Does high-level behavior specification tool make production of virtual agent behaviors better?

Jakub Gemrot, Zdeněk Hlávka, Cyril Brom

Charles University in Prague, Faculty of Mathematics and Physics, Czech Republic
`jakub.gemrot@gmail.com, hlavka@karlin.mff.cuni.cz, brom@ksvi.mff.cuni.cz`

Abstract. Reactive or dynamic planning is currently the dominant paradigm for controlling virtual agents in 3D videogames. Various reactive planning techniques are employed in the videogame industry while many reactive planning systems and languages are being developed in the academia. Claims about benefits of different approaches are supported by the experience of videogame programmers and the arguments of researchers, but rigorous empirical data corroborating alleged advantages of different methods are lacking. Here, we present results of a pilot study in which we compare the usability of an academic technique designed for programming intelligent agents' behavior with the usability of an unaltered classical programming language. Our study seeks to replicate the situation of professional game programmers considering using an unfamiliar academic system for programming in-game agents. We engaged 30 computer science students attending a university course on virtual agents in two programming assignments. For each, the students had to code high-level behavior of a 3D virtual agent solving a game-like task in the Unreal Tournament 2004 environment. Each student had to use Java for one task and the POSH reactive planner with a graphical editor for the other. We collected quantitative and qualitative usability data. The results indicate that POSH outperforms Java in terms of usability for one of the assigned tasks but not the other. This implies that the suitability of an AI systems-engineering approach is task sensitive. We also discuss lessons learnt about the evaluation process itself, proposing possible improvements in the experimental design. We conclude that comparative studies are a useful method for analyzing benefits of different approaches to controlling virtual agents.

Keywords: Virtual agents, Agent development techniques, Empirical studies

1 Introduction

Reactive planning is currently the dominant paradigm for controlling virtual agents in 3D videogames and simulations. Prominent reactive planning techniques used in the industry are derivations of finite state machines (FSMs) [1] and behavior trees [2]. Technically, these are implemented in a scripting language, such as general-purpose Lua [3] or special-purpose UnrealScript [4], or they are hard-coded in a game's native

language, typically C++ [5]. Advantages and drawbacks of different industry approaches have been commented on widely [6, 7, 8]. The general agreement in the academia is that scripting languages do not provide enough expressivity for creating complex human-like agents, or it is cumbersome to use them for this task, and that there should be a better way for creating virtual agents behaviors.
At the same time, academia is producing action selection (AS) systems that seek to improve cognitive performance of agents. These include the decision making modules of cognitive architectures, e.g. Soar and ACT-R [9, 10], stand-alone BDI-based languages, e.g. GOAL [11], and reactive planners such as POSH [12]. It has been demonstrated that some of these systems, e.g. Soar [9], POSH [13], GOAL [11] and Jazzyk [14], can be used for controlling virtual agents acting in game-like environments. However, cognitive performance of an agent is not the only concern of the game industry. Ease of use, code readability and re-usability (of parts of code) play an important role in eagerness of adoption of new systems. In fact, these features may be more important than the agent's cognitive performance as the industry will be unlikely to adopt systems that are hard to use or produce code incomprehensible to anybody except the author.

Academic approaches often use custom behavioral languages to disguise underlying low-level code (in Lua, C++ etc.) forcing the programmer to think in high-level behavioral constructs, such as mental states, goals, action competences or triggers. These constructs are also defined explicitly as language primitives to be organized by programmers into behavioral plans that are interpreted by an AS system. Still, these AS systems are tied with the disguised low-level code for the purpose of communication with the environment, including information gathering and processing, and action executions and monitoring the course of the execution. The high-level languages typically lack synchronization or generic *while* statements to deal with application protocols gracefully, therefore AS systems must also define interfaces between these two levels. This two-layer architecture is thought to have several positive outcomes: 1) the low-level agent "periphery" should be reusable by different high-level plans, 2) well structured low-level code should improve comprehensibility, 3) correctly-separated high-level constructs should be easier to understand and extend, 4) a high-level plan is thought to be easily grasped by non-programmers, such as game designers, as it is more intuitive. Essentially, the high-level plan is to the low-level code what SQL is to Cobol.

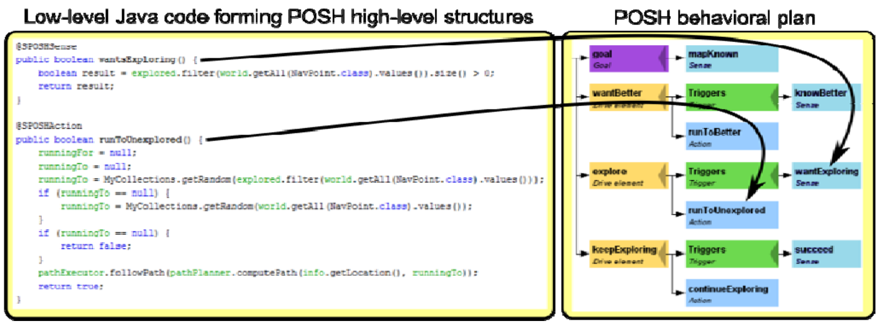
Technically, one does not need an academic AS system featuring a high-level behavioral language to create complex behaviors. An option exists to hard-code everything inside the low-level code as we witness in many computer games. Which approach is better?

Two particular scenarios are encountered in game companies often and it is worth investigating this question in the context of these situations. First, when an AI developer leaves a company, somebody needs to continue his or her work. It is desirable that a developer's code is as comprehensible as possible (even without documentations and code commenting), so that it is easy to extend. Second, if a company creates a sequel to its game, it might be desirable that some existing code for the agents' behaviors is reused. Thus, it should be easy to augment or refactor existing behaviors.

