Proceedings of the Estonian Academy of Sciences,
2013, **62**, 1, 27–38
doi: 10.3176/proc.2013.1.04
Available online at www.eap.ee/proceedings

# Implementing artificial intelligence: a generic approach with software support

Teemu J. Heinimäki[a*] and Juha-Matti Vanhatupa[b]

[a] Department of Mathematics, Tampere University of Technology, P.O. Box 553, FI-33101 Tampere, Finland
[b] Department of Pervasive Computing, Tampere University of Technology, P.O. Box 553, FI-33101 Tampere, Finland; juha.vanhatupa@tut.fi

Received 18 August 2011, revised 25 November 2012, accepted 17 December 2012, available online 20 February 2013

**Abstract.** In computer games, one of the eminent trends is to create large virtual worlds with numerous non-player characters. Usually their artificial intelligence (AI) is implemented by scripting, which can be a burden for the application developers involved. This paper suggests an approach facilitating designing AI functionalities and striving to reduce, via software tool support, the amount of hand-written AI script code needed. Our approach is suitable for, e.g., creating autonomous agents with personal characteristics, capable of behaving in a natural manner. For instance, the goal-oriented agent paradigm can be applied easily with the approach. The definitions needed are written using a script language. Therefore, the agent configurations can be tested rapidly. We have extended an existing AI environment and created a script framework for implementing general-purpose AIs. Moreover, we have implemented software tools capable of generating script code for helping in the AI structure design and for simplifying the actual code-writing task. For demonstrating the applicability of our approach, three example scenarios specialized from the framework are presented.

**Key words:** artificial intelligence, autonomous agents, finite state machines, software tools.

## 1. INTRODUCTION

Over the recent years, virtual worlds of computer games have expanded beyond measure. They have become complicated constructs with numerous non-player characters (NPCs) and places. Having a huge number of NPCs makes it possible to offer lots of possible opponents, allies, sources of information, and trading partners. Thus, by increasing the number of NPCs, the game can often be made more interesting. However, this also means more time spent on implementing artificial intelligence (AI) for these NPCs. The NPC AI is a crucial aspect of computer games, but achieving sophisticated intelligent behaviour for a very large number of NPCs is not possible using conventional AI methods or engines [19]. The great challenge for the AI is to get the NPCs to behave in a believable, human-like fashion. The player should be able to imagine that interactions between real characters take place. The same facts apply also when considering AI agents (without graphical presentation) for, e.g., playing strategy or board games.

In this paper we propose a novel script-based[1] AI approach for computer game development. It is applicable with several kinds of games, and can be applied also outside the gaming domain. We use a separate AI framework that can be attached to the actual game engine or other program using the AI features. The purpose of the AI framework is to offer virtually all the AI support needed for creating interesting games. Thus, the application developers can concentrate on the real AI problems instead of writing lots of similar scripts

* Corresponding author, teemu.heinimaki@tut.fi

[1] Nowadays, game AIs are often implemented by means of scripting; scripts may be used, among other things, for communication dialogues, decision logics, stage direction, and fighting tactics [1,3]. The separation of the AI code and the main program simplifies the development process and makes it easier to modify the AI by different interested parties, like game designers or end user "modders". The matter of scripting AIs is discussed more, e.g., in [5].

for different agents. Scripting is used as a helping method, but most of the actual work is done by the framework. This paper is a derivative of the existing conference article [7], and has been extended with numerous additional considerations, clarifications, and an extra demo scenario. Also the software tool support has been extended and improved since the original paper.

Our approach is based on the famous idea of dividing and conquering: we aim at splitting the problem into smaller ones, and structuring the code accordingly. Moreover, our goal is to facilitate the work of AI developers by making it easier to design the AI structure naturally, in terms of the temporal flow of, e.g., the decision-making process. In addition, our approach is meant to make it easy to take advantage of personalization of the agents and goal-oriented action planning (GOAP) [12] principles. For these ends we have developed a general-purpose scripting framework and software tools for creating specialized AI implementations easily.

Our hypothesis is that using the approach, the productivity of AI programmers can be increased, as implementing AI features requires less work; the tools help in saving time and work in many ways. The ultimate goal of this work is to improve the quality of AIs in digital games by facilitating more efficient use of the AI programmer resources.

