *real-time and highly dynamic environments.* To be more specific, recently the language has been used in a project with first year BSc students of computer science, in which groups of students had to program a team of agents to control bots in the first-person shooter game UNREAL TOURNAMENT 2004 (UT2004).

Software engineering aspects become increasingly important as applications get more complex. For this reason, in [16] a first empirical study was presented on how the language constructs are used in practice to program agents, and how easy it is to read the resulting programs with the aim of designing a set of *best practices and programming guidelines* that support GOAL programmers. In that paper, three GOAL programs for the dynamic blocks world domain were studied.

In this paper, we take this a step further and analyze GOAL programs that were developed for UT2004 by the students of the project. This study is much more extensive than [16]: the *application domain* of UT2004 is far more challenging than the dynamic blocks world, which has resulted in much larger programs (approximately 800 lines of code per agent for the larger ones, in comparison with around 100 for the dynamic blocks world); the GOAL *language* has been *extended* significantly since the programs studied in [16] were developed; the *number of available programs* to study is much larger, namely 12 for the UT2004 domain in contrast with 3 for the blocks world domain; the programs are *multi-agent systems*, rather than single agents, which gives us the opportunity to study organization structures as used and understood by students.

The focus is on a *qualitative* study of the code of the agent programs. In addition, we analyze several metrics on the code. Due to the size of the study we do not consider run-time behavior in this paper. We identify and discuss *structural code patterns* for the programming abstractions present in the latest version of GOAL, and analyze for which *purposes* the constructs are typically used. Through this empirical software engineering, we contribute to forming a body of knowledge leading to widely accepted and well-formed theories about engineering GOAL agents.

2 The Agent Programming Language GOAL

In this study, the agent programming language GOAL has been used. GOAL is a high-level language for programming *rational agents* using cognitive concepts such as *beliefs* and *goals*. The language is similar to other agent programming languages such as 2APL, Jadex, and Jason. Due to space limitations, the presentation of GOAL itself here will be very limited and we cannot illustrate all features present in the language. For more information, we refer to [8].

GOAL agents are logic-based agents in the sense that they use a knowledge representation language to represent their knowledge, beliefs and goals to reason about the environment in which they act. The knowledge representation technology we used is SWI Prolog [1]. One of GOAL's distinguishing features is that GOAL agents have a mental state that consists of the *knowledge*, *beliefs* and *goals* of the agent. Knowledge is used to represent conceptual and domain knowledge that is static. During a computation of the agent the knowledge of that agent is never modified. As knowledge is always true, it can be used in combination with both beliefs and goals to derive new beliefs and goals, respectively. For example, if an agent has a conjunctive goal to have a weapon and ammo, and knows that that combination always results in a loaded weapon, it also has the derived goal to have a loaded weapon. The belief base and goal base are the dynamic components of an agent's mental state. Beliefs change by performing actions; Goal also provides two built-in actions `insert(`φ`)` and `delete(`φ`)` to insert and remove information from an agent's belief base. Goals in a Goal agent represent so-called *achievement* goals. An achievement goal is a condition that the agent wants to be true but which is currently not believed to be true by the agent. An achievement goal φ thus never follows from the agent's beliefs (in combination with its knowledge) and this constraint is enforced as a rationality constraint. The rationale is that an agent should not put time and resources into realizing an achievement goal that has already been achieved. This also means that whenever a goal has been (believed to be) *completely* realized, the goal is *automatically* removed from the goal base of the agent. Goal also provides two built-in actions `adopt(`φ`)` and `drop(`φ`)` to, respectively, adopt a new achievement goal and drop some of the agent's current goals. The `drop` action allows an agent to revise its goals in light of, for example, changing circumstances.

Actions are selected by a Goal agent by inspecting their mental state and by means of *rules*. Goal agents are able to inspect their mental state by means of *mental state conditions*. Mental state conditions allow the agent to inspect both its beliefs and its goals, and provide Goal agents with expressive reasoning capabilities. In an agent program, mental atoms of the form `bel(`φ`)` and `a-goal(`φ`)` are used to verify whether φ is believed or φ is an achievement goal.

Actions are selected in Goal by rules of the form **if** $<$ *cond* $>$ **then** $<$ *action* $>$ where $<$ *cond* $>$ is a mental state condition. The $<$ *action* $>$ part may consist of single actions, or of multiple actions that are combined by means of the `+` operator. Rules provide Goal agents with the capability to react flexibly and reactively to environment changes but also allow a programmer to define more complicated strategies. Rules may be located in either the *program section* or the *perceptrule section* of an agent program. In the program section, every cycle of the interpreter a *single* applicable rule is selected and rules in this section are typically used to select actions that are executed in the environment. In the perceptrule section, every cycle of the interpreter *all* applicable rules are executed in order. Rules in the perceptrule section are typically used to process percepts from the environment and messages received from other agents. All built-in actions of Goal may occur in both sections but *user-specified* actions of both internal or environment actions may only occur in the program section. This restriction implies that the number of environment actions executed every cycle is limited to at most one.