The goal of this paper is to present results of a quasi-experimental, comparative study with both quantitative and qualitative measures modeling the abovementioned situations of AI developers, a method adopted from psychology and social sciences. The study's goal was to investigate whether an academic approach that combines both the lower and the higher level behavior description outperforms an industry approach employing only the lower level in the situations in question and at the same time, gain insight into the utilization of high-level constructs. We adopt Java as the industry approach and POSH reactive planner [12] as the academic approach. Java is at least as good as C++ for programming complex agent's behavior, but it is not a typical game industry language. We use it for two reasons. First, all our subjects, [22] university computer science students, know Java acceptably well, which models a situation in a company where programmers known their programming language. Second, POSH's lower level uses Java. We use POSH because it has been already demonstrated for controlling virtual agents [13]. It also benefits from a graphical tool developed for visualizing an agent's behavior plan using high-level constructs only [17]. POSH can be thought of as typical of a broad class of academic solutions such as a BDI-based systems, that include planning and primitive layers.

All our subjects attended a course on programming virtual agents for games where they learned POSH. Their situation corresponded to situation of game programmers considering using an academic system after experimenting with it for about three person-days. Our subjects were divided into two groups. Both groups worked on the same tasks from a first-person shooter domain using Unreal Tournament 2004 environment (UT04), but the first used Java only and the second used POSH to model higher level control. Our hypotheses were:

1. POSH outperforms Java in terms of subjectively-perceived usability and objective quality of the resulting agents.
2. POSH outperforms Java when the task is to catch up upon the work of someone else.
3. POSH outperforms Java when the task is to extend one's own code (three months later).



```
@SPOSHSense
public boolean wantsExploring() {
    boolean result = explored.filter(world.getAll(NavPoint.class).values()).size() > 0;
    return result;
}

@SPOSHAction
public boolean runToUnexplored() {
    runningFor = null;
    runningTo = null;
    runningTo = MyCollections.getRandom(explored.filter(world.getAll(NavPoint.class).values()));
    if (runningTo == null) {
        runningTo = MyCollections.getRandom(world.getAll(NavPoint.class).values());
    }
    if (runningTo == null) {
        return false;
    }
    pathExecutor.followPath(pathPlanner.computePath(info.getLocation(), runningTo));
    return true;
}
```

Fig. 1. Relation between high-level POSH plan and low-level Java code presenting separation of high-level behavioral code from the low-level code of sensors and actions.

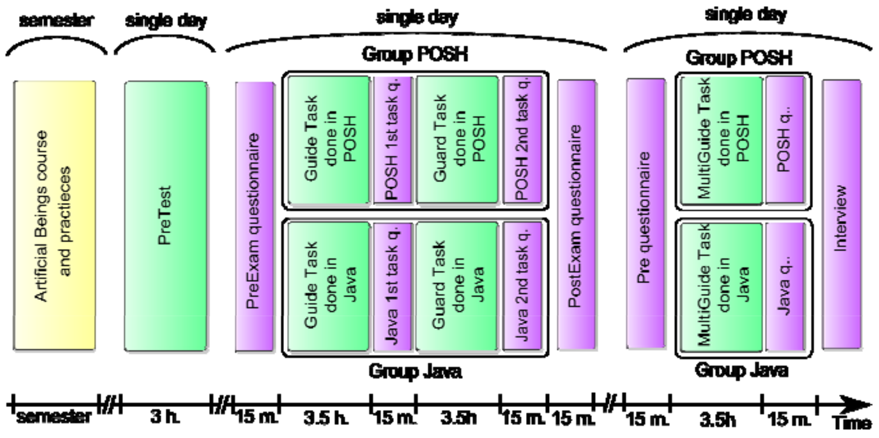


Fig. 2. The course of the experiment.

If these had been confirmed, there would have been an empirical argument for maturity of at least one particular academic solution. In fact, no hypothesis was supported by the data. This means that it is important to isolate the most beneficial and problematic features of POSH to suggest possible improvements. Features shared with different academic systems are the most important.

The rest of the paper is organized as follows. Section 2 presents related work. Section 3 introduces POSH [12] explaining its roots, architecture and Behavior Oriented Design [15] methodology for the design of virtual agent behaviors. Section 4 describes the experiment setup and following section 5 presents its results. Section 6 discusses results and presents ideas for general improvements to AS systems, which concludes the paper.

2 Related work

Empirical studies of academic AS systems are scarce. In our previous comparative study [28], we demonstrated that POSH, enhanced with a GUI, is at least as good as Java, but in that study, subjects programmed the high-level code only and for relatively simple tasks, which is a study's limitation. Doubts about using only the high-level constructs for programming complex agents behavior stem from the work of Píbil et al. 16, who reports on experiences from the creation of a team of agents solving a MAS game-like scenario inside a grid world using vanilla Jason [22] implementation, that is, using the Jason's high-level constructs only without modifying the low-level Java code. They report on hard Jason's corners and the inevitable implementation of complex behavior primitives in underlying Java language.

Hindriks et al. [19] conducted a qualitative analysis of the code of 60 first year computer science students developing (in teams) three Capture The Flag agents for UT04 using GOAL agent programming language. That work aimed at "providing insight into more practical aspects of agent development" and "better understanding problems that programmers face when using (an agent programming) language" and identified a number of structural code patterns, information useful for improvements to the language. However, that study was not comparative and did not report the programmers' feedback.

