An Empirical Study of Cognitive Agent Programs

M. Birna van Riemsdijk and Koen V. Hindriks and Catholijn M. Jonker

EEMCS, Delft University of Technology, Delft, The Netherlands
{m.b.vanriemsdijk,k.v.hindriks,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 elements in these languages may be and are in fact used in practice. Performing a study of these aspects contributes to the design of instruments for facilitating development of high-quality agent programs, namely programming language, programming guidelines & teaching methods, and development environment. In this paper we propose an approach for empirically studying how programmers use a programming language, in which we identify several analysis dimensions. We perform two case studies in which we analyze agent programs written in the GOAL agent programming language along the identified dimensions. The case studies concern programs for the dynamic Blocks World and for controlling bots in the first-person shooter game UNREAL TOURNAMENT 2004. We evaluate our experimental setup and discuss to what extent our findings generalize to other cognitive agent programming languages. This provides insight into more practical aspects of the development of agent programs, and forms the basis for improvement of instruments for facilitating agent development.

1 Introduction

Shoham was one of the first who proposed to use common sense notions such as beliefs and goals to build rational agents [40], 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 [8, 9]. 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 [50] and Jadex [37]), while for others the focus has been also or mainly on the languages' theoretical underpinnings (e.g., 2APL [12], GOAL [22], and Jason [10]).

In this paper, we take the language GOAL as object of study (Section 2). 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

[33], the language is being applied more and more in larger domains where agents have to function in *real-time and highly dynamic environments*.

As these applications get more complex, it becomes increasingly important that the agent programmer is supported by a comprehensive set of instruments to facilitate the development of high-quality agent programs. This set of instruments should consist of three strongly interrelated elements: *programming language, programming guidelines & teaching methods, and development environment*. In this paper we take a step towards improving on these three elements by studying how programmers use the GOAL language. Our observations in this study have implications for all three instruments.

We propose an approach for empirically studying how programmers use a programming language, in which we identify several analysis dimensions (Section 3). The focus is on a *qualitative* study of the code of GOAL programs. We perform two case studies in which we analyze GOAL programs along the identified dimensions: programs for the dynamic Blocks World in Section 4 and for controlling bots in the first-person shooter game UNREAL TOURNAMENT 2004, or UT2004 for short in Section 5. We evaluate our experimental setup and discuss to what extent our findings generalize to other cognitive agent programming languages in Section 6, after which we conclude the paper (Section 7). Through this empirical software engineering, we contribute to forming a body of knowledge leading to widely accepted and well-formed theories (cf. [5]) about engineering GOAL agents in particular, and, at least to some extent, for developing cognitive agents more generally.

Parts of this paper have been published in [45] (Case study 1) and [25] (Case study 2). In this paper we propose a general approach for the empirical study of agent programs, and analyze the results of the case studies according to this approach. Moreover, we provide an extensive evaluation of the experimental setup and discussion of generalization to other agent programming languages.

2 The Agent Progamming 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*. GOAL facilitates programming multi-agent systems and uses an environment interface called EIS to connect to environments [6]. The language is similar to other agent programming languages such as 2APL, Jadex, and Jason. A comprehensive overview of related agent programming languages can be found in [8, 9]. We present those features of GOAL relevant for our purposes here. For more detailed information on GOAL, we refer to [22, 26]. At the time of writing, GOAL is being modified based on some of the results presented in this paper, which would explain any differences a reader may note between the currently distributed version of GOAL and the presentation of GOAL below; we refer to the reader to the website `http://mmi.tudelft.nl/trac/goal`.

GOAL agents are logic-based agents in the sense that they use a knowledge representation language to represent their knowledge, beliefs and goals to rea-

```
main: towerBuilder {
  knowledge{
    % assume there is enough room to put all blocks on the table
    clear(table).
    clear(X) :- block(X), not(on(Y,X)), not(holding(X)).
    clear([X|_]) :- clear(X).

    above(X,Y) :- on(X,Y).
    above(X,Y) :- on(X,Z), above(Z,Y).

    tower([X]) :- on(X,table).
    tower([X,Y|T]) :- on(X,Y), tower([Y|T]).
  }
  beliefs{
     block(a). block(b). block(c).
     on(a,table). on(c,b). on(b,table).
  }
  goals{
     on(a,b), on(b,table).
  }
  program{
     % a block X is obstructing if it prevents moving a block Y in position
     #define obstructingBlock(X) a-goal( on(Y,Z) ), bel( above(X,Z); above(X,Y) ).
     % moving X on top of Y is constructive if that move results in X being in position.
     #define constructiveMove(X,Y) a-goal( tower([X,Y|T]) ),
               bel( tower([Y|T]), clear(Y), (clear(X) ; holding(X)) ).
     % a block is *in position* if it achieves a goal.
     #define inPosition(X) goal-a( tower([X|T]) ).

     ...
     if constructiveMove(X,Y) then putdown(X, Y).
     if bel(holding(X)), a-goal(on(X,table)) then putdown(X, table).
     ...
  }
  perceptrules{
    % assumes full observability
    if bel( block(X), not(percept(block(X))) ) then delete( block(X) ).
    if bel( percept(block(X)), not(block(X)) ) then insert( block(X) ).

    if bel( holding(X), not(percept(holding(X))) ) then delete( holding(X) ).
    if bel( percept(holding(X)), not(holding(X)) ) then insert( holding(X) ).

    if bel( on(X,Y), not(percept(on(X,Y))) ) then delete( on(X,Y) ).
    if bel( percept(on(X,Y)), not(on(X,Y)) ) then insert( on(X,Y) ).
  }
  actionspec{
    pickup(X) {
      pre{ clear(X), not(holding(Y)) }
      post{ true }
    }
    putdown(X,Y) {
      pre { holding(X), clear(Y) }
      post { true }
    }
    ...
  }
}
```

Table 1. Example GOAL Agent for Blocks World

son about the environment in which they act. The knowledge representation technology we used is SWI Prolog [3]. GOAL agents have a mental state that consists of the *knowledge*, *beliefs* and *goals* of the agent. Knowledge is used to

represent static, general domain knowledge. In the Blocks World, for example, the knowledge base may contain the definition of the predicate `clear(X)`, as illustrated in Table 1. During runtime the knowledge of an agent is never modified. As knowledge is assumed to be always true, it can be used in combination with both beliefs and goals to derive new beliefs and goals, respectively. In the Unreal Tournament environment, 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. In the Blocks World environment, the belief base may contain information about which blocks are present and how they are stacked, represented using the predicate `on(X,Y)`. 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 means that whenever a goal has been (believed to be) *completely* realized, the goal is *automatically* removed from the goal base of the agent.