Modules provide a means to structure action rules into clusters and to define different strategies for different situations [8]. In particular, modules facilitate structuring the tasks and role assignment of an agent, as it allows an agent to focus on some of its current goals and disregard others for the moment. Different types of modules are distinguished based on whether the module is entered by means of a trigger related to the beliefs or the goals of an agent.

Finally, *mas files* provide a recipe for launching multi-agent systems composed of several Goal agents. A mas file specifies which environment to start and how it should be initialized, which agent source code files are used to create agents, and when to create an agent. An agent may or may not be connected to an environment. In our UT2004 case study agents may be connected to bots; an agent may be launched e.g. when a bot becomes available in the environment. Agents connected to an environment are able to execute environment actions to change the environment and receive percepts from the environment which enables an agent to monitor its environment. Percepts - received every cycle of the interpreter - are stored in an agent's percept base. At the end of each cycle this percept base is cleared again and all percepts are removed. This implies that each cycle all percepts need to be processed immediately.

Additional features of Goal include among others a macro definition construct to associate intuitive labels with mental state conditions to increase the readability of the agent code, options to apply rules in various ways, and communication. Various communication primitives are available but the most basic action is the `send` action to send a message to another agent. Messages that are sent as well as those that are received are archived in the mailbox of an agent, and are only removed when the agent explicitly does so.

3 Experimental Setup

We perform a qualitative study (rather than a quantitative study) since it better fits the aim of this paper, namely to analyze how students use Goal as a step towards developing programming guidelines for Goal. Qualitative methods are used for exploratory research in which hypotheses are formed, while quantitative methods are used to test pre-determined hypothesis and produce generalizable results [12]. Our research is exploratory, since we are in the process of investigating which structural code patterns might be part of programming guidelines for Goal, as examples of recommended or not recommended uses of the language (comparable to design patterns and antipatterns used in software engineering).

In programming language research, several *criteria for good language design* have been identified. The following are particularly relevant in the context of this paper. The value of linear flow of control was, for example, recognized, primarily for its value in program debugging and verification, it was recognized that a language must be comprehensible, so that programs written in the language can be read and maintained, and modular program structures were observed to make an important contribution to the production of large software systems [17]. Moreover, in [10] several language evaluation criteria are distinguished among which: human factors (to what degree does the language alow a competent programmer to code algorithms easily and correctly, how easy is the language to learn), software engineering (maintainability, reusability, etc.), and application domain (how well a language supports development for a specific domain).

In agent research, software engineering has mainly been studied in the context of agent-oriented software engineering methodolgies such as Prometheus [13]. These methodologies, however, are either too abstract to provide programming guidelines for concrete agent programming languages, or, to the extent to which they provide concrete implementation guidance, do not fit the programming abstractions as used in languages like GOAL. In the agent programming field, [11] focuses on structural metrics related to dependencies between abstractions, which among others indirectly predict the likelihood of bugs. This paper can be viewed as complementary to ours.

Subjects The programmers whose code we have analyzed are first-year BSc computing science students who followed our second-semester course on Programming Multi-Agent Systems and the consecutive Project Multi-Agent Systems. These students are the subjects of our experimental research. In the course the students were trained in both Prolog as well as the agent programming language GOAL. As an indication of the level these students had, we briefly provide some observations related to their skills in Prolog which is a prerequisite for writing GOAL agents since Prolog is used as the knowledge representation language in these agents. The Prolog skills demonstrated by students are basic but sufficient. Students were, for example, able to apply negation as failure and recursion.

Project UT2004 is an interactive, multi-player computer game where bots can compete with each other in various arenas. The game provides ten different game types. The game type that was used in the student project is called *Capture The Flag* (CTF). In this type of game, two teams compete with each other that have as main goal to conquer the flag located in the home base of the other team. Points are scored by bringing the flag of the opponent's team to ones own home base while making sure ones own flag remains in its home base. Students have to implement basic agent skills regarding walking around in the environment and collecting weapons and other relevant materials, communication between agents, fighting against bots of the other team, and the strategy and teamwork for capturing the flag. We chose CTF because teams of bots have to cooperate, which requires students to think about coordination and teamwork in a mas.

In the project, students are divided into teams of five students each. Every group has to develop a team of GOAL agents that control three UT bots in the CTF scenario. In the project manual, it was suggested that although the number of bots in the UT environment is three, students can also implement agents that do not control bots in the environment, e.g., for coordination purposes. The time available for developing the agent team was approximately two months, in which each student has to spend about 1 to 1,5 days a week working on the project. At the end of the project, there was a competition in which the developed agent teams compete against one another. The grade is determined based on the students' report and their final presentation.