The fact that an AS system's usability is also closely linked to the quality of developers' tools and their ability to visualize complex behaviors in an intuitive way was recognized by Heckel et al. [20] in their work on BehaviorShop. A usability study of BehaviorShop demonstrated a well-thought GUI for editing high-level behavioral plans is easily graspable by non-programmers. However, their study was not comparative and they did not allow subjects to work with low-level code, which is arguably required for larger behavior modifications as argued by [15, 16].

The industry's interest in simple and intuitive tools is exemplified in Desai's work on ScriptEase [24]. ScriptEase is a graphical authoring tool of the BioWare's NeverWinter Nights game allowing non-programmers to create new game modules. She shows that her simplification of the GUI is welcomed by both programmers and designers.

3 POSH

POSH action selection was originally developed in the late 1990s in response to criticism of what was then an extremely popular agent design approach (at least in academia): Subsumption Architecture (SA) [23]. SA was used to produce considerable advances in real-time intelligent agents, particularly robotics. It consists primarily of two components: a highly modular architecture where every action is coded with the perception it needs to operate; and a complex, highly distributed form of action selection to arbitrate between the actions that would be produced by the various modules. Although well-known and heavily cited, the SA was seldom really used outside of its developers. Bryson hypothesized that the emphasis on modular intelligence was actually the core contribution of SA, but that the complexity of action selection, while successfully enforcing a reactive approach, confused most programmers who were not used to thinking about concurrent systems.

POSH was developed to simplify the construction of action selection for modular AI. A programmer used to thinking about conventional sequential programs is asked to first consider a worst-case scenario for their agent, then to break each step of the plan to resolve that scenario into a part of a reactive plan. Succeeding at a goal is the agent's highest priority, the thing the agent does if it can. The programmer must therefore describe for the agent how to perceive that its goal can be met. Then for each step leading up to the goal the same process is followed: a perceptual condition is defined allowing the agent to recognize if it can take the action leading most directly to its goal [12]. The actions are each small chunks of code that control the agent briefly, so-called behavior primitives (see Fig. 1).

After a period of experimenting with the system, Bryson embedded POSH in a more formal development methodology called Behavior Oriented Design (BOD) 18. BOD emphasizes the above development process, and also the use of behavior modules written in ordinary object-oriented languages (low-level code) to encode the majority of the agent's intelligence, including its memory. These modules provide the high-level behavior and sensory primitives; methods calls are the interface between a high-level POSH plan and the low-level code of the behavior modules (see Fig. 1). BOD includes a set of heuristics for recognizing when intelligence should be refactored either from a plan to a behavior module or to decompose a complex module using a plan.

Recently, a graphical editor for POSH plans has been developed. Its new version was used in the present study (Fig. 1).

4 Method

4.1 Experiment design

The study compared the usability of an academic AS system, POSH, enhanced with a graphical tool for the creation of high- level behavioral plan, and an unenhanced classical programming language, Java. The context of the comparison was two particular situations mentioned in Sec. 1 common in the game company. The study was divided into three tasks. Each task was to create a behavior for an agent that had to fulfill a game-like goal. Subjects using Java had a complete freedom in the way of coding the behavior. Subjects using POSH were constrained by the requirement to separate low-level Java code into behavior and sensory primitives, specific constructs, which were then used inside high-level POSH behavioral plan (see Fig. 1).

The study was set in an AI course for computer science students in REMOVED. Subjects were given a pretest (3 hours) after the course to ensure that they had acquired elementary skills for solving sub-problems from the final exam. Only subjects that had passed the pretest were admitted to the final exam.
The final exam was organized to obtain comparative data on Java and POSH usability and provide data for the first game company scenario (see Sec. 1). The final exam consisted of two tasks, the Guide Task (3.5 hours) and the Guard Task (3.5 hours), see Sec. 4.3. Subjects were split into two groups, the Java group and the POSH group. In the first task, subjects were to create the whole Guide behavior from scratch. In the second task, each subject received a code from a randomly selected colleague from the same group and was asked to extend it into Guard behavior. There was a 30 minutes long break between the first and the second task. Finally, three months later, some subjects participated in the final task, in which every subject was given his code from the first task and was asked to extend it into MultiGuide behavior (3.5 hours). The follow-up provided data for the second game company scenario.

Subjects were given 4 questionnaires in total during the final exam (15 minutes each) and 2 questionnaires during the follow-up. Subjects solving the follow-up also underwent a structured interview that was meant to provide more accurate qualitative data as the number of subjects was rather small for quantitative data analysis. The course of the experiment is summarized in Fig. 2. Subjects were always informed how long the task will take in advance, but the structure and the exact content were revealed only during the study. The assignments were given immediately prior to each task.

The whole package featuring Pogamut platform used, task texts given and questionnaires used can be downloaded from [29].

4.2 Participants

For the initial study, we recruited 22 students out of 33 attendants of the AI course. The study was the course's final exam and students were given their final grade based on performance of their agents in the Guide task. Students had the option of withdrawing from the study if they preferred a different kind of final exam.

We excluded 2 students from the analysis due to data incompleteness. In total, we analyzed data from 20 students. Students were randomly divided into two groups. The random assignment was stratified by year of study in order to guarantee that both groups contained similar number of students in each year of study.

For the follow-up task, we succeeded in recruiting 8 subjects (5 from the Java group and 3 from the POSH group) from the original 22. They participated for reward 30 USD. The number of follow-up participants was too small for statistical analysis, but provided qualitative data.