The framework was obtained by extending, modifying, and generalizing the AI support part of our existing CAGE game engine [18] and by making it totally independent from CAGE. At the beginning of this work, CAGE supported directly only NPC AIs implemented via state machines, and the application developer had to create more sophisticated AI implementations manually, if needed. Since, CAGE AI framework was extended with GOAP support, before the generalization and separation into our current framework.

The approach of this paper combines the use of finite state machines with the use of new kind of machines for implementing general-purpose AIs. With these machines, e.g., goal-oriented agents can be implemented easily. For testing the applicability of the approach, we have programmed three example scenarios using the framework. For taking advantage of our framework the software tools created for specializing the framework are essential.

The rest of the paper is organized as follows. In Section 2 some background and related work are covered. After that, in Section 3, we present our general AI approach in more detail. In Section 4 we explain the prototype AI framework – a realization of the general idea. Then, in Section 5, the implemented software tools and the benefits offered by them are discussed. In Section 6 we present example scenarios featuring AI implementations created by specializing the framework. The concluding remarks are given in Section 7.

## 2. BACKGROUND AND RELATED WORK

Almost all virtual worlds of computer games contain less intelligent NPCs in the form of animals and humans with simple behaviour; there are servants, guards, shopkeepers, and so on. State machines are a suitable method to model their behaviour. According to [12], the most of the decision-making systems in current games are implemented using state machines along with scripting. A problem with state machines is their rigidity: when encountering situations that have not been foreseen, the resulting behaviour can be poor [6].

Game worlds may contain also more intelligent agents, whose behaviours should be as complex as those of the player characters (PCs). Alas, in practice, often AI implementations – and thus, the resulting behaviours – of these agents are too simple. It is quite common to just stand still waiting for the interaction initiated by the PC, or to live only to die by the sword of the PC. This kind of simplified behaviour of NPCs can lead to boring, self-repeating, and unnatural virtual worlds. Therefore, our approach aims at facilitating creating intelligent agents of high quality.

One possibility for a basis of creating a sophisticated AI is to model personal characteristics, moods, and knowledge of the agent for inducing its behaviour. Often the AI can be improved by adding in-game interactions between NPCs. (The significance of them is discussed, e.g., in [17].) Also other methods for making games more interesting, developed over several decades by the multi-agent system community, can be used. However, the basic problem remains: often simply too much work is required to implement the wanted behaviour properly with conventional methods. Our approach strives to tackle this problem.

The usual approach for implementing new computer games is to use game engines; modern games are seldom developed from scratch [12]. Instead, a suitable game engine is modified for the new game, which is then implemented using the engine as a framework. Often some scripting language is used in addition to a conventional one. The differences between game engines are huge; their capabilities and features, including AI support features, differ a lot. It is common that the AI implementation of a game is not really supported by the game engine used, but the AI is somehow glued into the main game. On the other hand, many commercially used game engines offer some sophisticated AI support features. However, to the best of our knowledge, none of them use such a personal trait-based[2] approach as presented in this paper.

There is a wide array of publications on solving different kinds of AI problems in game worlds by NPCs and other agents. However, most of them seem to be solving quite specific problems in specific environments. Our framework, on the contrary, aims to be very generic.

[2] Our approach can be successfully used also without personality features, but the original CAGE AI framework extension was designed especially for using them.

Despite the genericity, it is as a rule easily applicable in a given task. The exact easiness depends on the case, but the lack of artificial limitations in the framework and tool support for specializing it help a lot.

The "Scripted Artificial Intelligent Basic Online Tank Simulator" (SAI-BOTS) [2] lets the players script the behaviours of their tanks using the Lua scripting language and then fight each other with them. It resembles, for instance, the CAGE system in the sense that its main programming language is C++ and it uses Lua for scripting. When using our approach, the application developer writes definitions for agents, which calculate their actions autonomously. In SAI-BOTS, on the other hand, the behaviour of the tanks is pre-scripted conventionally. Moreover, CAGE is a prototype game engine, which can be used to develop computer games of different genres. SAI-BOTS, instead, is a specific, scriptable computer game.