For the project, an interface was designed that is suitable for connecting logic-based BDI (Belief-Desire-Intention) agents to a real-time game. Such an interface needs to be designed at the right abstraction level. The reasoning typically employed by logic-based BDI agents does not make them suitable for controlling low-level details of a bot. It makes little sense, for example, to require such agents to deliberate about the degrees of rotation a bot should make when it makes a turn. Such low-level control is better delegated to a more behavioral control layer, which was built on top of Pogamut [5]. At the same time, however, the BDI agent should be able to remain in control and the interface should support sufficiently finegrained control. Details on the interface can be found in [9].

Sample In quantitative research, a random and relatively large sample of subjects to study is selected such that results can be generalized to the population of interest. By contrast, in qualitative research the most productive sample to answer the research question is selected, e.g., based on experience or expertise of the subjects. In our case, 12 teams of 5 students participated in the project. The focus of our qualitative analysis is on the code of Teams 1, 2, and 3 who performed best in terms of code and performance in the competition, and Team 12 who performed worst in terms of code and performance.

4 Identification of Patterns

In this section, we present the observations we made by doing a qualitative analysis of the code of our sample. We identify numerous *structural code patterns*, and augment this qualitative analysis with *metrics* concerning, e.g., the number of times certain Goal constructs were used. Also, we analyze for which *purposes* the constructs are typically used. Sections 4.1 to 4.7 each treat a particular language element; sect. 4.8 discusses coordination and mas organization; sect. 4.9 discusses more general software engineering aspects.

4.1 Knowledge and Belief Base

The knowledge base typically was used to define predicates for computing, e.g., distances and other relevant aspects related to navigation. The belief base was used to keep track of the actual state of the environment and typical functions of code in the belief base are to (i) represent global features of the environment (e.g., where is the flag), and (ii) represent assigned tasks or roles (agents were typically assigned a single role or task at any one time). On average the knowledge base was significantly larger than the belief base (23.25 versus 15.67 clauses, with a standard deviation of 24.23 versus 8.7, respectively); moreover, the number of predicates defined in the knowledge base is larger (ranging from 7 to more than 25 predicates) than that in the belief base (about 5) with some exceptions. This suggests that most of the domain logic was located in the knowledge base, in line with its main function to represent conceptual and domain knowledge.

One observation made by inspecting the code of various teams is that this code includes predicates in the knowledge base that have motivational connotations such as `priority` to indicate relative importance and `needItem` and `wants`. The code fragments for defining these predicates are significant portions of the code, sometimes more than a 100 lines of code.

4.2 Goal Base

The use of explicit goals has been limited. On average about 1.13 initial goals were used with a standard deviation of 1.36. By inspection of code, it turns out that initial goals most of the time are abstract goals such as `visitFlags` or even `win`. These abstract goals are not actually used in action or percept rules and are never removed, neither explicitly using a drop action nor implicitly by inserting a belief into the belief base which implies the goal has been achieved. These abstract goals thus are redundant and serve no functional purpose. In 6 out of 12 teams goals are added during runtime by using the adopt action; on average 3.86 adopts are used by these 6 teams with a standard deviation of 4.29.

The goals adopted dynamically are used in context conditions of modules. In these cases, the context condition consists of a check on a single goal which forms the goal of the module, e.g., goal `protectBot` for the module `protector` (Team 3). In these cases, goals are *removed explicitly* (never implicitly) using drop actions (occurring in both action and percept rules). In Team 3, the goal of a module is removed only *after* the module was exited explicitly based on beliefs about role changes. In Team 2, an action rule `if goal( not(camp) ) then exit-module.` is present at the top of the camp module, to express that the module should be exited if the agent no longer has the camp goal. However, this behavior is already in the semantics of GOAL, and thus the rule is redundant. Another observation on the goals used by Team 3 is that some goals could naturally be modelled as *achievement goals* (even though not used as such), while others rather express an *activity over time*. For example, the goal `getFlag` (which expresses an activity) could be replaced by the achievement goal `haveFlag`. In fact, Team 3 uses an action rule to drop the goal `getFlag` if the agent believes `haveFlag`. The goal `protectBot` expresses a behavior that is not so easily transformed into an achievement goal, since it is not clear in which state the agent has "achieved" protecting a bot. Finally, Team 12 has a one-to-one relation between goals and modules where each module corresponds with a different role or task. The use of goals in conjunction with modules and their function is a recurrent pattern in the code that has been analyzed.