4.3 Materials

The Course. The students attended an introductory course on the control of virtual characters. The course is intended for students without previous AI or 3D graphics knowledge but with previous programming experience. Only students from the second or a higher year of study could attend. The course comprises of 12 theoretical lectures (90 minutes each) and 6 practical lessons at computers (90 minutes each). The theoretical classes are detailed in [21]. In practical lessons, the students are taught how to work with virtual agent's library (2 lessons) and develop behavior of virtual agents using both Java (2 lessons) and POSH (2 lessons).

Pretest. The general aim of the Pretest was to rule out subjects that were not sufficiently prepared for the final exam. Unprepared subjects would bias the data as they would likely fail during the final exam which would influence their answers in questionnaires.

The Pretest task was to create an agent capable of exploring the environment of UT04 game and collect items of a specific type. The agent had no adversaries in this task. Implementation language was assigned to subjects at random.

Three programmers skilled in VR technology solved the pretest task in advance to calibrate the difficulty of the test. The time allotment (2 hours) was at least three times longer than average time needed by these programmers to finish the task. Subjects had 3 attempts to pass the Pretest. Most passed on their first attempt.

Experimental task.

Guide. The Guide Task was to create an agent capable of finding a lost Civilian agent and leading it home. At the beginning, Civilian agent was standing still at random position broadcasting its position with a "mobile phone." The Guide agent must communicate with the Civilian agent if it wants the Civilian agent to follow its lead. The communication had a fixed and rather simplistic protocol described in the assignment.

Communication was reliable but Civilian was willing to reply to Guide over the mobile only if Guide was not too far away. Apart from finding Civilian, there were two obstacles that Guide had to overcome in order to successfully lead Civilian home. First, Civilian was willing to start following Guide only if Civilian can see Guide. Second, if Civilian lost Guide from sight, it stopped walking. Thus, the challenge was not only to find Civilian and persuade it to follow Guide home, but also to constantly observe whether Civilian is doing so. Once Civilian was home, it restarted itself in another location in the map. Guide's goal was to rescue Civilian as many times as possible within 5 minutes.

Guard. The Guard Task was an extension of the Guide Task. Again, the task of the Guard agent was to find and guide the lost Civilian agent home. However, there was also an adversary Alien agent in the environment created to hunt down both Guard and Civilian. Thus resulting Guard behavior must have correctly prioritized the following intentions: 1) finding a weapon, 2) finding and leading Civilian home, 3) responding to Alien. For instance, the Guard agent should have stopped leading Civilian when Alien was spotted and started attacking Alien.

MultiGuide (follow-up). The MultiGuide Task was also an extension of the Guide Task. This time, there were two Civilian agents in the environment and the MultiGuide agent had to get them together first and then lead both home.

Organization. For all tasks, subjects were told to code both low-level behavior primitives as well as high-level plans. Concerning the lower level, both groups used Java. For the higher level, i.e., organizing complex motor primitives based on already processed sensory information, Java group subjects hard-coded their own if-then rules or finite state machines (simple *switch* statements), whereas POSH group subjects had to use the POSH graphical editor for specification of a high-level POSH plan. POSH group was also encouraged to use the BOD design methodology. Tasks were solved by two skilled programmers in advance and their feedback was used to adjust the tasks difficulty.

Subjects received the assignment written on the paper prior to every task and they were provided with sufficient time (30 minutes) to read it and ask questions to clarify ambiguities.

Groups were working in parallel in two different rooms. Subjects were not allowed to cooperate on the solution but they were allowed to utilize any documentation about used virtual agent's library available on the Internet.

Questionnaires. Every subject received 4 questionnaires during the initial study and participants of the follow-up received an additional 2.

Questionnaires contained both quantitative (11 level Likert items; 10 maximum, 0 minimum) and qualitative questions. Questions were designed to 1) control for influences (comprehensibility of the assignment, task difficulty, whether the course has prepared subjects well, etc.), 2) investigate how appropriate was a language for a particular task, 3) report on language preferences, 4) report on how easy/hard was to extend the received code, 5) identify hard corners of Java/POSH behavior development. Follow-up subjects also undergone a structured interview.

4.4 Data analysis

All quantitative answers from the questionnaires were analyzed. Quantitative answers from the Guide task and Guard task post-questionnaires were compared. Answers from Java and POSH groups were also compared. We have used paired and two-sample t-tests with Welch approximation to compare the means in the two groups. Having discrete data, it would seem natural to look for methods using contingency tables (chi-square test of independence) or rank-based tests (Wilcoxon test, sign test or two-sample Kolmogorov-Smirnov test). However, using contingency tables here would suffer from low number of observations in cells while rank-based tests would suffer from lots of ties in our data. Moreover, the reasons for rejecting the null hypothesis may not be clear in some situations. Therefore, we have decided to compare the means observed in the two groups by applying paired and two-sample t-tests. Assuming that two-sided two-sample t-test is used to compare two groups of size 11 (see Sec. 5) and that the standard deviation is 2, the test detects difference 2 with probability approx. 60% and difference 3 with probability more than 90%. Notice that Central Limit Theorem guarantees that t-test may be used in this setup because the observed means are approximately normally distributed, see also [26, 27] for a more detailed justification of this approach. For other data from contingency tables, we have used χ^2 tests of independence with p-values obtained by Monte Carlo simulation in contingency tables. Additionally, agents from the Guide task were tested for quality. We executed the corresponding task scenario 10 times for each agent (Civilian's random position sequence has been fixed) and checked how many Civilians the agent saved in 5 minutes. The agent's score was computed as the average of all runs. Guard-task agents were not evaluated as most subjects solved this assignment only partially due to insufficient time and increasing fatigue (the study lasted 8 hours). Statistical tests were not run for the follow-up questionnaires and the follow-up agents were not tested due to the small number of participants.