Various operators are available to inspect an agent's mental state. The `bel(`φ`)` operator is used to inspect an agent's belief state. The condition φ is evaluated as a Prolog query on the knowledge and belief base. 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. The operator `goal(`φ`)` is used to inspect the goal base and 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:

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

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.

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 [44]. They have a true postcondition, and therefore do not update the belief base. Since these actions take time, the results of their execution is incorporated into the beliefs through percepts, rather than through postconditions of the actions.

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. Beliefs can be changed by performing two built-in actions `insert(`φ`)` and `delete(`φ`)` to insert and remove information from an agent's belief base. 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 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. The `drop` action allows an agent to revise its goals in light of, for example, changing circumstances. It may, for example, not be opportune anymore to collect the opponent's flag in a capture the flag scenario if that opponent managed to steal our own flag. A GOAL agent inspects and modifies its state at runtime analogously to how a Java method operates on the state of an object. Agent programming in GOAL therefore can also be viewed as *programming with mental states*.

Actions are selected by a GOAL agent by means of *rules* of the form **if** $<$ *cond* $>$ **then** $<$ *action* $>$, where $<$ *cond* $>$ is a condition on the mental state of the agent and $<$ *action* $>$ is the action that can be selected for execution if the condition holds. In the Blocks World, for example, an agent needs to know what the current configuration of blocks is and needs to have basic knowledge about such configurations (e.g., when a block is considered to be clear) to make a good decision. 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 reason-

ing capabilities. In essence, writing such conditions means specifying a *strategy* for action selection that will be used by the GOAL agent. 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. 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 [22]. 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.

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. In the student project, GOAL agents were used to control three UT2004 bots on different maps. Agents did not have to be connected to the UT2004 environment which allowed students to design multi-agent systems with more than three agents, including, for example, a management agent for coordination purposes.

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. 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. This interface is defined using the environment interface mentioned above [6]. 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, if considered relevant by the agent('s designer). Each time after a GOAL agent has completed one reasoning cycle, 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 can be found in Table 1. In the Blocks World, for example, percepts 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.

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. Example macro definitions are specified in the *program* section of the example agent in Table 1. 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 Research Approach

In this section we outline an approach for conducting research in empirical software engineering for agent programming. Future research will be needed to further refine it.

3.1 Qualitative Research in Empirical Software Engineering

Research into agent programming frameworks is largely formulative, in the sense that new language constructs and specification techniques are proposed and formally analyzed. This is important for establishing solid foundations for the area of agent programming language research. Now that significant parts of these foundations have been established, we propose that in order to advance we need to take a more *empirical* perspective in which we do systematic empirical studies on how our languages and platforms are in fact used in practice. Based on these findings we can make improvements to the languages and frameworks themselves, to their IDEs, and to the guidelines for using them. We believe that the foundations and programming environments are now well-enough developed to take that step. This is not to say that foundational research is no longer important, but we argue that *in addition* to that research it is important to also take an empirical perspective in order to advance the state-of-the-art.

One can distinguish two broad ways of conducting empirical research: using a *qualitative* or using a *quantitative* approach. The two approaches have fundamentally different aims. As Marshall states in a paper on sampling for qualitative

research [32]: "The aim of the quantitative approach is to test pre-determined hypotheses and produce generalizable results. Such studies are useful for answering more mechanistic 'what?' questions. Qualitative studies aim to provide illumination and understanding of complex psychosocial issues and are most useful for answering humanistic 'why?' and 'how?' questions." Qualitative research is often used for the investigation of social phenomena, i.e., processes in which people are involved, but is also used more and more in the context of software engineering (see, e.g., [17]).

Considering our research question, namely studying *how programmers use an agent programming language* (GOAL in this case), we argue that a qualitative approach is most suitable. We do not yet have clearly defined hypotheses stating, e.g., that one programming pattern is better than another. Rather, the research is exploratory and aimed at identifying possible hypothesis and areas for improvement of the various development instruments. This seems natural, given the small number of empirical studies that have been done in the area of agent programming languages.

3.2 Research Process

In this section we outline a basic step-wise approach for conducting this kind of empirical research in agent programming languages. The steps are a simplification of the process proposed in [16] for building theories from case study research, and are specialized for the particular context of agent programming language research.

1. **Getting started**: The main task in this phase is formulation of a research question, without specifying hypotheses or a theory. In our case, our overall research question is "how do programmers use GOAL"? Answering this question then should form the basis for improving how programmers use GOAL, and improving support for this in the form of the language itself, the IDE and programming guidelines.
2. **Selecting cases**: 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 we selected subjects with different levels of GOAL programming skills, so that we were able to compare the way in which less competent GOAL programmers use the language with the way in which more skilled programmers use it. More details are provided when discussing the case studies in the next sections.
3. **Data collection**: Different data collection methods may be combined, such as interviews, questionnaires, observations, and results of executing the task of interest. In our case, the most important source of data that we considered was the latter, namely the agent programs themselves. In future work we plan to combine this with other sources of data. The advantage of combining different sources is that it allows to investigate the same phenomenon from

different perspectives. For example, in our analyses of the agent programs it was often not clear what the cause of certain observations was, e.g., whether this was related to lack of understanding or issues in the language itself. Interviews may provide more insight into this.

4. **Data analysis**: Two broad analysis techniques may be considered, namely within-case analysis and cross-case pattern search. The first allows to gain familiarity with the data and preliminary hypotheses formulation, the second requires researchers to look beyond initial impressions and see observations from different perspectives. In our study the emphasis is on within-case analysis, but in the conclusion we also relate the finding from the two cases and perform a preliminary cross-case analysis. A difficulty in conducting the latter was that our two cases differed from a relatively simple single-agent domain to a highly-dynamic, real-time multi-agent domain. In future work more case studies of both categories would have to be conducted to identify similarities and differences across cases but within domains. Concerning the types of aspects to consider when analyzing the data, we provide more concrete guidelines in Section 3.3.
5. **Shaping hypotheses**: The process of shaping hypotheses from the analysis of data is iterative, i.e., formation of overall impression and tentative hypotheses is alternated with taking a closer look at the data and assessing to what extent these hypotheses fit. In our study, the emphasis is on data analysis and extraction of possibly interesting observations, e.g., concerning how programmers use constructs like modules. Generally speaking, hypotheses related to such observations could stipulate that using the construct in a certain way is beneficial in agent programming, e.g., when it comes to understandability, maintainability, etc. Alternatively, if we observe a pattern that we expect is not beneficial, a similar hypothesis could be formed. Also one can form hypotheses about the reason for observing a certain phenomenon. We suggest such hypothesis in various places, but the focus of this study is on extracting interesting observations.