We investigated various hypotheses related to the use of goals, built-in goal-related actions, and modules. First, for all teams except Team 6, whenever the code contains occurrences of `drop` actions the code also contained `adopt` actions. The reason that in one agent of Team 6 only one `drop` action was used is that the agent has one goal `start` in the initial goal base that is used to *initialize* the roles of other agents and thereafter is dropped. Second, whenever an `adopt` action occurs it occurs in tandem with `drop` actions. And, finally, occurrences of `adopt` actions entail the presence of modules. The latter suggests that goals have been typically used to implement roles.

4.3 Rules

As explained, rules in a GOAL agent can be placed in the *program* and the *perceptrule* section. The former kind of rules are called *action rules* and are used among others to select actions that are performed in the environment. These rules define the agent's strategy or action selection policy, and determine what the bot that the agent controls will do in the environment. The latter kind of rules are called *percept rules* and are used, among others, to process percepts and messages. Rules can be classified along other dimensions based on their use and in comments in analyzed code we find that rules are used as *communication rules* to send messages, *exit rules* to exit a module, as *mailbox cleanup rules* to cleanup messages stored in an agent's mailbox, etc.

Some examples of patterns observed in rules are:

```
if bel(received(_, role(X)), role(Y))
  then insert(role(X)) + delete(role(Y))
```

This rule inserts an instance of a predicate `role` that has been received via communication and overwrites an old instance of that predicate.

The following rule retrieves the agent's name and communicates the role with the name to all other agents *once*:

```
if bel( me(X) ) then sendonce(allother, navServer(X)).
```

Although the last rule can only be used to select the single `sendonce` action, using the `+` operator multiple actions may be selected simultaneously as illustrated by the second last rule above. This feature allows an agent to execute more than one action in a cycle of the interpreter. All teams make frequent use of the `+` operator to execute multiple actions with one action rule.

The average number of action rules per agent over all twelve teams is approximately 28. The average number for agents that are connected to the environment is 42. The average number for agents connected to the environment for Teams 1, 2 and 3 is 65.5. As action rules determine strategy, this suggests that Teams 1, 2, and 3 have implemented the most elaborate strategies and suggests more strategic programming. This is in line with performance in the competition where Teams 1, 2, and 3 outperformed other teams. The hypothesis that Teams 1, 2, and 3 have coded more elaborate strategies is also corroborated by the fact that the number of percept rules used by these teams is only little above average.

Since goals are used to a very limited extent, the majority of mental state conditions in action rules consists of conditions on beliefs. The number of conjuncts of belief conditions varies, but typically no more than five conjuncts are used. Since most conditions are on beliefs only, never more than one belief operator is used per action rule. This holds for all twelve teams.

Percept rules, i.e. rules in the perceptrule section, are used for several main purposes: processing percepts and messages, sending messages, cleaning up the mailbox, and adoption and dropping of goals (e.g. Team 3). The average number of percept rules per agent over all twelve teams is approximately 51. The average number for agents that are connected to the environment is 69. The average number for agents connected to the environment for Teams 1, 2 and 3 is 78. Note that the number of percept rules overall is higher than the number of action rules per agent. This probably is related to the fact that *all* applicable percept rules are executed in every cycle of the interpreter whereas only *one* applicable action rule is executed in that same cycle. The perceptrule section thus allows to process *all* incoming percepts and *all* received messages. It also facilitates updating mental states in other ways, for example, to adopt a goal when the agent learns the environment has changed.

4.4 Program Section

The program section contains all the action rules, from which exactly one of the applicable action rules is selected for execution. This section comes with the option to evaluate rules randomly or in linear order. When rules are evaluated randomly, a rule is chosen randomly, and the conditions associated with the rule and action(s) are evaluated; in case these conditions hold, the action(s) is executed, otherwise randomly another rule is chosen. Linear order evaluation means that rules are evaluated in order. This type of evaluation is deterministic and potentially ease programming as conditions of rules that have been evaluated but failed can be assumed to be false in rules below these rules. Linear order may provide a programmer thus with a greater sense of control. It turns out that all teams use the option `order=linear` to enforce linear execution of action rules.

The management bot of Team 1 does not have action rules in the program section. All other agents have (functional) action rules in the program section. The number of action rules on top level, i.e., not within modules, is typically small (ranging from 0 to 2 in Teams 1, 2 and 3).

4.5 Modules

Modules facilitate structuring code as well as the behavior of agents and are used by all teams. A module may be entered when an associated context condition holds and thereafter only *action* rules inside the module are executed. A module can be exited automatically or by means of selecting and executing an `exit-module` action. Automated exit of modules works differently for the two types of modules, namely *reactive* and *goal-based* modules. Reactive modules have a context condition that does not check whether goals are present but does inspect the beliefs of the agent; such modules are automatically exited when there are no options anymore to execute an action. Goal-based modules have context conditions that inspect the goal base of an agent and after entering the module focus on goals that satisfy the context condition; such modules are automatically exited when all goals have been achieved. Note that the semantics of exiting a module is built-in but is a delayed effect. That is, exiting may happen after a number of cycles of the interpreter that is not easily predicted.