5 Results

Results can be divided into objective performance of created agents and subjective assessment of the used tool. We will show quantitative data first and discuss qualitative data later. Only the most important data are reported due to space limitations.

5.1 Quantitative data

Quantitative data reports on:

A. how well the subjects understood the assignment; analyzing answers to the question “Have you understood the assignment?”;
B. how well the subjects were prepared for solving the task; analyzing answers to the question “Did practice lessons prepare you well for solving this kind of task?”;
C. how satisfied they were with the behavior they had created; analyzing answers to the question “How do you feel about the behavior you have produced? Is it ok?”;
D. the agent’s objective performance, in the case of Task 1;
E. how appropriate the tool the subjects were using was for solving the task; analyzing answers to the Guide Task’s question “Do you find Java to be the appropriate for the assignment?”;
F. satisfaction with comprehensibility of received code, in case of Task 2.

5.1.1. Task 1 - Guide Agent.

Ad A. Subjects in both groups understood the assignment very well and there were no between-group differences (mean for Java group = 9.36±0.77; mean for POSH group = 9.36±0.98).
Ad B. Subjects in both groups were equally prepared for the Task 1 (mean for Java group = 8.91±1.81; mean for POSH group = 8.5±1.9; p-value = 0.621).
Ad C. Subjects in POSH group were slightly less satisfied with their agents (mean for Java group = 7.64±0.67; mean for POSH group = 5.82±2.75; p-value = 0.056). The observed difference is not quite statistically significant, but given the low N we report the trend.
Ad D. Agents’ objective performances (Table 1) did not statistically differ between the groups (p-value = 0.722).
Ad E. Satisfaction of subjects with their programming tool in the Guide Task (Fig. 3, 4) was slightly higher in Java group but the difference was not significant (mean for Java group = 8.09±1.81; mean for POSH group = 7.09±3.51; p-value = 0.414).

Table 1. Task 1 agents’ performances.

Perf. / Group	Weak	Moderate	Good	Total
Java	3	4	4	11
POSH	1	4	4	9
Total	4	8	8	20

5.1.2. Task 2 - Guard Agent.

Ad A. Subjects in both groups understood the assignment very well and there were no between-group differences (mean for Java group = 9.46±0.99; mean for POSH group = 9.73±0.62).

Ad B. Subjects in POSH were prepared slightly better for this task (mean for Java group 6.67±3, mean for POSH group 8.8±1.3; p-value = 0.075). Again we report the trend due to the low N and weak power of the test.

Ad C. Subjects in both groups were similarly unsatisfied with their agents (mean for Java group = 3.82±2.09; mean for POSH group = 3.36±2.46; p-value = 0.646).

There was also highly significant shift of satisfaction visible when answers from both groups combined from Task 2 were compared to combined answers from the Task 1 (mean for Task 1 = 6.73±2.16; mean for Task 2 = 3.59±2.24; p-value of paired t-test < 0.001).

Ad E. Subjects' satisfaction with their tool did not differ between groups (Fig. 5, 6, mean for Java group = 6.67±2.06; mean for POSH group = 6.64±3.04; p-value = 0.979).

Ad F. We also have asked subjects whether they find the received code comprehensible. The result showed no between-group differences (mean for Java group = 5.8±3.6; mean for POSH group = 6.27±2.97; p-value = 0.747).

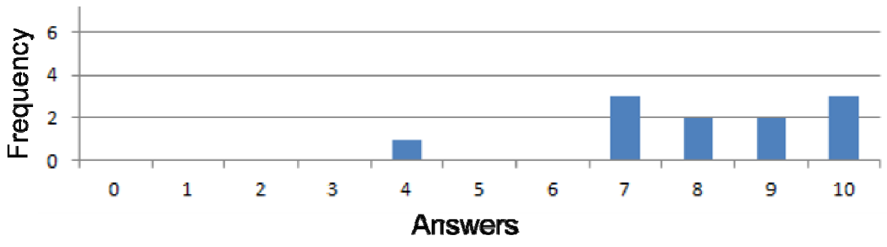


Fig. 3. Java group satisfaction with their tool (Task 1).

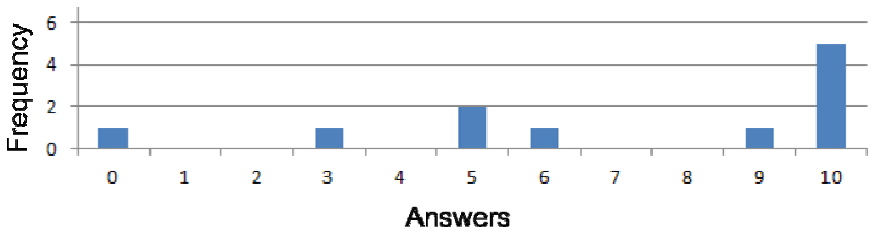


Fig. 4. POSH group satisfaction with their tool (Task 1).

5.1.3. Task 3 - MultiGuide Agent. All subjects reported that they understood the assignment perfectly (mean for both groups = 10±0). All subjects were able to extend their old code and create the MultiGuide agent. Interviews did not bring any dramatic comments on comprehensibility of code written in Java vs. POSH. Subjects from both group reported that reading through the code took around 10 minutes for both groups. Opinions regarding Java/POSH preference are included below.

