Enhancements for Reactive Planning – Tricks and Hacks

Tomáš Plch, Cyril Brom
Charles University, Faculty of Mathematics and Physics
Malostranské nám. 2/25, Prague, Czech Republic
tomas.plch@gmail.com, brom@ksvi.mff.cuni.cz

Abstract. Behavior based action selection is utilized for agents residing in complex and unpredictable virtual worlds. The techniques based on reactive planning, where rules are organized into a hierarchy where single plans consist of if-then rules ordered by priority, are popular not only for their simplistic design but also for the possibilities they provide in modeling believable behaving agents. In this paper we propose simple and easy to integrate enhancements for the if-then rule based behavior models, providing the designer with more possibilities to control the designed behavior and to overcome the by us observed limitations of the basic reactive planning concept. The result is an improved formalism for reactive rules, which could be utilized for more believable behavior modeling. The proposed concept upgrades are designed to be backward compatible with the basic concept.

Keywords: If-Then Rules, Reactive Planning, Behavioral Tree, Virtual Agents, Believable Behavior

1 Introduction

Virtual agents are becoming very popular in both academic and industrial domain, used in computer simulations and various applications. Thanks to the growing processing power of commonly available computer equipment, they also became appealing for general public, most notably in computer games [1], interactive worlds and education in the field of artificial intelligence [2].

The application of virtual agent's range from movie industry, computer games, and academic studies [3], trough virtual storytelling [4] to military simulations [5, 6] and civilian applications [7].

A virtual agent can be viewed as an autonomous system that processes the percepts from the environment it inhabits (*virtual world*) and takes actions satisfying a goal or following a task [8], given by the designer or acquired by the agent himself - from other agents, from the environment or from a perception of the world. The agent can be put into a dynamic and unpredictable environment, thus making it more difficult to achieve its goals in timely and believable fashion. A virtual agent might be equipped with a virtual body, allowing interaction with the environment in a more realistic way.

Most researchers now agree, that intelligence is a manifestation of behavior [9]. Systems that use behavior to model intelligence are known as *Behavior-based*

Systems (BBS) [10]. The idea of BBS is based in ethology [11] – a branch of biology that studies animal behavior patterns.

The main concern in virtual agent research and application is to mimic intelligence by behaving in a believable fashion, focusing on visualization and action selection. The aspect of proper visualization gives the agent and its virtual world a more believable look, where selecting adequate actions in a timely fashion gives the feel of intelligence. The idea of intelligence manifested trough behavior requires from the agent to provide acceptable behavior. An agent that bumps ten times into a wall or takes minutes to decide its next action can be considered inadequate for most applications when observed in real-time (games, online simulations etc.).

The purpose of this paper is to propose improvements for the broadly adopted techniques of reactive planning. We propose a set of changes to the formalism of reactive rules that are the building block of reactive plans. Our improvements are aimed at improving the believability of the agent's behavior. The improvements have been implemented into a simple prototype. We intend to create a more detailed implementation within a complex virtual world.

Section 2 describes our motivation. Section 3 describes the techniques of reactive planning in further detail. Section 4 introduces our improvements to the reactive rules. Section 5 reviews other approaches related to our proposal. Section 6 contains the conclusion and intended future work.

2 Motivation

The *Pogamut platform* [13], which is under heavy development at our laboratory [14], gains in popularity not only in the educational process, but also in fan and even academic based [15] development of believable *non player characters* (NPC's) also known as *bots*, *virtual characters* (VC) or agents. The usage of reactive planning is popular thanks to the simplicity of the overall approach providing a strong and versatile tool in the creation of a believable and agile agent.

Our main motivation is to provide the users of these techniques with a better tool, but keeping true to the basic principles of reactive planning, thus making the resulting NPCs even better and more fun to play with/against.

The need to keep the overall concept as simple as possible, providing tricks and hacks with backward compatiblity is essential because Pogamut in conjunction with reactive planning is an important topic for students who try to dive into the topic of artificial intelligence and virtual characters.