Teams 1, 2, and 12, who make use of a management agent, have significantly fewer (sub)modules for this agent (0, 1, and 0 respectively) than for the agents that are connected to bots (13, 7, and 4, respectively). The average number of (sub)modules used in the agents of all twelve teams is approximately 3. Although a module may contain the same sections as a GOAL agent except for the perceptrule section, often, only the program section is used in modules.

Modules are used to encapsulate behavior for *roles or (high-level) tasks*. For example, Team 2 distinguishes the modules defender, assault, bodyguard, flagcarrier, and hunter on top level, which form the roles as indicated by corresponding context conditions such as `bel ( role(defender) )`. Team 1 distinguishes capture, defend, attack, and waitAtEnemyBase, which form tasks as indicated by corresponding context conditions such as `bel(task(capture(_)) )`.

If submodules are used, they are used one level deep, i.e., a module within a module. Team 1 makes frequent use of submodules (1 to 3 per top level module) and Team 2 uses one submodule (`camp` as a submodule of `defender`). Teams 3 and 12 do not make use of submodules.

Several patterns can be observed concerning strategies for *entering and exiting modules*. The context condition usually consists of a single belief or goal condition, expressing the *task* (Team 1 uses, e.g., `bel(task(capture(_))` and similarly for other modules) as the context condition for the module `capture`), the *role* (Team 2 uses, e.g., the context condition `bel ( role(defender) )` in the module `defender` and similarly for other modules), or the *goal* of the module (Team 3 uses, e.g., the context condition `a-goal ( getFlag )` in the attacker module and similarly for other modules). Teams 1, 2, 3 and 12 use the `exit-module` action to explicitly specify when to exit the module. Modules typically start with such an action rule, which has as the condition the negation of the context condition of the module, e.g., Team 2 uses `bel( not(role(defender)) )` in the defender module where the context condition is `bel ( role(defender) )`. Sometimes, additional action rules for explicitly exiting modules are introduced. For example, Team 1 uses rules that allow the agent to exit the module because it has a more important task (if the agent sees an item it needs, it will get it and afterwards continue).

Interestingly, Team 6 uses modules for initialization purposes. Their management agent uses a single goal `start` which is present in the initial goal base of that agent to enter a module that contains some initialization code; after executing that code the initial goal `start` is dropped and the module is exited. (Recall that Team 6 also is the only team that has an agent with a `drop` action without an `adopt` action; this explains why.)

4.6 Actions specification

The action specification section needs to contain specifications for all actions that are used in the agent program but not built-in into GOAL. Such actions are called *user-specified* actions, and can be actions with effects only on the mental state, called *internal actions*, as well as actions which also change the environment, called *environment actions*. In principle there is no need to introduce *internal actions* as whatever can be achieved with such actions can be achieved with the built-in actions of GOAL but introducing such actions may increase readability.

Concerning *internal actions*, i.e., actions that are not executed in the environment, we observe that only Teams 1, 2 and 4 have used these. Team 1 only implements a dummy `nothing` action. Teams 2 and 4 implement internal actions only in the management bot which is not connected to the environment.

All agents that are connected to the environment contain action specifications for *environment actions*. The interface to the UT2004 environment made available in the student project [9] provides 9 different actions with a range of different parameters to select from. Actions, without mentioning parameters, include, for example, `selectWeapon`, `goto`, `pursue`, `lookAt`. On average the `goto` and `halt` actions are used 23 times versus 13 times that other actions are used.

The `goto` and `halt` actions thus are used about 4 to 5 times more often than other actions. This suggests that navigational issues are dominant in the project.

In action specifications, we make several observations concerning the use of *pre- and postconditions* in environment actions. First, we can distinguish actions for moving around in the environment, namely `goto`, `pursue`, `halt` and `respawn`, from other actions such as `selectWeapon`. For moving actions, Teams 1, 2, and 3 use pre- and postconditions that express how to change the agent's moving state. The moving state is expressed by all three teams as `state(moving(Route))`, `state(pursue)`, or `state(reached([])`. This is related to the fact that moving actions are typically *durative* (except for the `halt` action), and it needs to be recorded whether the agent is currently executing such an action. For *instantaneous* actions, postconditions typically express the (immediate) effect of the action, such as the current weapon for `selectWeapon` (Teams 2 and 3), or the postcondition `true`, in which case percepts are used for observing the effect of the action in the agent's next reasoning cycle (Team 1).