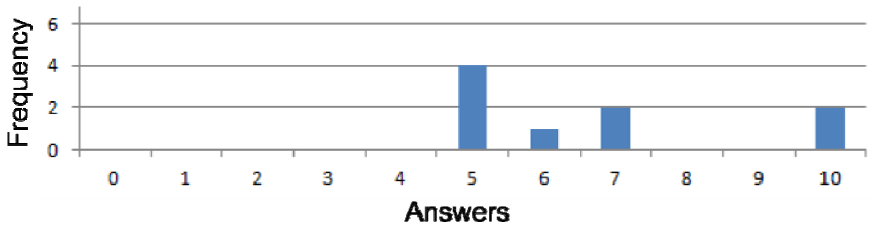


Fig. 5. Java group satisfaction with their tool (Task 2).

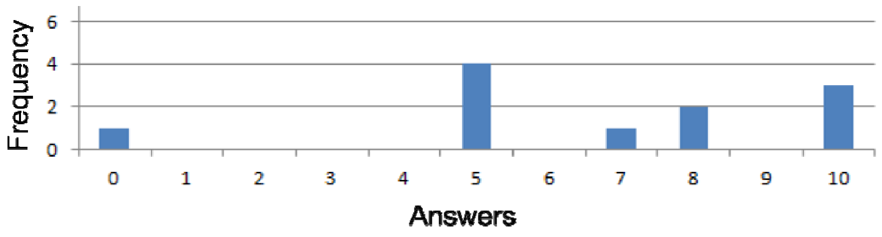


Fig. 6. POSH group satisfaction with their tool (Task 2).

5.2 Qualitative data

Quantitative results present an overall view on how subjects were satisfied with POSH or Java in the situations we modeled. These results have not revealed substantial differences between POSH and Java, suggesting that more fine-grained, qualitative approach is needed. Our qualitative data came from answers to "Explain" questions to abovementioned questions from questionnaires and from the interviews.
Answers can be divided into two categories: conceptual, pointing out strong and weak points of behavior design using POSH, and technical, such as wrong POSH engine settings. We will discuss mainly the former category as technical points might be eliminated easily by tweaking our POSH implementation.

Recall that POSH strictly separates behavior into a high-level plan, which uses behavior modules that define low-level code of behavior and sensory primitives (see Sec. 3). A well thought out POSH plan depicts how the agent will respond to the environment without revealing any technical details of the low-level code. When summarized, the qualitative data revealed rather strong, and opposite, opinions regarding this ability of POSH and its graphical editor: this feature was praised but also hated.

Many subjects found the separate thinking about the high-level behavior plan to be positive.

"I think it is pretty easy to make the idea in POSH and then just write few simple methods."
"The plan helps you to keep track of the important stuff that your agent does and reminds you to keep the behavioral triggers simple."
"In POSH, I can clearly distinguish states."
"Behavior states written in Java are harder to debug."

“POSH enforces good behavior architecture.”

However, some found that unsuitable to their style of work.

“The lack of variables at the level of POSH plans that would visualize flow of ***low-level*** *data from senses to actions seems limiting.” [POSH does not have variables at the high-level]*
“POSH limits you when you’re coding the behavior.”
“You still need to write Java code.” [note this is intentional in POSH/BOD approach, and cf. this with [16], who find it difficult to use vanilla Jason without underlying Java]
“Switching between POSH GUI and Java IDE was confusing me.” [refers to the necessity of switching between two modes of programming; the low-level and the high-level]

The last opinion contrasts with:

“POSH is a convenient way to clearly write agent decision logic and underlying Java is powerful enough to code all details.”
Some users failed to see any advantages in POSH at all or at least in having a separate graphical program and action-selection mechanism to run it.
“I can simply write POSH decision tree in Java.”

We noticed that POSH subjects cannot program the required behavior exclusively inside the high-level plan. The subjects always coded also their own low-level POSH primitives or made changes to primitives of other subjects (Task 2) or their own (Task 3).

POSH was frequently criticized with technical comments. Students usually disliked writing names of actions and senses twice, first in POSH and then in Java. But there were a few comments that revealed some conceptual flaws of POSH behavioral language as well.

“POSH has fixed order of action priorities; this becomes too limiting for complex behaviors.” [that points to POSH’s (intentionally) simple conflict resolution mechanism]
“POSH does not provide any mechanisms for action-switching, it is hard to track that for yourself.” [like in many other agent-based systems, support for transition behaviors, including *action-in* and *action-out* constructs, is limited or none, see also [25]]
“POSH does not support parallel behaviors; parallel behavior is especially hard to manage.” [original POSH used on robots allowed for parallel behaviors, but this is more difficult in the present VR incarnation due to the game engine]

Qualitative data provide interesting points for further discussion. Some points can be generalized to other agent-based languages.

6 Discussion and conclusion

This study has compared the usability of the academic AS system POSH empowered with a graphical editor to that of a common programming language Java in two situations common in a game company: a) catching up upon the work of a colleague, and b) extending one's own work from several months ago.

Unfortunately quantitative results could only be gathered on two of the three tasks we assigned. Here we showed no difference between Java and POSH groups in subjectively reported readiness for utilization of the tool in Task 1 (see Sec. 5.1.1.B). Subjects from POSH group reported they were prepared slightly better for Task 2, which could be to POSH's advantage, but the effect was rather small (see Sec. 5.1.2.B). The qualitative data seem to argue that we prepared the students well for the tasks no matter the technique; the groups were not biased. Because there are no differences in objective agents' qualities (see Sec. 5.1.1.D), the first hypothesis that POSH is better in terms of usability and efficiency of resulting behavior is not supported.