The tricks and hacks presented in this paper should essentially provide more control over the agent's resulting behavior, possibly leading to more believable behavior. It is important to realize, that the resulting behaviour is largerly the result of proper application of the basic techniques with the proposed hacks and tricks to augment the agent.

3 Reactive planning in detail

Reactive planning [16] denotes a set of techniques for *action selection mechanism* (ASM) used to handle *autonomous agents,* residing in unpredictable and dynamic (virtual) environment. Reactive planning exploits the idea of *reactive plans,* which consist of *condition-action* rules.

A condition-action, also known as *if-then* rule, has a simple form

if *condition* **then** *action*

representing a simple concept – ***if*** *condition* holds ***then*** perform action. Action selection mechanism (ASM) evaluates these rules at every given opportunity (periodically) choosing next one to be performed. This can be viewed as choosing the next step. The checks might be performed at strict time intervals – ticks, or after the currently executing action ends (various actions might take various amounts of engine ticks).

In a *reactive plan*, the condition-action rules can be ordered by their *priority*. A *rule* is chosen for execution based upon its priority and the *holding condition,* maximizing the *priority.* Every action can be either *external* (influencing the world or physical appearance of the agent) or *internal* (influencing the agent's mind).A rule that is chosen for execution is called *active* or *executing.* Rules with holding conditions are called *preactive* and all others are *inactive.* Rules, which were active but are surpassed by higher priority rules, are called *suspended*.

Plans can be organized into hierarchies – a behavioral tree (*be-tree*) [17]. The root of the be-tree is a Top-Level goal - "do exist". The nodes in the be-tree are reactive plans and the leafs represent action primitives – atomic actions, action sequences, sequences of action primitives, etc.

The if-then rule can be formalized as a triplet - *{prio, cond, exec}*.

- *prio* denotes the priority of the rule in the containing reactive plan
- *cond* is the boolean-based condition, also called a *releaser*
- *exec* is the performed action

The concept of reactive planning is designer dependant – the AI designer creates the reactive plans and aligns them into the hierarchy that forms the NPC's brain. The plan's structure does not change during the agent's runtime, remaining static.

3.1 Action selection mechanism (ASM)

The idea of the ASM is to recursively follow the *be-tree* structure until a leaf is reached. The algorithm chooses the highest priority rule with a holding condition and follows it if possible. The rule is either a reactive plan (the algorithm steps down in the be-tree structure) or action primitive (and results in executing an atomic action). When the newly chosen leaf (action primitive) differs from the previously executed one, the previous one is suspended or forced to fail. The situation, where no rule can be chosen, can be handled differently, depends on the implementation of the algorithm. When choosing amongst two or more rules with the same priority, the choice is random.

4 Reactive rules updated

We introduced the already well known and commonly used concept of reactive planning. In this section, we present our variation to the structure of the rules, which we believe, might provide practical benefits when applied. First we introduce the by us observed problems with the hierarchical variant of reactive planning – with the basic-simple ASM presented above.
The observed problems of reactive planning can be summarized into five categories:

- interrupting
- condition fault
- delaying rule activation
- failure & success
- random rule selection

4.1 Interrupting – behavior consistency

Humans often abstract and view behavior as something more complex than just simple atomic actions, perceiving sequences of actions that have to follow distinct order and keep consistency. Interrupting consistent behavior patterns can be viewed as a disorder.

We need to take constrains of the virtual world into account. Some action sequences are demanded to be uninterrupted and consistent, to keep the visualization or environment of the virtual world intact.

The *interrupting* can pose a serious problem for the ASM. When demanding longer sequences to be consistent and the environment is dynamic enough to provide input for ASM to interrupt those sequences, without modification, behavior consistency cannot be (easily) guaranteed.

More complex behaviors may not be possible to boil down to single atomic actions. When introducing more complex action structures, like action sequences or action loops, it can be desirable for some of these structures, to be uninterruptible – *interrupt-safe*.