4.7 Communication

Plain communication in which send actions of the form `send(A,Proposition)` are used is distinguished from *advanced* communication with mental models in which actions of the form `send(A,:Proposition)`, `send(A,!Proposition)`, `send(A,?Proposition)` are used. Mostly plain communication is used. Team 3 uses a few messages with `:`, e.g., `send( allother, :myTeam(MyName, MyRole) )`. The management agent of Team 1 uses a few instances of messages with `!`, e.g., `send(Bot, !task(capture(return)))`, to tell other agents what to do.

Two main ways of *handling received messages* can be distinguished. The first is by *preprocessing* messages using percept rules, which insert the received information into the belief base and delete the received message. The following pattern for preprocessing messages is used by Teams 1 and 3, and the agent connected to the environment of Team 2.

```
if bel(received(A,Proposition))
then insert(Proposition) + delete(received(A,Proposition))
```

The second is by using the received messages directly in conditions of action rules to select the next action (the management agent of Team 2), without preprocessing them. Team 2 also uses the `received` predicate in the knowledge base of the management agent. The first method yields better readable code because action rules and knowledge base are not cluttered with `received` predicate, and allows reasoning with the added propositions using the knowledge and belief base. The second method may have efficiency benefits since no preprocessing is needed, and is simpler since no preprocessing rules have to be written.

4.8 Coordination and MAS Organization

The *organisation structures* chosen by the students were hierarchical and network [7]. Irrespective of the organisation structure the teams used roles (or tasks) to differentiate in behaviour and let the bots change their behaviour over time, with the exception of Team 11. Team 11 had a static role division over the bots. Team 7 uses a bit of a mixture; two of their bots have to change roles depending on the game state, the third always has to defend the flag.

The hierarchical models all consist of one management agent and three team member bots, where the team members were just copies of each other. The bots in the teams using a network organisation (Teams 3, and 11) did not collectively deliberate about strategy and tactics. Each bot decides for itself when to switch roles and only informs the others of its new role. In the hierarchical teams the management agent gets progress information from the team member bots and on the basis of that information decides on role changes for the bots.

The initialisation differed at bit over the teams. Some had the management agent assign the roles arbitrary over the bots (e.g., Team 12), some initially gave the bots a kind of `nothing` role (e.g., Team 1), some intially gave each of the bots a specific active rol like defender, attacker (e.g., Team 3), and Team 11 used three differently coded bots (an attacker, a defender and a support bot).

The roles and their number in different teams vary. The smallest number of roles used is two: attacker and defender (Team 5). Some introduced three roles: hunter, defender, and supporter. Typically, however, a bit more variation was used, as for example by Team 2 who used: attacker, bodyguard, defender, flagcarrier, hunter, and none. The more roles, the more rules were defined to switch between behaviours, and in general the more sophisticated the code to determine the expected behaviour for the various roles.

4.9 Human Factors & Software Engineering

We make several observations concerning human factors and software engineering, in particular with respect to readability, maintainability, and reusability.

We observe that none of the teams have used *macros*. Readability of mental state conditions in rules might have been improved by the use of macros, since the number of conjuncts in these conditions can become relatively large (see Section 4.3). A large number of conjuncts can make it difficult to grasp what is expressed by the condition. Macros may not have been used because they received little attention in the lectures preceding the project, since their definition and meaning is relatively simple. Another reason may be related to the fact that the students used only one belief operator per rule. This may make it less natural to use macros, since one might expect that multiple macro definitions would be used to replace belief conditions with many conjuncts. This would then require the use of multiple macros in rules, instead of using a single belief condition.

Another observation related to human factors and software engineering is that we found frequent occurrences of *duplicate code*. The most notable example was found in the code of Team 3, which coded two agent files that are almost exact duplicates (lines of code = 884). The only difference seems to concern the initial role of the agents. Duplicates are undesirable since it makes it more difficult to understand resulting programs (readability), as it is often not easy to identify the differences between very similar pieces of code. Also, it has a negative influence on maintainability, since changes have to be duplicated too.

Further, we observe that Team 1 uses *hardcoding of agent names* both in the manager agent as well as in the agent program that is used to launch agents that are connected to a bot in the environment. This introduces dependencies between these files which are hard to maintain as, for example, such hardcoding makes it difficult to extend or reduce the number of agents launched in a mas file. Reducing the number of agents would cause runtime errors (as messages are being sent to agents that do not exist) and extending the number of agents would decrease the functionality of these new agents as messages will never be sent to these additional agents. An example of the use of hardcoded agent names is the following. In the agent program that is connected to the environment, percept rules are used to store information about the environment in the belief base, and to send this information to the manager agent. The information sent to the manager agent is divided over the other agents, yielding the following patterns for percept rules, where `zombieA` is the name of an agent connected to the environment, and `godMother` is the name of the manager agent:

```
if bel (me(zombieA), percept(<Percept>))
   then insert(<Percept>) + send(godMother, :<Percept>)
if bel (not(me(zombieA)), percept(<Percept>) )
   then insert(<Percept>).
```