3.3 Analysis Dimensions

We propose several dimensions along which to analyze how programmers use GOAL. These dimensions are general enough to be applied to other (agent) programming languages.

1. A *functional analysis* which identifies what the available language constructs are used for, and which general principles are applied when using them.
2. A *structural analysis* which identifies structural code patterns, and which determines quantitative metrics on the code.
3. An *analysis of software quality* along standard software quality measures like maintainability, reusability, readability, etc.
4. An *analysis of run-time behavior* which identifies efficiency issues, and which shows how often certain parts of a program are executed when running the agent.

The third dimension is closely related to the first two, as certain functional or structural usages of the language may influence software quality. Good programming guidelines & teaching methods can improve software quality by improving on functional and structural usages of the language. Software quality can also be improved by improving the programming language to facilitate certain usages of the language.

Existing research related to these dimensions is the following. In programming language research, several criteria for good language design have been identified. 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; modular program structures were observed to make an important contribution to the production of large software systems [47]. Moreover, in [27] several language evaluation criteria are distinguished among which: human factors (to what degree does the language allow 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 [35]. 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, [29] focuses on structural metrics related to dependencies between abstractions, which can be used to indirectly predict the likelihood of bugs. This paper can be viewed as complementary to ours.

4 Case Study 1: Dynamic Blocks World

In this section we present the result of our first case study, in which we examine GOAL programs for the so-called dynamic Blocks World. In Section 4.1 we present the dynamic Blocks World domain, and in Section 4.2 we explain which subjects we used for this study. We present a structural analysis (in particular analysis of quantitative metrics on code) in Section 4.3, a functional analysis in Section 4.4, an analysis of software quality (in particular readability) in Section 4.5 and an analysis of run-time behavior in Section 4.6. Some of the more novel features of GOAL were not yet available when we performed this case study, in particular modules and macros. Consequently, the use of these constructs was not evaluated in this study.

4.1 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., [42]) 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 [33], 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" in the sense that all blocks fit on the table). 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 [44].

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

4.2 Subjects and Programming Assignment

We have asked three subjects (two researchers and one programmer) to program a GOAL agent for the dynamic Blocks World. We refer to the resulting programs as A, B, and C. The person who programmed A had the least experience with GOAL, while the programmer of C had the most. Their experience in programming in GOAL ranged from writing simple programs used for trying out the language constructs (A) to writing several programs for relatively simple single agent domains (B) and simple multi-agent domains (C). None of them had used the language in large domains like UT2004 (see Section 5). All subjects were somewhat familiar with the Blocks World domain.

As a starting point, they were given the action specification and percept rules of Table 1. 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 [44]; all test cases and results can be found via the link provided with this reference. After the programs were handed in for analysis, they were not modified anymore.

4.3 Structural Analysis: Quantitative Metrics on Code

In this section, we compare the three GOAL agent programs for the dynamic Blocks World based on numeric measures of their 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 4.2) and are the same (with a slight exception for A, see [44]). 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.

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

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

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.*

Besides confirming the previous observation, this observation is related to the expressive power of the knowledge representation language used in GOAL, i.e., Prolog. The latter is so expressive that a large part of the action selection strategy can be coded as part of the knowledge base, without making much use of the GOAL's additional language features. We believe that this effect will be particularly common in cases where the programmer has more experience in Prolog than in GOAL, and where the domain is relatively simple as in the dynamic blocks world. For such programmers, it can be quicker to stick to what they know, while the added benefit of using GOAL's features is limited. In Section 5 in which a more extensive case study is described, this issue was not observed.

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.*

This illustrates that goal dynamics comprises a considerable portion of the agent's reasoning, and is thus an important aspect to consider when teaching GOAL.

4.4 Functional Analysis

In this section, we discuss the code of the GOAL programs in more detail. We first describe these programs with an emphasis on functional aspects, and then discuss similarities and differences.

Description of Programs To facilitate understanding of the programs, we explicitly discuss how they handle dynamics of the environment (caused by someone moving blocks while the agent is building towers). We distinguish the following cases:

1. No influence: A block is moved that *does not influence* the agent. This can be the case if a block not part of the goal configuration is moved from a tower not containing blocks of the goal configuration (or from the table) to a tower not containing blocks of the goal configuration (or to the table).

2. Problematic situation: A block is moved that *prevents the execution of an action* that could be executed before the move, or the agent holds a block that it *does not want to move*. A `pickup` action can no longer be executed if a block is moved on top of the block that the agent wanted to pick up (2a), or if a block is moved into the gripper that is not the one the agent wanted to pick up (2b). A `putdown` action can no longer be executed with the intended result if a block is moved on top of the block on which the agent wanted to put down the block that it is holding (2c), or if the block is taken away from the gripper (2d). If a block is moved into the gripper while the agent did not want to move this block at all, it can be the case that the agent holds a block that it does not want to move (2e).
3. Efficiency: A block is moved that makes the *execution of an action superfluous* or that results in the fact that a *better action could be executed*. The former occurs if the move corresponds to the move that the agent wanted to execute (3a), if the block onto which the agent wanted to move the block, is no longer placed in the right position in the tower (no longer a constructive move) (3b), or if the move makes it possible to make a constructive move, rather than a move to the table (3c). The latter can be the case if a block is moved into the gripper, through which a constructive move can be made, even though the agent wanted to execute a different move (3d).

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 [44] are handled by the knowledge base, and some are handled by one of the action rules (see [44]).

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 [44]). 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 [44]). 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.

Similarities We discuss similarities of the GOAL programs as can be found when looking in more detail at the code. We discuss our observations structured along the components of GOAL about which the observations are made.

Knowledge base Our first observation concerns the knowledge base.

Observation 4 (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.

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

Observation 5 (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 6 (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 [36] 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

Goals 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 built 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 7 (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 4.6 it will be 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.

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.

Program section 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 10) of a initial program, where refinement was done both through the addition of rules, as well as by modifying conditions of rules.

Observation 8 (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.*

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 4.6, 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.

4.5 Software Quality: Readability

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. 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.

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.

4.6 Run-time Behavior

Besides looking at the code of the programs, we have also analyzed their run-time behavior. We have run the programs using four test cases, 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. More precisely, the test cases are the following:

1. The goal is "`a` on `b` on `c` on `table`", the initial state is "`a` on `table`, `b` on `table`, `c` on `table`".

2. The goal is "a on b on c on d on table, e on f on g on table, h on i on j on table", the initial state is "k on l on f on table, h on a on b on d on table, j on i on m on c on table, g on e on table".
3. The goal is "a on b on c on d on table", the initial state is "c on b on table, a on table, d on table". Wait until the gripper has picked up c, and has moved to the top of the screen. Then use the mouse to put a onto d.
4. The goal is "a on b on c on d on table", the initial state is "c on b on table, a on table, d on table, e on table". Wait until the gripper moves downwards in order to pick up c. Before it can pick up c, move block e into the gripper.