We propose an interrupt-safe flag for the rules. The rules having the flag cannot be replaced/surpassed by a higher priority rule chosen by the ASM. An interrupt-safe rule stays active until finished, or is not interrupt-safe anymore. The flag can be understood as an override for condition checks and action selection, allowing the rule to execute unhindered. The interrupt-flag is therefore a tool to let sequences of multiple actions appear atomic[1].

The example shown in Table 1 is designed based on behavior of computer game players. Running around in the virtual world of an action game being low on ammunition in a weapon is risky. Encountering an enemy with almost no ammunition can lead to reloading in mid-battle conditions, which is the worst possible scenario. Reloading regularly, when no enemy is around and when low on ammunition, is common, even when potentially dangerous - an enemy could catch upon you while

[1] "Atomic like sequences" are atomic from the execution style point of view, still taking multiple cycles to perform. Atomic actions take only one cycle to perform

reloading. When the enemy is sighted even while reloading, the reloading has to be finished and then the enemy can be engaged.

Table 1. Early reload behavior – the snippet of a reactive plan used by an simple agent in a first person shooter.

Priority	Condition	Action
4	No ammo	drop(current magazine) take(full magazine) reload
3	See (enemy)	shoot(enemy)
2	Low on ammo	store(current magazine) take(full magazine) reload

The main problem in this scenario is to delay shooting an empty weapon when an enemy is sighted during *low-on-ammo reloading*. Therefore, the *reload-when-low-on-ammo* action sequence can be flagged interrupt-safe. The agent will finish reloading the weapon and after that, having the enemy still in sight, start shooting. We call this *behavior consistency* – the agent behaves more consistent in its behavior patterns, not switching wildly between rules.

With responsiveness of the agent in mind, the interrupt-safe action sequences have to be as short as possible, to prevent the agent from being stuck in long *interrupt-safe* sequences, missing out on the world.

When large sequences of atomic actions are used, marking the whole sequence can be considered futile or undesired. We propose introducing the interrupt-safe flag not to the action sequences (or other action primitive containing more than one atomic action) as a whole, but as a boolean flag for atomic actions (within the action primitive) - creating *interrupt safe zones* within.

Table 2. A sequence with a interrupt safe flag used for atomic actions creating interrupt safe zones.

Priority	Condition	Action	Interrupt safe flag
1	true	a1	0
		a2	1
		a3	1
		a4	0

There is a sequence of six actions {a1, a2, a3, a4} presented in Table 2. The interrupt safe zone starts at action a2 and stops at action. Employing the concept of interrupt-safe zones can render large action sequences more adaptable, allowing a better structured design of longer sequences. The interrupt-safe trait propagates upwards trough the entire be-tree, meaning when a child node is considered interrupt-safe, its parent is interrupt-safe as a consequence.

It can be argued that the behavior shown in Table 1 can be better expressed, introducing more complex releaser configuration not needing the interrupt-safe flag at

all. It is computationally cheaper to use a boolean flag than a complicated rules (i.e. releaser) configuration. .Mimicking interrupt-safe zones behavior employing only releasers for rules can be considered an unmanageable task for the AI designer to handle.

4.2 Condition fault

A condition fault is a situation, where the *condition* of the rule ceases to hold in mid-execution of an associated action sequence, either by external or internal (by the sequence itself) causes. Plain ASM will not continue executing the action, not considering it in the search of a candidate for the active rule. The condition fault will not occur when only single atomic actions are employed.

This represents a significant problem, when action sequences disrupt their own condition, rendering it false, ceasing their execution before they finish (i.e. they will "never" finish, always corrupting the condition).

Let us consider a simple rule shown in Table 3 in a reactive plan of a virtual character representing a dog.