5 Discussion

Explicit Control Several of our observations suggest that programmers prefer *explicit control* over *built-in semantics with delayed effects.* In particular, determinism (by selecting *linear rule order* evaluation, Section 4.4) is preferred over non-determinism (random action option selection). This is related to linear flow of control, which has been proposed as a criterion for good language design (see Section 3). Another well-known paradigm of computing that involves non-determinism is concurrent programming. Non-determinism in concurrent programming stems from the fact that it is unknown how much of one process is executed during the time another one executes an instruction. Interestingly, high-school students of concurrent programming were found to avoid using concurrency [2]. Another observation related to explicit control is that explicit strategies for exiting modules were programmed using the `exit-module` action, rather than relying on the automatic exit mechanisms of the language (see Section 4.5). Also, goals were not used as often as could have been. What's more, if goals *were* used, automatic goal deletion upon achievement was not exploited, since corresponding beliefs were never added to the belief base.

We conjecture that these findings are on the one hand due to an *inherent preference for explicit control*, and on the other hand due to *lack of understanding* of these mechanisms. Exam results indicate that students were more competent in explaining and/or applying action rules, action specifications, linear rule order option and basic Prolog than they were able to do so for modules and subtle differences between communication primitives (`send` versus `sendonce` command). Scores on questions related to the former were significantly higher than those related to the latter. Moreover, the use of explicit module exit strategies in cases where use of built-in mechanisms would have been simpler, also suggest a lack of understanding. To some extent, lack of understanding of the nature of achievement goals is indicated by the fact that corresponding beliefs are never inserted into the belief base, but more research is needed to explain the code fragments in some agent programs related to motivational notions in the knowledge base instead of the goal base. These findings provide valuable input for teaching the language, since it suggests more time needs to be devoted to explaining and practicing with the features of GOAL that have built-in semantics with delayed effect. In particular, programming examples and patterns will have to be developed to demonstrate possible uses of the language.

A possible *pattern for using modules*, derived from the observations and discussion above, is the following. For each role that the agent should be able to take, create a module with the goal of the module as the context condition. If the goal of the module is adopted, the agent can enter the module to perform the corresponding role. The program rules of the module should aim at achieving the goal of the module. If the goal is reached, the agent will automatically exit the module. If the agent should no longer pursue the goal because, e.g., more important goals should be pursued, percept rules can be used for specifying when the goal should be dropped, in which case the agent would also exit the module automatically. It is important to specify such *goal revision policies*, due to incomplete information and incomplete control over the environment. New observations of or changes in the environment may cause an adopted goal to become obsolete, requiring the need for specifying when the goal should be dropped. A similar observation about dropping of goals being used for dealing with dynamics of the environment was made in [16].

Language Design The idenfication of patterns has yielded not only insights on how GOAL constructs are (to be) used, but also gives rise to multiple possibilities for *language improvement* and further investigation of *language design choices*. For reasons of space, we briefly discuss some of them.

Mailbox clean-up as performed in percept rules suggests investigation of whether keeping `received` and `sent` messages by default in the mailbox is to be preferred over cleaning up the mailbox in every cycle. This can be done by introducing these modes as an option in an agent program. In this way, we can find out by experience and practice what is preferred by the programmer.

One of the difficulties of continuous language design is to monitor whether code parts keep providing useful functionality throughout the changes that are made to the language. For example, the GOAL syntax requires agent files to provide an agent name. However, this agent name is just a label at the top of an agent file which is never used as the functionality of naming and making agent names public has been delegated to the mas file. Using these labels in agent files thus only creates confusion and it is better to remove these agent names.

Similarly, early requirements on syntax may not be so useful anymore as the language is extended. In particular, after introducing the perceptrule section the requirement to have at least one action rule in the program section seems not as useful anymore (Team 1 introduced a trivial 'obligatory' rule in the program section in their management agent). We plan to remove this requirement and allow an empty program section, and only generate a warning at parse time.

We will consider the introduction of warnings and automatic dependency analysis and checks: check on whether goals can ever become beliefs of the agent (to indicate proper use of achievement goals); check for single `send` actions in the program section, since these could just as well have been added in the percept rules; automated support for dependency analysis to identify duplicate code, etc. Also, support will have to be added to prevent duplicate code, e.g., by providing import and extension functionalities.

6 Conclusion

In this paper, we have studied Goal programs that were written by first year computer science students for the domain of UT2004. This study is far more extensive than a previous study of Goal programs for the dynamic blocks world. It has provided insights into how students use Goal to program agent teams for a real-time dynamic environment. Overall, we can conclude that Goal and the interface that was provided between Goal and UT2004 allow students to program multi-agent systems in which high-level team strategies are used, in combination with navigation and interaction with the virtual environment.