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 9 (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.*

[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.

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.[3] 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 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 *underspecify* 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 ways (see, e.g., [20, 4, 21] for approaches that take advantage of this).

Observation 10 (Underspecification) GOAL *naturally induces the specification of non-deterministic agents, i.e., agents of which the behavior is underspecified.*

4.7 Discussion

From the structural analysis of Section 4.3 we can conclude that there are considerable differences in the extent to which programmers use GOAL's features. From these observations we cannot directly conclude whether programmers who use more of the features use them effectively. However, underutilization of the language features is not desirable. This would mean that either the features are not useful, or the programmers do not know how to use them. Combined with the observation that usage of the language's features differs depending on the experience of the programmers, we hypothesize that programming guidelines and

[3] Comparing for each test case the program with the lowest number of executed environment actions with that with the highest, the difference is only around 30% for Test case 2, while being even lower (ranging from 0 to 20%) for the other cases. Taking the total number of actions, the differences are sometimes around 100%.

teaching can have a significant impact on the extent to which the features are used.

We have made several observations about the use of goals. We propose that these observations may form the basis of programming guidelines for using goals. For example, the observation that adoption and dropping of goals is used for handling the dynamics of the environment could be an effective use of goals in other dynamic environments as well. The use of single instance goals could form the basis for adding a feature to GOAL and its IDE that allows to specify which goals are single instance. This could either be used to automatically update the goals as a new instance is adopted (namely by removing the old instance) or by checking this during execution and giving a warning when multiple instances are present in the goal base. To transform the observation about abstraction levels of goals into a guideline, more research is needed to determine in which situations which abstraction level is appropriate. The observation that a considerable portion of the program's rules are formed by rules for adopting and dropping goals corresponds with the observation that a considerable portion of the actions executed at run-time are adopt and drop actions. The relative importance of goal dynamics suggests that this is an important aspect to address when teaching GOAL. More research is needed to design detailed guidelines on how to handle this.

5 Case Study 2: UT2004

A second case study that we performed involved the much more complex and dynamic environment UNREAL TOURNAMENT 2004 (UT2004). Students in a large student project were asked to code teams of agents for this environment. We have performed similar analysis as in the Blocks World case study on source code. The emphasis has been on structural aspects and the identification of patterns in the agent code samples that we analyzed. We did not perform a detailed analysis of run-time behavior of agents, however. Such analysis is quite complicated in an environment such as UT2004. Instead, we ranked multi-agent systems based on their overall performance in a final competition at the end of the student project and used this ranking in our analysis.

5.1 UT2004 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, including, for example, *Deathmatch* in which each bot is on its own and competes with all other bots present to win the game where points are scored by disabling bots, and *Team Deathmatch* which is similar to Deathmatch but is different in that two teams have to compete with each other. One of the key differences between Deathmatch and Team Deathmatch is that in the latter bots have to act as a team and cooperate and coordinate.

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 their 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 one's own home base while making sure one's 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 multi-agent systems.

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' code, their 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 the 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 has to make a turn. This type of low-level control is better delegated to a more behavioral control layer, which was built on top of Pogamut [11]. 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 [24].

5.2 Subjects and Sample

The programmers whose code we have analyzed are first-year BSc computer science students who followed our second-semester course on Programming Multi-Agent Systems and the consecutive Project Multi-Agent Systems (both taught by the first two authors of this paper). These students are the subjects of our experimental research. In the course the students were trained in both Prolog as well as in 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 the students are basic but overall sufficient. Students were, for example, able to apply negation as failure and recursion without problems.

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] Criteria for grading the code were general software engineering criteria such as readability, structure, consistent use of certain patterns, use of comments for explanation, etc. Students were also asked to write about the use of features present in GOAL in coding their agent and to explicitly reflect upon this in a report which was one of the deliverables of the project. We have used the students' comments in our analysis as far as possible but the typical reports did not fit our purposes very well. Part of the reason has been the structure of the reports and we have adjusted the requirements in this regard for future projects to obtain more useful data.

5.3 Structural and Functional Analysis: Identification of Patterns

In this section, we present the observations we made by doing a qualitative, structural and functional analysis of the code of our sample. The emphasis has been on the identification of particular patterns present in code samples. Functional aspects, or the *purposes* for which language elements are typically used, will be discussed throughout this section as well. 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. The patterns that we have found are only informally described and not structured in a format such as is usual in software engineering (cf. [2]) because a more formal description cannot be justified on the basis of code analysis alone and would risk over-interpretation of the results we have obtained. Our work provides a basis for further development of agent based programming patterns or idioms in future work. The presentation is structured around the main language elements of GOAL. We also discuss the more general aspect of coordination and MAS organization. In Section 5.4 more general software engineering aspects are discussed.

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. This is in line with the main function of the knowledge base to represent conceptual and domain knowledge. 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

[4] To be more specific, all teams had high grades on their code, Teams 2 and 3 played in the final of the competition while Team 1 finished in the bottom half.

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 logic about the domain was located in the knowledge base, in line with the main function of the knowledge base to represent conceptual and domain knowledge.

Observation 11 (Use of Knowledge Base) *The knowledge base is used to represent conceptual and domain knowledge in line with its main function.*

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 the predicates `priority` and `weight` to indicate relative importance and predicates such as `needItem` and `wants`. The code fragments for defining these predicates are significant portions of the code, sometimes more than a 100 lines of code.

Observation 12 (Motivational Concepts in Knowledge Base) *The knowledge base is also used to represent and define concepts with motivational connotations.*

Goal Base The use of explicit goals has been limited, which may in part be explained by Observation 12. 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.

Observation 13 (Redundant Abstract Goals) *Code samples include redundant abstract goals that are never actually used to generate agent behavior.*

In 6 out of twelve teams goals are dynamically 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.

Observation 14 (Explicit Dropping of Goals) *Goals, if used, are always removed explicitly by the built-in* `drop` *action.*

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*. We call the latter types of goals *activity goals*. 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.

Observation 15 (Achievement and Activity Goals) *Goal labels that indicate achievement as well as activity goals are used.*

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.

Observation 16 (Use of Goals) *Goals have been mainly adopted to satisfy entry conditions of modules and are dropped in order to or upon exit of a module.*

Action 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.