Table 3. A rule in a reactive plan that will cause a "condition fault"

Priority	Condition	Action sequence
1	Next to(meat)	Smile, Eat, Bark

When the dog eats a piece of meat the dog stands next to, the condition will cease to hold in the next iteration (because the meat has been eaten, and the dog stands next to nothing) and the last action – bark – is never executed. The rule itself *"breaks"* its own condition – corrupting its own context. The condition could fail due to events in the virtual world (someone eats the piece of meat before the dog does), leading the "eat" action eventually to fail. The bottom line is the question: *What will happen with the rule when the condition is evaluated false in mid-execution of an action sequence, reactive plan or an execution vehicle that takes more than a single atomic action to finish?*

In ABL language [18], two conditions/releasers are used to cope with this problem - *precondition* and *context condition*. The precondition, when satisfied, puts the rule into an active state. It is a releaser in a more literal sense. The context condition has to be evaluated true during the execution of the rule's actions. During execution, the precondition releaser is not taken into account. When the context condition fails, the rules execution is stopped – it fails.

We propose a different approach, where a rule can be marked as *releaser-safe*. When marked and its condition fails to hold, it is ignored by the ASM and the releaser is considered "true". The mark of a releaser-safe rule acts as an override – even when a condition fail occurs, the failed condition (evaluated false) is ignored and considered "true". This can be implemented either by a special flag similar to the *interrupt-safe flag* or simply by adding (temporarily) an "OR true" to the condition's logical formula, rendering it always true when evaluated.

It is noteworthy to say, that the releaser-safe trait serves as an override only when the rule is in execution and its condition fails. The releaser-safe flag does not provide an override for a not holding condition for a not executing rule.

The concept used in ABL language is more powerful then the releaser-safe flag, providing a better but more complex tool. Our approach is less computationally demanding, providing a faster option for a speed oriented design.

4.3 Delayed rule activation

Delayed rules are a problem tightly bound to the interrupting issue presented earlier. Rules of higher priority that have a holding condition should be chosen by the ASM, but they cannot, because of an executing rule with consistent behavior (interrupt-safe). It can be deemed necessary to put such rule to execution anyway, possibly later. Based on this assumption, the rule has to be delayed and considered by the ASM later, even when the condition of the delayed rule ceased holding (it held in the past). Compensating this problem can lead to more believable behavior, keeping the rule execution in a more consistent appearance – rules will execute, with a short, hardly noticeable delay.

Let us consider a rule R1 to be interrupt-safe and executing an action sequence {a1, a2, a3, a4, a5}. A higher priority rule R2 gets active during the execution of the R1 action sequence. After the action sequence finishes and allows other rules to be active and executed, the condition for R2 does not hold anymore. It might be desirable that R2, when its releaser holds, is performed (even when a little bit later).

It could be a response to some high priority event – seeing an *enemy*. We know that the *interrupt-safe* sequence of R1 is short, but the conditions in the virtual world that lead to the activation of R2 could change rapidly – the *enemy* hid behind an object.

We need for the R2 rule to be delayed, to stay in the *preactive state*, until it can be executed or considered a candidate by the ASM. Therefore we introduce a *sticky-rule flag* to mark rules activation of which has to be delayed if they cannot be executed for some reason at the present time.

The basic idea is that the rules, when activated, become "*sticky"* and even when the condition does not hold anymore, they are considered as a choice for the ASM at a later point. Eventually this could lead to a lot of pending low priority *rule*s executed long after their releaser held. Therefore, validity of *sticky-rule flag* should be limited by a *timeout*.

This concept can be viewed as a simple memory, where information about delayed execution during longer *interrupt safe* executions, is stored.

4.4 Fail & Success

The issue of fail and success can be divided into two categories – *reactive plan event* and *reactive rule event*. The reactive plans can fail or succeed only by executing an explicit atomic action (i.e. fail/success action) and the plan fails/succeeds as a whole. The reactive rule, on the other hand, can fail due to various reasons – the contained

action primitive has failed (e.g. when an atomic action within an action sequence could not finish) or the condition of the rule failed and the rule was not marked as releaser-safe.

The issue of fail and success within the hierarchy of reactive plans raises two questions:

1. How many times can they occur?
2. How to propagate fail and success event?

The first question originates from a simple scenario, when the ASM does not recognize a repeating faulty behavior of an agent, e.g. the agent stands in front of a locked door trying to open it repeatedly. This renders the agents behavior less believable.