The second hypothesis also has not been supported by the quantitative data from the first condition, as subjects did not report improvements to the code's readability due to POSH's visible organization of the sensory and behavior primitives into a high-level plan (see Sec. 5.1.2.F). However, verbal comments are interesting. Whereas complete freedom of coding high-level behavior in Java was praised by some Java group subjects, it was a source of confusion for others. For POSH, negative comments were focused only on complexity of behavior primitives in low-level code, constraints of the high-level language, but never on the problems with the high-level plan comprehensibility (see Sec. 5.2 and below).

"Single routine from hell." [a Java subject referring to a single Java method that executed the whole behavior]
"The logic method was a long list of ifs that were kinda obscure and it was unclear to me which part was taking care of which part of the behavior." [a Java subject referring to overly complex if-then rules in Java, which were mixing high-level behavioral code with low-level code]

That contrasts with negative comments of POSH subjects related to the low-level code written by a different subject:
"Senses and actions were quite complex."
"Some of the primitives were unfamiliar; there was some extra stuff I did not understand."
"The naming was good, but there were about 5 senses/actions that didn´t do anything."

Finally, the third hypothesis also has not been supported (see Sec. 5.1.3). Subjects from both groups did not report any problems with reading own code that they had created 3 months ago, even if they had not been interacting with the code all over the period.

This last may indicate that Task 1 and 3 were too simple to get much advantage from a programming tool, at least for the 8 programmers who had completed Task 2 and were willing to come to their code again. This is particularly true since the basic structure of POSH could be indeed replicated with Java conditional statements if the hierarchy or plan was not too complex. Had we been able to complete the full course of the study with all programmers, we may have found subjects that the POSH structure assisted.

6.1 Main interpretation

It is useful to conceive the results from the standpoint of the metaphor separating behavioral code into the "low-level" and "high-level". When adopting this perspective, the results argue that tasks of a medium complexity (compared to common tasks of an industry programmer) already imply programming at both levels, and consequently, switching the programmer's attention between the levels. Note that POSH/BOD already recognizes that and the study of Píbil et al. [16] also supports this interpretation. However, this cast doubts on the idea that non-programmers, such as game designer, could ever use "intuitive high-level languages" *only*, except for the simplest tasks.

An interesting point is that majority of subjects seem to praise the separation of high-level behavior plan from the low-level code, which is a general finding, but they were not satisfied with concrete limitations that POSH enforces on the architecture of behavior primitives. Still, some subjects seem not to have internalized their thinking in terms of this two-level architecture at all and to have problems with switching between levels of abstraction.

What we still do not know is whether the explicit materialization of the low-level / high-level separation realized in POSH/BOD and agent-based languages in general, would eventually turn out to be more of an advantage than a burden. The fact that students think the former does not necessarily mean it really is. Some qualitative data concerning Guard task and one quantitative outcome (see Sec. 5.1.2.B) suggest that at least when one has to read the code of some else, the explicit materialization of high-level constructs is an advantage. At the same time, however, as said above, some qualitative data suggest that some students may have problems when the interface between two levels is explicit. This might be similar to object-oriented programming; one has to undergo a long journey to fully appreciate the concept, and perhaps some programmers are always happier in assembly. Future research is needed to elucidate what exactly is a POSH's and its GUI's technical limitation and what is a deeper conceptual issue.

6.2 Generalization

Many comments on Java vs. POSH can be transposed to other academic AS systems due to general approach they all share with POSH. All of them try to separate behavioral code out of low-level code. We will now summarize the study's results into the list of guidelines that should be considered when assessing AS systems for the purpose of authoring behaviors for virtual agents.

1. The study's result supported the idea that low-level code should be used for coding behavior primitives. A high-level AS system should not try to supply processing of sensory information or attempt to supply logic of low-level actions directly. An AS system should understand that behavior primitives always need to be created in low-level code forming the agent periphery and provide appropriate support for organizing it.
2. The interface of an AS system with low-level code should be simple and interface requirements should be assessed as they will indicate design patterns a programmer will need to follow. If those design patterns are complex or over-constraining, as is the case of parameter-less sensor and action methods in POSH, it may lead to time consuming implementation of agents' peripheries.
3. An AS system should be prepared for the execution of transition behaviors. Whenever an AS system decides it is time to switch from one action to another, it should also notify low-level code it is doing so, i.e., it should be part of AS interface to the low-level code.

From the methodological perspective, the lessons learnt from this study are that both quantitative and qualitative data are useful for assessing engineer performance.

7 Acknowledgements

Research is supported by the Czech Science Foundation under the contract P103/10/1287 (GACR), by student grant GA UK No. 0449/2010/A-INF/MFF, by student grant GA UK No. 655012/2012/A-INF/MFF and by SVV project number 263 314. Human data were collected respecting APA ethical guidelines. We kindly thank Joanna Bryson, University of Bath, UK, for her consultations regarding design of the experment and data analysis.