An example of a communication pattern observed in rules is:

```
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.

Observation 17 (Use of Communication Rules) *Communication rules are used to overwrite one instantiation of a predicate* `pred` *with another. Such rules have the form:*

```
if bel( received(_, pred(NewParameter)), pred(OldParameter) )
then insert( pred(NewParameter) ) + delete( pred(OldParameter) )
```

Note the use of the `+` operator to perform multiple actions in a single rule. 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. Finally, 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 as discussed above, 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.

Although not observed in all code samples, a useful pattern for handling percepts has been used by some of the student teams.

Observation 18 (Percept Rule Pattern) *Percepts typically provide a reason to insert new information but also to remove information from the belief*

base that is no longer up to date. The following programming pattern can be used to do this effectively in many cases. It does assume, however, that reliable information with respect to `fact` *is always available (full observability relative to* `fact`*).*

```
if bel( percept(fact), not(fact) ) then insert(fact).
if bel( fact, not(percept(fact)) ) do delete(fact).
```

The first rule inserts a perceived fact that is not (yet) believed. The second rule removes facts that are believed but not perceived (anymore).

As discussed above, percept rules have a different interpretation than action rules. That is, all applicable instances of a percept rule are applied whereas only one applicable action rule instance is fired. This may be confusing and to clarify the differences in a recent release of GOAL we have introduced a different notation to highlight this difference. The "perceptrule interpretation" now can be written as: `forall ... do ...` instead of `if ... then ...`.

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 eases 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.

Observation 19 (Linear Execution) *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, except for one obligatory rule `if bel(true) then nothing`. This rule is inserted since GOAL enforces the inclusion of one action rule. 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).

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(_)) )`.

Observation 20 (Modules Code Roles) *Modules are used to code different roles of an agent.*

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).

Observation 21 (Exit-module Pattern) *The first rule(s) in a module are used to check reasons for exiting the module, and to do so by means of the built-in action* `exit-module` *if those reasons apply.*

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.)

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.

Observation 22 (Definition of Internal Actions) *Even though it is unnecessary to do so some teams introduce new internal actions that only have effect on the agent's mental state.*

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 [24] 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 have dominated during 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.

Observation 23 (Indicator Predicates for Durative Actions) *Action specifications for durative actions introduce indicators (predicates) to keep track of the fact that such an action is ongoing in the postcondition.*

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).

Communication We distinguish *plain* communication, in which send actions of the form `send(A,Proposition)` are used, from *advanced* communication with mental models in which actions of the form `send(A,:Proposition)`, `send(A,!Proposition)`, `send(A,?Proposition)` are used.[5] Mostly plain communication is used. Team 3 uses a few instance of 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 the other agents what to do.

Two main ways of *handling received messages* can be distinguished. The first 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. A benefit of this method may be efficiency since no preprocessing is needed, and is simpler to some extent since no preprocessing rules have to be written. A second way to handle received messages is by *preprocessing* messages using percept rules, which insert the received information into the belief base and delete the received message. Team 2 also uses the `received` predicate in the knowledge base of the management agent.

Observation 24 (Message Processing) *Messages are often preprocessed using percept rules by inserting the received information into the belief base.*

Observation 25 (Message Processing Pattern) *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))
```

This pattern is closely related to Observation 17 but instead of *replacing* a proposition the deletion part of this rule removes references to the source (the agent that sent the message) of the proposition that is inserted into the belief base. The benefit of using a pattern like this is that it yields better readable code because action rules and knowledge base are not cluttered with the `received` predicate, and allows reasoning with the added propositions using the knowledge and belief base.

[5] For each of these send actions, there is a `sendonce` variant with which a message is sent only once, assuming that sent messages are kept in the mailbox.

Coordination and MAS Organization The *organisation structures* chosen by the students were hierarchical and network [14]. 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 a 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 role 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.

Observation 26 (Role Switching) *Dynamic switching between roles within an agent is achieved by means of dedicated rules.*

5.4 Software Quality

We make several observations concerning human factors and software engineering (see also Section 3.3), 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 5.3). A large number of conjuncts can make it difficult to grasp what is expressed by the condition.

Observation 27 (No Use of Macros) *None of the student teams have used macros to enhance readability of code.*

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. Putting more emphasis on the readability benefits that they provide and presenting convincing examples may positively influence the use of macros. 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 conditions (expressed as 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*. It is clear from students' reports that they are aware of this duplication but do not see how to avoid it.

Observation 28 (Code Duplication) *Code from one agent program is duplicated in another.*

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 (see also Section 5.3). 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. Code duplication is a clear violation of the Don't Repeat Yourself (DRY) principle and should be avoided whenever possible. It thus is imperative that a language provides facilities for code reuse to avoid duplication. The fact that students found it difficult to avoid code duplication is is an important finding pointing to a need for tools or programming constructs that help avoid code duplication. This topic has received little attention in the agent-oriented programming community, although, for example, work on modules may have potential for addressing this issue.