The second question originates from the need to deliver the event of fail/success further upward in the hierarchy of plans, e.g. the agent fails an important plan deeper in the hierarchy and it requires to render the whole subtree (rooted upwards) to fail.

How many times to fail/succeed?

Let us consider the simple example presented earlier, where the agent opens a door in front of him. The door is locked, so the action fails. The agent's *door-open rule* has a high priority and therefore, even when lower priority rules can be active, the agent retries the door-open rule failing multiple times.

To overcome this, we propose a *finite amount of fails/successes* to be specified for a rule and after using them up, the rule will be *disabled* until the plan is *reset*. We propose to monitor the amount of events by an integer counter – *fail/success counter*. A disabled rule is excluded from the ASM on the current plan. A similar approach was already presented in the extended POSH architecture [19].

There is the issue of *resetting the rules/plans* – we need to distinct a *hard reset* and a *soft reset*. The hard reset resets the fail/success counters that keep track of how many times the rule actually had failed/succeeded. The soft reset only resets the rule (e.g. resets the counters of a loop action sequence back to 0), keeping the fail/success counters intact.

The hard reset is invoked when the rule is forcefully aborted *"from above"* (in the hierarchy of plans, i.e. an above plan stops to execute – succeeds or fails), where the soft reset is invoked when the rule *fails on its own.* A condition fault induces a forced fail, which is considered to be "from above".

To summarize the reset of rules – when either finished or terminated in mid execution (e.g. action sequences, reactive plans etc.), the rules and the part of the hierarchy beneath the point of reset have to return to a state from which they can be executed again – i.e. reset. A hard reset differs from the soft reset in that it returns fail/success counters to their default value.

How to propagate failure/success?

A *fail/success* of a plan or rule can happen at any level and in any branch of the hierarchy, raising the question, "How to deal with *it*?". As seen in Fig. 1, the event has to propagate to the parent of the source, to report what has happened and to the children plans, to either abort or fail them. The children plans are either the active plan or some of the suspended plans that are rooted under the plans rules.

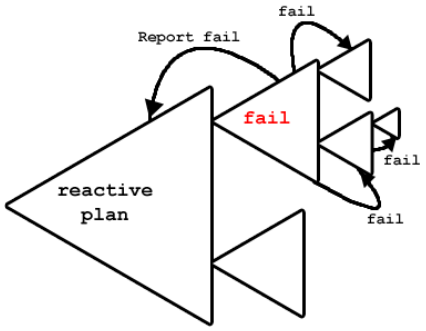

Fig. 1. The propagation of a failure event - the event is propagated upward and downward. Downward - to abort the suspended branches and fail the executing branch (if such exists). Upward - reporting this event to the above plan (in the hierarchy - a parent)

The issue of proper propagation can be seen from the upward and downward point of view. The upward direction presents a problem of "how far to go". Sometimes, "up to the top" approach may be required. In other cases, only a one or two levels up propagation is necessary. For the upward propagation we propose to use an integer *counter* for the propagating fail that indicates how far an event has to be reported. An additional *compensation counter* can be introduced into the plan structure, representing how much (having the default value set to 1) to subtract from the propagating of an event *counter*.

At every level of the hierarchy the event passes, the counter is decreased by the compensation counter and stops at the level where it reaches zero (or less). As the event propagates upward, it passes levels within the hierarchy until it reaches the root. Within those levels, two kinds of rules can be encountered – the suspended rules and at most one active rule. Under those rules, action primitives or reactive plans are rooted.

In the case of a *suspended* rule, we *abort* the rule – the post-execution responsibilities (like cleaning up objects etc.) are taken over by the *origin* of the forced abort. The suspended rule is *not* given any execution time, it is simply pronounced as finished. The *active* rule is forced to *fail,* but providing execution time to deal with it. Therefore, when the propagation of an event reaches a level, where an active rule is present, the propagation continues first in the downward direction until the whole active branch can be pronounced as failed (i.e. finished). After all rules within a level are failed or aborted, the propagation can continue upwards (until the counter reaches zero or less).