8 References

1. Fu, D., Houlette, R., "The Ultimate Guide to FSMs in Games," AI Game Programming Wisdom II, Charles River Media (2004): pp. 283-302.
2. Champandard, A. J.: Behavior Trees for Next-Gen Game AI. Internet presentation. URL: http://aigamedev.com/insider/presentations/behavior-trees (11.10.2011)
3. Schuytema, P.: Game Development with Lua. Charles River Media (2005)
4. UnrealScript programming language. URL: http://udn.epicgames.com/Two/UnrealScriptReference.html (11.10.2011)
5. Schwab, B.: AI Game Engine Programming. 2nd edition. Charles River Media. (2008)
6. AiGameDev community. URL: http://aigamedev.com/ (11.10.2011)

7. Rabin S.: AI Game Programming Wisdom series. URL: http://www.aiwisdom.com/ (11.10.2011)
8. Gamasutra webpage. URL: http://www.gamasutra.com/ (11.10.2011)
9. Magerko, B., Laird, J. E., Assanie, M., Kerfoot, A., Stokes, D.: AI Characters and Directors for Interactive Computer Games, Proceedings of the 2004 Innovative Applications of Artificial Intelligence Conference, San Jose, CA, July 2004. AAAI Press (2004)
10. Best, B. J. & Lebiere, C.: Cognitive agents interacting in real and virtual worlds. In: Sun, R. (Ed) Cognition and Multi-Agent Interaction: From Cognitive Modeling to Social Simulation. NY, NY: Cambridge University Press. (2006)4
11. Hindriks, K. V., van Riemsdijk, M. B., Behrens, T., Korstanje, R.,Kraaijenbrink, N., Pasman, W., de Rijk, L.: Unreal GOAL Bots: Conceptual Design of a Reusable Interface. In: Agents for games and simulations II, LNAI 6525, pp. 1-18. (2010)
12. Bryson, J.J.: Intelligence by design: Principles of Modularity and Coordination for Engineering Complex Adaptive Agent. PhD Thesis, MIT, Department of EECS, Cambridge, MA. (2001)
13. Partington, S.J., Bryson, J.J.: The Behavior Oriented Design of an Unreal Tournament Character. In: Proceedings of IVA'05, LNAI 3661, Springer-Verlag (2005)
14. Köster, M., Novák, P., Mainzer D., Fuhrmann, B.: Two Case Studies for Jazzyk BSM. In: Proceedings of Agents for Games and Simulations: Trends in Techniques, Concepts and Design, F. Dignum, J. Bradshaw, B. Silverman and W. van Doesburg (ed.), AGS 2009, LNAI 5920 (2009)
15. Bryson, J. J.: Behavior-Oriented Design of Modular Agent Intelligence. In: Agent Technologies, Infrastructures, Tools, and Applications for e-Services, R. Kowalszyk, J. P. Müller, H. Tianfield and R. Unland, eds., pp. 61-76, Springer. (2003)
16. Píbil, R., Novák, P., Brom, C., Gemrot, J.: Notes on pragmatic agent-programming with Jason. In: Proceedings of Programming Multi-Agent Systems, AAMAS workshop, Taipei, Taiwan, pp 55-70. (2011)
17. Gemrot, J., Brom, C., Kadlec, R., Bída, M., Burkert, O., Zemčák, M., Píbil, R., Plch, T. Pogamut 3 – Virtual Humans Made Simple. In: Advances in Cognitive Science, Gray, J. eds, The Institution Of Engineering And Technology (2010) pp. 211-243
18. Bryson, J. J.: Action Selection and Individuation in Agent Based Modelling. In: Proceedings of Agent 2003: Challenges of Social Simulation, Argonne National Laboratory (2003) pp. 317-330
19. Hindriks, V. K., van Riemsdijk, M., Jonker, B., C. M., 2011, An Empirical Study of Patterns in Agent Programs: An Unreal Tournament Case Study in GOAL, PRIMA 2010.
20. Heckel, F. W. P, Youngblood, M., Hale, D. H.: Behavior Shop: An Intuitive Interface for Interactive Character Design. In: Proceedings of the Fifth Artificial Intelligence and Interactive Digital Entertainment Conference, AIIDE 2009, October 14-16, 2009, Stanford, California, USA. The AAAI Press (2009)

21. Brom, C.: Curricula of the course on modelling behaviour of human and animal-like agents. In: Proceedings of the Frontiers in Science Education Research Conference, Famaguta, North Cyprus. 2009, pp. 71 - 79
22. Bordini, R.H., Hübner, J.F., Wooldridge, M.: Programming Multi-Agent Systems in AgentSpeak Using Jason. John Wiley & Sons, Ltd. (2007)
23. Brooks, R.A: Intelligence Without Representation, Artificial Intelligence 47 (1-3) (1991) pp. 139-159.
24. Desai, N.: Using Describers To Simplify ScriptEase. In: Master Thesis. Department of Computing Science, University of Alberta, Edmonton, Alberta, Canada. (2009)
25. Plch, T.: Towards Believable Intelligent Virtual Agents with StateFull Hierarchical Reactive Planning Action Selection, In: Proceedings of Week of Doctoral Students, Charles U in Prauge(2011), in press.
26. Rasch, Teuscher, Guiard, How robust are tests for two independent samples?, Journal of Statistical Planning and Inference, Volume 137, Issue 8, pp. 2706-2720.
27. Heeren T, D'Agostino R. Robustness of the two independent samples t-test when applied to ordinal scaled data. Stat Med. 1987 Jan-Feb;6(1): pp 79-90.
28. Gemrot, J., Brom, C., Bryson, J., Bída, M.: How to compare usability of techniques for the specification of virtual agents' behavior? An experimental pilot study with human subjects. In: Proceedings of Agents for Games and Simulations, AAMAS workshop, Taipei, Taiwan. (2011) pp. 33-57
29. Experiment reusable packages: http://pogamut.cuni.cz/pogamut-devel/doku.php?id=human-like_artifical_agents_2010-11_summer_semester_exam_info (9.4.2012)