Our analysis has identified patterns that seem to be very useful, such as the use of modules to implement agent roles; patterns that indicate a preference for explicit control and lack of understanding of implicit built-in semantics, such as use of the `exit-module` action to explicitly exit modules; patterns that suggest improvements to the language are needed, such as the frequent occurrence of duplicate code; patterns that require further analysis, such as the use of preprocessing of received messages versus direct use of messages, and the limited use of goals. One issue that is hard to disentangle is whether problems we identified in the source code are due to programming skills and teaching effort, or rather due to the design and semantics of the language studied. To deal with this issue, here we have tried to establish by looking at exam results, for example, if code practices could be related to skills. More research is needed to get a better grip on this issue, however. It remains to be established, for example, why students use the knowledge base in ways not envisaged at design time.

Through this analysis, we have come closer to the development of best practices and programming guidelines for Goal, we have identified aspects that can be improved in the language, and we have gained a better understanding of which aspects of the language are easy to use and which are more difficult to grasp. A better understanding of problems that programmers face when using the language will help us make better debugging and development software. Note also that some of our main findings seem applicable to other agent programming languages as well. E.g. the use of modules to program roles has also been suggested elsewhere [3]. Our method and the results obtained may extend in particular to languages such as 2APL and *Jason* as the components in these languages are similar in many respects, but, of course, more research is required.

In future work, we plan on improving GOAL along the lines suggested in this paper, using the identified patterns to improve teaching of how to use GOAL and studying the effects of this, and further investigating the hypotheses formed through our analysis, e.g., concerning the reasons for the use of explicit control rather than built-in semantics.

References

1. SWI Prolog. `http://www.swi-prolog.org/`.
2. M. Ben-Ari and Y. Ben-David Kolikant. Thinking parallel: The process of learning concurrency. In *Fourth SIGCSE Conference on Innovation and Technology in Computer Science Education*, pages 13–16, 1999.
3. Rafael H. Bordini, Mehdi Dastani, Jürgen Dix, and Amal El Fallah Seghrouchni. *Multi-Agent Programming: Languages, Tools and Applications.* Springer, 2009.
4. Rafael H. Bordini, Jomi Fred Hübner, and Michael Wooldridge. *Programming Multi-agent Systems in AgentSpeak using Jason.* Wiley, 2007.
5. Ondrej Burkert, Rudolf Kadlec, Jakub Gemrot, Michal Bída, Jan Havlíĉek, Martin Dörfler, and Cyril Brom. Towards fast prototyping of IVAs behavior: Pogamut 2. In *Proc. of IVA'07*, 2007.
6. Mehdi Dastani. 2APL: a practical agent programming language. *JAAMAS*, 16(3):214–248, 2008.
7. Virginia Dignum. *A Model for Organizational Interaction: Based on Agents, Founded in Logic.* PhD thesis, 2004.
8. Koen V. Hindriks. GOAL Programming Guide. `http://mmi.tudelft.nl/~koen/goal`, 2010.
9. Koen V. Hindriks, M. Birna van Riemsdijk, Tristan Behrens, Rien Korstanje, Nick Kraaijenbrink, Wouter Pasman, and Lennard de Rijk. Unreal GOAL agents. In *Proc. of AGS'10*, 2010.
10. James Howatt. A project-based approach to programming language evaluation. *ACM SIGPLAN Notices*, 30(7):37–40, 1995.
11. R. Jordan Howell and Rem Collier. Evaluating agent-oriented programs: Towards multi-paradigm metrics. In *Proc. of ProMAS'10*, pages 63–79, 2010.
12. Martin N. Marshall. Sampling for qualitative research. *Family Practice*, 13(6):522–525, 1996.
13. Lin Padgham and Michael Winikoff. *Developing Intelligent Agent Systems: A Practical Guide.* Wiley Series in Agent Technology. John Wiley and Sons, 2004.
14. Alexander Pokahr, Lars Braubach, and Winfried Lamersdorf. Jadex: a BDI reasoning engine. In *Multi-Agent Programming.* Springer, Berlin, 2005.
15. Yoav Shoham. Agent-oriented programming. *Artificial Intelligence*, 60:51–92, 1993.
16. M. Birna van Riemsdijk and Koen V. Hindriks. An empirical study of agent programs: A dynamic blocks world case study in GOAL. In *Proc. of PRIMA'09*, volume 5925 of *LNAI*, pages 200–215. Springer, 2009.
17. Anthony I Wasserman. Issues in programming language design— an overview. *SIGPLAN Notices*, 1975.
18. Michael Winikoff. JACK^{TM} intelligent agents: an industrial strength platform. In *Multi-Agent Programming: Languages, Platforms and Applications.* Springer, 2005.