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>).
```

Similar rules are added for other agent names and different percepts. Hardcoding is generally considered bad practice as it introduces problems for maintaining code. It is, however, not a deficit of a programming language that it allows such practices but rather requires programmers to be aware that hardcoding is bad practice. Considering that our students are first-year BSc students which have not yet been trained in the area of software quality, it seems clear that this issue relates to experience and teaching how to program good agent programs.

5.5 Discussion of UT2004 Observations

In this section, we discuss several items based on the findings of the previous sections.

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, see Observation 19) 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. 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 [7]. 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 Observation 21). 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 (Observation 14).

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. This observation is similar to Observation 5 about dropping of goals being used for dealing with dynamics of the environment.

Language Design and Development Environment 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*, as well as for improvement of the development environment.

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). At the time of writing, these insights have

been used and it is allowed to remove sections that would otherwise have been empty.

We will consider the introduction of warnings and automatic dependency analysis and checks in the development environment: checks on whether goals can ever become beliefs of the agent (to indicate proper use of achievement goals); checks 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 Discussion

In this section we evaluate our experimental setup (Section 6.1) and we discuss to what extent our results are generalizable to other cognitive agent programming languages (Section 6.2).

6.1 Evaluation of Experimental Setup

We evaluate our experimental setup by discussing how different choices might impact our findings.

Subjects The subjects of our first case study were researchers and a programmer who had varying experience with GOAL, while the subjects for the second case study were first year BSc students who had just learned GOAL (and Prolog). Observation 1, which notes that the use of specific language elements of a programming language are more often used by experienced programmers in that language, suggests that some of our findings in the second case study, e.g., concerning the use of macros and goals, might be different if the subjects were more experienced. While this might be the case, we believe that the latter findings should not be dismissed because of this. Our aim is to shape the instruments that support programming of GOAL agents such that novices are able to use the language's essential constructs in a suitable and effective way. Consequently, adaptations to these instruments should be considered to achieve this objective.

Moreover, one might consider it desirable to perform the second case study also on subjects that are not students (e.g., researchers or professional software developers) to investigate how that would influence the results. While this might be interesting, our main target audience for the language at this stage of its development *is* students. Until the language is ready for more wide-scale adoption also in industry, our main aim is to teach students about programming multi-agent systems.

Teachers Concerning the second case study, our findings might be influenced by who taught the course. Here we note that the teachers of the course (the first two authors of this paper) are designers of and experts on GOAL, which

has been used in teaching at our university since 2007. Any issues that were identified are thus likely to have arisen also if other teachers, less experienced in GOAL, would have taught the course. Vice versa, we consider it possible that that would have given rise to *more* issues because of lack of experience. For studying this, it would be interesting to compare results of GOAL courses taught at other universities. We hope that the instruments for programming GOAL agents that we are continuously improving (also taking into account the results of this study), will have a positive effect on the number of agent programming courses that use GOAL, which would facilitate such comparisons. We conjecture that programming guidelines and teaching materials (assuming that these are followed by the teachers) will have more impact on the results than the teachers themselves.

Application domains The applications considered in our case studies range from the dynamic blocks world (a small single-agent domain with limited dynamics and real-time requirements) to UT2004 (a real-time highly dynamic multi-agent environment where agents have to cooperate within their team and compete with the opposing agent team). The advantage of studying programs for a small domain like the blocks world is that the resulting programs are relatively small, which allows a detailed analysis of all parts of their code. Also, run-time behavior can be studied in detail as test cases can be specified precisely. However, this domain does not include certain essential characteristics of more challenging agent applications, in particular those that have to function in highly dynamic environments and that consist of multiple agents. Our UT2004 case study does have these characteristics, which makes it suitable as a case study for agent programming. The observations and programming patterns identified in that case study are general, i.e., it seems it would be likely that they could be found in other real-time highly dynamic multi-agent applications.

That said, we do believe it is important to study the use of the language in other domains and contexts like social robotics or organization-aware agents [46] which should be able to understand the norms and work processes of organizations in which they participate. Applying the language in such other domains will provide a better understanding of the language's suitability for these domains, and might give rise to extensions that are specifically tailored to those contexts.

Which Instruments to Improve As indicated in Section 1, with this work we are aiming at improvements of three interrelated instruments: programming language, programming guidelines & teaching methods, and development environment. From our observations it is not always clear which of these instruments should be changed to have a certain effect, i.e., which instruments are the cause of certain issues in the code. Throughout the paper we have indicated modifications that we would like to incorporate in these three instruments, based on our estimation as to what would most likely yield better results in terms of code. We generally take a conservative approach when it comes to proposing changes in

language design: if we believe better programming guidelines and teaching methods would mitigate some of the issues we found, we prefer this over changing the language. The language design is based on solid theoretical foundations and we believe changes should not be made unless we have a clear idea of which aspects would improve. Future studies will have to show to what extent our proposed improvements indeed yield better agent programs.

Evaluation of Approach The method that we have applied in this paper is mainly of a qualitative nature although we have used some rudimentary statistics to substantiate some our findings. In this paper the focus has been on making various useful observations, identifying patterns and potential issues. We have made several hypotheses concerning reasons for these observations, but future work will have to validate these using a more quantitative approach. It is, moreover, hard to conclude that we have not overlooked important observations and it would be useful to explicitly address this issue in the research approach, beyond the within-case and cross-case analysis discussed before. Nevertheless, given the current state-of-the-art, we believe that our qualitative approach has much to bring to the agent community and one might argue that first more similar studies are needed before we can refine the approach and formulate it in more detail.

6.2 Generalizability to Other Cognitive Agent Programming Languages

Most cognitive agent programming languages are rule-based as GOAL is. Moreover, as many agent programming languages have features and concepts such as beliefs and goals as GOAL does, it is only to be expected that at least some of our results will also apply to these other programming languages. Below we discuss this in more detail.

Observation 1 which notes that the use of specific language elements of a programming language are more often used by experienced programmers in that language is probably universal and will apply to other languages as well. It highlights again the importance of teaching and language understanding. It is also well-known though not well-documented that solutions to programming problems encountered in agent-oriented programming can be resolved in different ways. It has, for example, also been observed by others that the underlying knowledge representation technology (e.g. Prolog) can be used to a large extent to code solutions to such problems as was observed with the programmer of program A, see Observation 2.