The AI structure of "being-in-the-world" [4], a multi-user dungeon (MUD) agent, consists of two asynchronously working modules. The reasoning module includes, e.g., logical reasoning and goal maintenance, while the real-time coping module deals with the world. The basic structure of the solution could be implemented using our approach, while it might not be the best possible way to perform ontological reasoning. The solution of being-in-the-world resembles that of the CAGE engine for goal-oriented agents. While our AI framework uses Lua, being-in-the-world has been written in Common Lisp. The framework presented in this paper is general enough to be suitable for different kinds of games and other purposes, but being-in-the-world is a MUD agent created for surviving in a certain kind of MUD.

Basketball Artificial Intelligence Tool (BAiT) [16] is a data-driven software tool system for implementing AI for a basketball game. It aims at offloading work from programmers by transferring some of it to designers. Our approach strives to offer help for generic AI implementation – not only for implementing a finite set of AI features for a single game (or genre). This, on the other hand, means that we cannot offer as high a level of abstraction as BAiT does; we expect the users of our tools to know as much of the target system and its interfaces as if they were to script the AI conventionally. (Of course, application-specific simplified scripting interfaces can be offered to be used via our tools, if necessary.) BAiT takes benefit of the fact that basketball as an activity is sequential by nature. This is the case in most sports. Also our approach is extremely suitable for creating AIs for such sequential activities.

## 3. ARTIFICIAL INTELLIGENCE APPROACH

For ensuring suitability for different needs, our approach includes both state machines and more advanced methods intended originally to be used with autonomous agents with personalities. We divide artificial agents into two categories according to the methods used: there are state machine agents (SMAs) and advanced AI agents (AAIAs). Instead of state machines, AAIAs use a new kind of AI machines (AIMs) for behaviour generation. The focus of this paper is on AAIAs, but the support for SMAs is included in our approach, as sometimes trying to mimic complex human-like "thinking" processes for inducing actions can be an overkill. This matter is discussed, e.g., in [10].

In modelling and developing agents, the goal orientation as a paradigm is increasingly recognized [15]. AAIAs support fully creating goal-oriented agents (GOAs) that model and use goals and motivations in their action-generating processes. We consider them as a subcategory of AAIAs. The agent types of our AI approach are depicted in Figs 1 and 2.

In our approach, the application developer describes – depending on the type of the agent – either the agent behaviour as a finite state machine or AIMs for generating behaviour autonomously. (This is the original idea of using AIMs, but they can be applied freely and offer benefits also in other kinds of settings.) The descriptions are given using a script language[3]. Scripting allows rapid application development and testing. Downside of this is the lack of efficient debugging capabilities. However, it is possible to implement the needed machines piece by piece and to test each added code block separately, without any need to recompile the framework during the process.

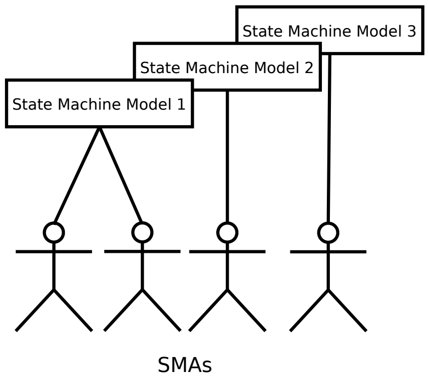


**Fig. 1.** State machine agents. Several agents may share the same state machine model.

[3] Using script languages or writing scripts are not bad things per se; the important thing is how and when to script. When scripting AI traditionally, the application developer writes the whole AI implementation with the given language somehow. Various methods can be used, including using state machines. This means that we recognize the usefulness of some traditional scripting methods in some cases. However, at least as far as the AAIAs are considered, instead of giving the application developers a language and free hands to use it, we want to reduce their workload by defining a clear structure and making automatic code generation possible without taking away their freedom to apply their creativeness.