4.5 Random rule selection

The random choice mechanism for rules with equal priorities can be insufficient. In some cases, a designer could prefer some rules based on their significance. When multiple rules of the same priority are a candidate to be active, a winner is chosen from their ranks based upon unbiased random selection. How unbiased random

selection can result in less believable behavior is shown on a simple *dog* based example in Table 4.

The dogs tend to run around and bark. He does not choose these actions on an equal basis, but prefers to run around a lot, occasionally barking. When unbiased random rule selection is employed, this renders a minor but recognizable drawback.

Table 4. A snippet of reactive plan for a simple virtual dog character, which only barks and runs around

Priority	Releaser	Action
0	True	Bark
0	True	Random run

We propose to expand the priority by adding *weight* of rules to the concept.

Table 5. A snippet of reactive plan for a extended virtual dog character, which barks and runs around on a unbiased basis

Priority	Weight	Releaser	Action
0	20	True	Bark
0	60	True	Random run

A rule should be chosen based upon its weight amongst the equal priority candidates. Rules shown in Table 5 have their weight set. The *"Random run"* rule has a weight of 60 and the *"Bark"* rule has a weight of 20. When both are chosen as candidates (their releaser holds), the "Bark" rule has a chance of 25% and the "Random run" rule has a chance of 75% to be chosen.

4.6 Rule formalism revised

Based upon the observed problems of ASM, we propose an update on the rule formalism.

A reactive plan is a set of rules, where a rule is a tuple
{ prio, w, cond, action, flags, sticky timeout, fails, successes }

- *prio*, *cond* and *action* semantics do not change
- parameter *w* denotes the weight of the rule
- parameter *flags* is composed of flags for *interrupt-safe*, *releaser-safe* and *sticky rule* boolean flag (it can also be used to store other flags)
- parameter *sticky timeout* specifies, how long until the sticky-rule expires. (the timeout should be reset every time the rules releaser holds)
- parameter *fails/successes* specified how many times a rule can fail/succeed in its execution. When the parameter reaches zero, the rule is disabled from further execution, until the plan is (hard) reset

5 Related work

We took the inspiration for our work from Bryson's work on reactive plans [12, 19]. Our approach differs by the formalism extensions we use, because we aim to provide our agents with broader and more complex plans and behavior than Bryson.

Similar work on complex agent behavior can be seen in game projects like Halo2© [20], where the work was aimed at the hierarchy structure containing static plans, where the main advantage is derived from the modularity of the hierarchy itself, with optimization for low memory consumption and large scale use (multiple agents) in mind. Our work is aimed at the single rule structure, where the plans are only vehicles for the rules and the overall hierarchy is static. Our rules can induce changes to the structure to a certain degree (fail/success propagation).

The ABL [18] is aimed at specifying not only the agent's behavior, but also on creating entire simulations with the agents embedded in a "reactive" scenario. We were inspired by some concepts employed within the ABL, but we are more concerned with the individual agent's brain and improving the agent's behavior believability to the human observer.

6 Conclusion and Future work

The tricks and hacks we presented in this paper should easily be implementable as extensions providing potentially powerful modification for the basic reactive planning concept. When properly used, they can provide more control and lead to better results in the context of an agent's behavior.

The reactive rule extensions are backward compatible, so their use can be thought at a later point in a course teaching the VC basic topics like reactive planning, simply by extending the prior learned concepts. We intended to provide a set of useful techniques, which can aid the user when expressing his idea of the agent's behavior with the computational complexity of the result in mind, to keep the timely fashion of reactive planning intact.

We intend to implement the concepts presented in this paper into the Pogamut platform - to prove that these concepts can live up to their expectations and test them on competing agents in the domain of games like Unreal Tournament 2004 and other similar first person shooters. We already created a pilot prototype implementation [21] of these techniques to prove that they can be used and introduced into a reactive planner for a virtual character (e.g. a virtual lumberjack).