As far as the observations concerning the use of knowledge or beliefs are not specific to the knowledge representation language of GOAL (i.e., Prolog), we expect that they are applicable to most other cognitive agent languages as well since they usually incorporate similar informational attitudes. Concerning Observation 2 (focus on knowledge base or action rules), we expect that it generalizes to other languages of which the expressive power of the language used for representing knowledge and beliefs is high (like in the case of Prolog). If no rules

can be used in the knowledge or belief base like in *Jason* [10], this observation may not hold.

We have made quite a few observations related to the use of explicit, declarative goals in agent programs. These are particularly applicable to languages that incorporate such goals, most notably GOAL and 2APL [13]. Most of the observations also show that there is still work to do to facilitate programming with explicit, declarative goals and to enhance understanding. We speculate that the use of predicates with motivational connotations (Observation 12) may occur even more often in languages without a notion of an explicit, declarative goal such as *Jason*.

Although other languages such as 2APL and *Jason* also have been extended with modules [13, 31], experience with programming with modules in agent-oriented programming is relatively new and our work is probably one of the first to document such experiences in the context of a large programming project. These module concepts have in common that they encapsulate various notions associated with an agent, such as beliefs and goals. As we have observed in GOAL agents, modules in other languages have also been proposed to code roles or to code specific agent capabilities.

Comparing our finding that students have a preference for explicit control (Section 5.5) with implementations of other languages, we note that many of the latter inherently incorporate explicit control features in the language. For example, the implementation of 2APL imposes a linear rule evaluation order without giving the programmer the freedom to choose other evaluation options. Also, frameworks like Jack and Jadex that build on an imperative language inherently require the programmer to make control aspects explicit. We believe that the flexibility that GOAL provides in terms of allowing the programmer to choose whether or not to use explicit control features, on the one hand gives more freedom to the programmer and on the other hand facilitates research performed in this paper as to preferences of programmers.

We believe that observations related to perceiving an environment (Observation 18) and communication with other agents (Observations 17, 24, and 25) may be relevant for other agent programming languages as well. Although particular code snippets for implementing these patterns may differ, the basic ideas are general and we believe they can be applied across platforms. To be more specific, regarding perception of the environment we note that the pattern of Observation 18 applies to languages that allow for explicit modifications to the agent's beliefs based on percepts. If beliefs are updated automatically, this pattern does not apply. *Jason* implements automatic updates of beliefs, but annotates these based on the source of information. A similar pattern as we observed could be used there to transform beliefs received from another agent to beliefs that the agent itself really believes.

The observations related to creating structure and organization in a multi-agent systems are general. It would be useful to obtain more examples that illustrate how such organization is achieved in various languages for comparison.

Most of our observations relate to the basic concepts that are used within cognitive agent programming languages. Therefore, with the exception of observations that specifically apply to the declarative nature of the knowledge representation language of GOAL as discussed above, we believe that they may also apply to agent programming languages that are not so much logic-based but more Java-oriented such as JACK, Jadex, and AF-APL [1, 37, 38]. However, of course more research is needed to confirm this.

We strongly believe that developing patterns will help mature the agent-oriented paradigm. As patterns capture best practices and experience, they can contribute to a formalization of common and recurring solutions to problems agent developers encounter during implementation and testing of multi-agent systems. Of course, we are not the first to make this observation. Various authors have suggested the need for a *pattern-based approach* to agent orientation. Patterns provide ways to describe best practices and proven designs, and to capture experience in a way that enables others to reuse this experience [2]. The agent-oriented paradigm requires the design of new patterns because it is based on intentional and social concepts and has its own associated set of implementation languages [18]. In [49] it is pointed out that establishing patterns for multi-agent systems will promote expertise for developing such systems. For similar reasons [48] promotes a pattern-based approach to agent development. This has motivated a broad research agenda on agent-oriented patterns [15, 30, 39, 41, 43]. In [34] a comprehensive and state-of-the-art overview of work on these patterns is provided. One of the few AOP patterns that have been published to date are motivated by work on the semantics of goals [28] instead of being derived from the practice of agent-oriented programming. We believe that our own work provides a more bottom-up approach to establishing such patterns although we realize that more research is needed in this area.

7 Conclusion and Future Work

In this paper, we proposed an approach for empirically studying how programmers use a programming language, in which we identify several analysis dimensions. We have performed two case studies in which we analyzed agent programs (the first for the dynamic Blocks World and the second for UT2004) written in the GOAL agent programming language along the identified dimensions. We have made several observations based on a qualitative analysis of the code of these agent programs. Here we summarize the most important observations along the analysis dimensions of our approach.

Functional analysis In both case studies, the knowledge base is used to represent domain knowledge. Concerning the use of goals, two out of three programmers in the first case study use goals extensively, while goals are used to a limited extent in the second case study. In the former, dropping of goals is used for dealing with dynamics of the environment, while in the latter it is used for entering and exiting modules. In the first case study we have also observed the existence of

single-instance goals, and that goals can be used on different abstraction levels. Concerning the program section, in the second case study we have observed that all teams use a linear rule order evaluation. This contrasts our finding in the first case study, where we observed that GOAL programs induce the specification of non-deterministic agents. Possibly programmers are more likely to specify non-deterministic agents if the domain is relatively small and thus easier to grasp. Percept rules, which were given to the subjects in the first case study, are used to update the belief base based on percepts received from the environment and messages received from other agents. Modules are used to program roles, and dedicated rules are used to switch between roles (and thus between modules). Concerning action specifications we observed in the second case study a difference between durative actions (which record the fact that such an action is ongoing on the postcondition) and instantaneous actions (which express the immediate effect in the postcondition or use a true postcondition in combination with belief updates on the basis of percepts).

Structural analysis While we observed in the first case study an emphasis either on the knowledge base or on action rules (in terms of numbers of rules/clauses), in the second case study such a distinction was not observed. This may indicate that in a challenging environment both of these essential elements of GOAL are necessary/useful for programming the agents. Also we noticed that the good teams had on average a significantly higher number of action rules, presumably because they programmed a more elaborate strategy. The number of percept rules in the second case study was comparable to the number of action rules, while it was significantly smaller in the first case study (at least for the programs that made extensive use of action rules). This may be due to the fact that percept rules are also used for message processing, which was not needed in the first case study. In the second case study we have identified code patterns for updating the belief base on the basis of communication and percepts, and for using modules.

Analysis of software quality In the first case study a readability experiment was conducted. It was found that the high number of belief and goal conditions in action rules makes them hard to read. Macros have been added to GOAL to improve this, but these were not used by the programmers in the second case study. We conjecture that this is due to lack of emphasis on this construct when GOAL was taught, and limited use of goals (which often results in only one belief operator being used for a rule). In the second case study we observed frequent occurrences of duplicate code. This may be prevented by adding support for importing modules and by adding a feature to the development environment to detect this.

Analysis of run-time behavior Run-time behavior was only analyzed in the first case study. There it was observed that in the programs that use goals, the adopt and drop actions formed a considerable portion of the total number of executed actions. Although we did not perform this analysis in the second case study, the limited use of goals can be expected to result in relatively few adopt and drop

actions. This can be expected to change when goals would be used more.

Overall, we can conclude that GOAL allows programmers to develop multi-agent systems in which high-level team strategies are used, in combination with interaction with the virtual environment.

Through our analysis, we have come closer to the development of instruments for facilitating programming of high-quality GOAL agents. 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 may also help us in developing better debugging and development software that provides better support to programmers. An example of this related to the automatic checking of whether goals may ever be believed by an agent has been discussed in Section 5.5.

In future work, we plan on improving GOAL, its programming guidelines and development environment along the lines suggested in this paper. We plan to study the effects of this, and to further investigate the hypotheses formed through our analysis, e.g., concerning the reasons for the use of explicit control rather than built-in semantics.

References

1. AOS group. Jack: an agent infrastructure for providing the decision-making capability for autonomous systems (whitepaper). `http://www.aosgrp.com/downloads/JACK_WhitePaper_US.pdf`, September, 2011.
2. B. Appleton. Patterns home page. 2011. [`http://www.hillside.net/patterns/`; accessed February 2011].
3. SWI Prolog. `http://www.swi-prolog.org/` (Accessed 30 Jan 2010).
4. L. Astefanoaei and F. S. de Boer. Model-checking agent refinement. In *AAMAS*, pages 705–712, 2008.
5. V. R. Basili and L. C. Briand, editors. *Empirical Software Engineering: An International Journal*. Springer, 2009. `http://www.springer.com/computer/programming/journal/10664`.
6. T. Behrens and K. V. Hindriks and J. Dix. Towards an environment interface standard for agent platforms *Annals of Mathematics and Artificial Intelligence*, 1–35, 2010.
7. 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.
8. R. H. Bordini, M. Dastani, J. Dix, and A. El Fallah Seghrouchni. *Multi-Agent Programming: Languages, Platforms and Applications*. Springer, Berlin, 2005.
9. R. H. Bordini, M. Dastani, J. Dix, and A. El Fallah Seghrouchni. *Multi-Agent Programming: Languages, Tools and Applications*. Springer, Berlin, 2009.
10. R. H. Bordini, J. F. Hübner, and M. Wooldridge. *Programming Multi-agent Systems in AgentSpeak using Jason*. Wiley, 2007.
11. O. Burkert, R. Kadlec, J. Gemrot, M. Bída, J. Havlíèek, M. Dörfler, and C. Brom. Towards fast prototyping of IVAs behavior: Pogamut 2. In *Proc. of IVA'07*, 2007.

12. M. Dastani. 2APL: a practical agent programming language. *Autonomous Agents and Multi-Agent Systems*, 16(3):214–248, 2008.
13. M. Dastani. Modular Rule-Based Programming in 2APL. In A. Giurca, D. Gasevic, and K.Taveter (eds.), Handbook of Research on Emerging Rule-Based Languages and Technologies: Open Solutions and Approaches, pages 25-49, Information Science Reference, Hershey, PA, USA, 2009.
14. V. Dignum. *A Model for Organizational Interaction: Based on Agents, Founded in Logic*. PhD thesis, 2004.
15. T. T. Do, M. Kolp, and S. Faulkner. Agent-oriented design patterns: the skwyrl perspective. In *Proceedings of the 6th International Conference on Enterprise Information Systems (ICEIS'04)*, pages 48–73, 2004.
16. K. M. Eisenhardt. Building theories from case study research. *The Academy of Management Review*, 14(4):532–550, 1989.
17. O. Hazzan. Orit Hazzan's Column: Qualitative Research in Software Engineering. `http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.137.7613`, 2006.
18. C. Heinze. *Modelling intention recognition for intelligent agent systems*. PhD thesis, The University of Melbourne, 2003.
19. K. V. 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.
20. K. V. 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.
21. K. V. 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.
22. 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.
23. K. V. Hindriks and M. Birna van Riemsdijk. A computational semantics for communicating rational agents based on mental models. In *ProMAS'09*, volume 5919 of *LNAI*, 2010.
24. K. 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.
25. K. V. Hindriks and M. B. van Riemsdijk and C. M. Jonker. An empirical study of patterns in agent programs. In N. Desai, A. Liu, M. Winikoff, editors, *Principles of Practice in Multi-Agent Systems 2010*, 2011. To appear in LNAI. Best paper award (runner up).
26. K. V. Hindriks. GOAL Programming Guide. `http://mmi.tudelft.nl/~koen/goal`, 2010.
27. James Howatt. A project-based approach to programming language evaluation. *ACM SIGPLAN Notices*, 30(7):37–40, 1995.
28. J. F. Hübner, R. H. Bordini, and M. Wooldridge. Programming declarative goals using plan patterns. In M. Baldoni and U. Endriss, editors, *Declarative Agent Languages and Technologies IV*, pages 123–140. Springer, 2006.
29. R. J. Howell and R. Collier. Evaluating agent-oriented programs: Towards multi-paradigm metrics. In *Proceedings of the International Workshop on Programming Multi-Agent Systems (ProMAS'10)*, pages 63–79, 2010.

30. M. Kolp, T. T. Do, S. Faulkner, and T. T. H. Hoang. Introspecting agent oriented design patterns. In S. K. Chang, editor, *Advances in Software Engineering and Knowledge Engineering*, 2005.
31. N. Madden and B. Logan. Modularity and compositionality in Jason. In Programming Multi-Agent Systems: 7th International Workshop, ProMAS 2009, Budapest, Hungary, May 10-15, 2009, pp. 237-253.
32. M. N. Marshall. Sampling for qualitative research. *Family Practice*, 13(6):522–525, 1996.
33. N. J. Nilsson. Teleo-reactive programs and the triple-tower architecture. *Electronic Transactions on Artificial Intelligence*, 5:99–110, 2001.
34. A. Oluyomi, S. Karunasekera, and L. Sterling. A comprehensive view of agent-oriented patterns. *Autonomous Agents and Multi-Agent Systems*, 15(3):337–377, 2007.
35. L. Padgham and M. Winikoff. *Developing Intelligent Agent Systems: A Practical Guide*. Wiley Series in Agent Technology. John Wiley and Sons, 2004.
36. 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.
37. 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.
38. S. Russell, H. Jordan, G.M.P. O'Hare, and R.W. Collier. Agent Factory: A Framework for Prototyping Logic-Based AOP Languages. In *Proceedings of the Ninth German Conference on Multi-Agent System Technologies (MATES 2011)*. Springer, Berlin, 2011.
39. L. Sabatucci, M. Cossentino, and S. Gaglio. A semantic description for agent design patterns. In *Proceedings of the 6th International Workshop From Agent Theory to Agent Implementation (AT2AI'08)*, 2008.
40. Y. Shoham. Agent-oriented programming. *Artificial Intelligence*, 60:51–92, 1993.
41. C. Silva, J. ao Araújo, A. Moreira, and J. Castro. Designing social patterns using advanced separation of concerns. In *Proceedings of the 19th international conference on Advanced information systems engineering (CAiSE'07)*, pages 309–323. Springer-Verlag, 2007.
42. J. Slaney and S. Thiébaux. Blocks World revisited. *Artificial Intelligence*, 125:119–153, 2001.
43. Y. Tahara, A. Ohsuga, and S. Honiden. Agent system development method based on agent patterns. In *Proceedings of the 21st International Conference on Software Engineering*, pages 356–367, 1999.
44. 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/trac/goal/wiki/Projects/Empirical_AOP`.
45. M. B. van Riemsdijk and K. V. Hindriks. An empirical study of agent programs: A dynamic blocks world case study in GOAL. In J.-J. Yang, M. Yokoo, T. Ito, Z. Jin, and P. Scerri, editors, *Principles of Practice in Multi-Agent Systems*, volume 5925 of *LNAI*, pages 200–215. Springer, 2009. Best paper award.
46. M. B. van Riemsdijk, K. V. Hindriks, and C. M. Jonker. Programming organization-aware agents: A research agenda. In *Proceedings of the Tenth International Workshop on Engineering Societies in the Agents' World (ESAW'09)*, volume 5881 of *LNAI*, pages 98–112. Springer, 2009.

47. A. I. Wasserman. Issues in programming language design— an overview. *SIGPLAN Notices*, 1975.
48. M. Weiss. Patterns for motivating an agent-based approach. In *Conceptual Modeling for Novel Application Domains*, pages 229–240, 2003.
49. D. Weyns, A. Helleboogh, and T. Holvoet. How to get multi-agent systems accepted in industry? *International Journal of Agent-Oriented Software Engineering*, 3(4):383–390, 2009.
50. M. Winikoff. JACK$^{\text{TM}}$ 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.