To provide a more rigorous proof of the presented concepts, we tend to employ agents into a more complex environment (like UT2004) via the Pogamut platform. We intend to use the test-bed of scenarios that is being created at our laboratory, to let agents of various capabilities compete against each other in achieving various goals (e.g. speed, adaptability, believable behavior etc.).

Acknowledgments: This work and writing of this paper in particular supported by the project CZ.2.17/3.1.00/ 31162 financed by the European Social Fund, the Budget of the Czech Republic and the Municipality of Prague. The research was partly

supported by Program “Information Society”, project 1ET100300517, and by the research project MSM0021620838 of the Ministry of Education of the Czech Republic.

Bibliography

1. EA Games (Maxis), http://thesims2.ea.com/ (2004)
2. Brom, C., Gemrot, J., Burkert, O., Kadlec, R., Bída, M.: 3D Immersion in Virtual Agents Education, In Proceedings of First Joint International Conference on Interactive Digital Storytelling, Erfurt, Germany, Springer, Germany, p59-70 (2008)
3. Balicer, R. D.: Modeling Infectious Diseases Dissemination Through Online Role-Playing Games, Epidemiology Volume 18 - Issue 2, p260-261 (2007)
4. Mateas, M., Stern, A.: Facade: An Experiment in Building a Fully-Realized Interactive Drama, Game Developers Conference, Game Design track (2003)
5. Reidl, M. O., Stern, A.: Believable Agents and intelligent Scenario direction for Social and Cultural Leadership Training (2005)
6. Bohemia Interactive: Virtual Battlespace 2, http://virtualbattlespace.vbs2.com (2009)
7. project Agentfly, ČVUT, Fakulta Elektechnická, http://agents.felk.cvut.cz/projects/agentfly (2009)
8. Wooldridge, N.: An Introduction into MultiAgent Systems, John & Wiley & Sons (2002)
9. Pfeifer, R., Scheier C.: Understanding Intelligence, MIT Press (1999)
10. Brooks, R.A.: A robust layered control system for a mobile robot, In: IEEE Journal of Robotics and Automation, RA-1, April, p14-23 (1986)
11. Lorenz, K.: Foundations of Ethology, Springer-Verlag (1981)
12. Bryson J., Stein L.A.: Modularity and Design in Reactive Intelligence (2001)
13. Gemrot, J., Kadlec, R., Bida, M., Burkert, O., Pibil, R., Havlicek, J., Zemcak, L., Simlovic, J., Vansa, R., Stolba, M., Brom C.: Pogamut 3 Can Assist Developers in Building AI for Their Videogame Agents. In: Proc. Agents for Games and Simulations, AAMAS workshop 144-148 (2009)
14. Artificial Minds for Intelligent Systems – AMIS, http://artemis.ms.mff.cuni.cz/main/tiki-index.php (2009)
15. Kadlec, R.: Evolution of intelligent agent behaviour in computer games, Master Thesis, Faculty of Mathematics and Physics, Charles University, Prague (2008)
16. Bryson, J.: Intelligence by Design: Principles of Modularity and Coordination for Engineering Complex Adaptive Agents. PhD thesis, Massachusetts Institute of Technology (2001)
17. Brom, C.: Hierarchical reactive planning: Where is its limit?, Charles University, Prague (2005)
18. Mateas, M., Stern A.: A Behavior Language: Joint Action and Behavioral Idioms (2004)
19. Brom, C., Gemrot, J., Bída, M., Burkert, O., Partington, S.J., Bryson J.: POSH Tools for Game Agent Development by Students and Non-Programmers, In: CGAMES IEEE DUBLIN (2006)
20. Isla, D.: Handling Complexity in the Halo2 AI, In: Proceedings of the Game Developers Conference (2005)
21. Plch, T.: Action selection for an animat, Master Thesis, Faculty of Mathematics and Physics, Charles University, Prague (2009)