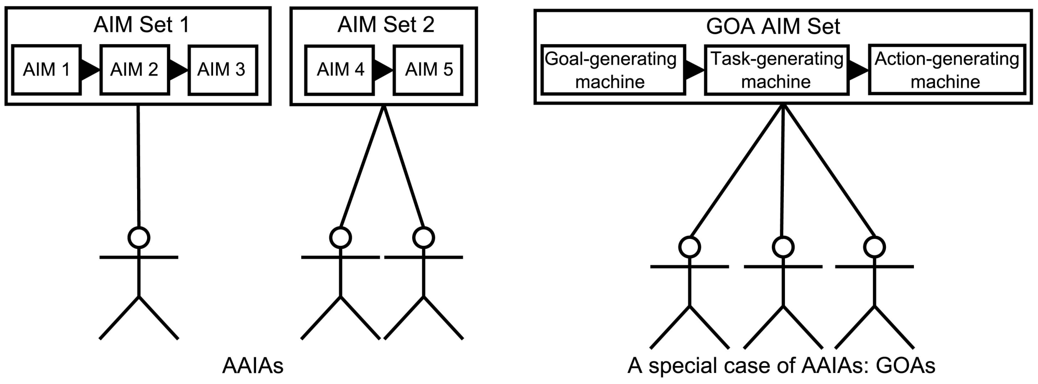


**Fig. 2.** Advanced AI agents. Goal-oriented agents can be seen as a special case.

Our approach enables the creation of huge virtual worlds with lots of different inhabitants with diverse AI capabilities. Of course, when the number of agents grows enough, making any kind of individual AI calculations for all of them in real time will become impossible. While our approach cannot solve this fundamental problem, it facilitates and speeds up, nevertheless, the creation of a large number of believable AI implementations. The AAIAs are not only suitable for advanced NPCs, but also for believable and seemingly intelligent co-players or adversaries. AAIAs can even be used in implementing different decision-making systems, planners, and so on. Similarly, SMAs can be used for various purposes.

## 3.1. State machine agents

Although the main focus of this paper is in AAIAs, we have included the support for conventional state machine-based agents as well for convenience. Simple behaviour, after all, is easily achieved using them. Another good side (in some cases) in them is the high predictability of the actions[4]. Moreover, finite state machines are not generally computationally expensive [11].

State machines could easily be implemented using AIMs. However, we choose to use a specific structure for implementing them. They are, after all, rather simple constructs that do not really benefit from being implemented as AIMs.

The application developer describes the state machine models by scripting. This work includes defining the states, state transitions, and inputs capable of firing the state transitions. It is possible to define a separate state machine model for each agent, but several agents having similar behaviour can also share one. This reduces the needed amount of copying and pasting code. An example state machine model is depicted in Fig. 3.

In [7], the general idea was that each agent using state machines has its own state machine instances and

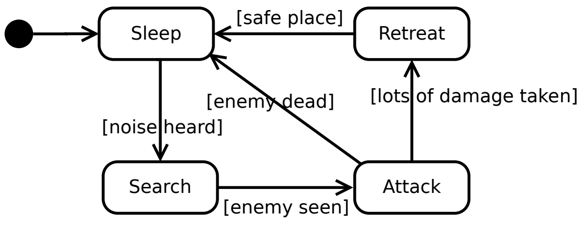


**Fig. 3.** An example state machine model.

can be in a different state than other agents regardless of whether they use the same state machine model or not. Connecting only one agent to a state machine instance was suggested for simplicity and general implementability. However, currently we are focusing on using shared state machines. This requires keeping track of states of individual agents, either by themselves or by the state machine construct, but in many cases considerable space savings can be achieved by not replicating similar machines.

## 3.2. Advanced AI agents

In our approach, every AAIA uses a number of AIMs that may modify its personal parameters, possibly adding also new ones. The application developer defines these attributes for the agent, as well as the set of AIMs to calculate new agent parameters, for example, goals and means to acquire them, and to modify the existing ones. The AIM set establishes the desired AI architecture.

The composition of a single AIM is depicted in Fig. 4. Each machine can contain several layers of calculation nodes (CNs). A CN may perform comparisons, calculations, and basically any program code. However, the idea is to keep the logics of an individual CN as simple as possible. The layers are iterated through in a fixed order according to their numbering (in Fig. 4, from Layer 1 to Layer *n*). Inside of a layer, the execution

[4] Of course, there are numerous state machine-based approaches that use probabilities and randomness, but they are out of the scope of this paper.

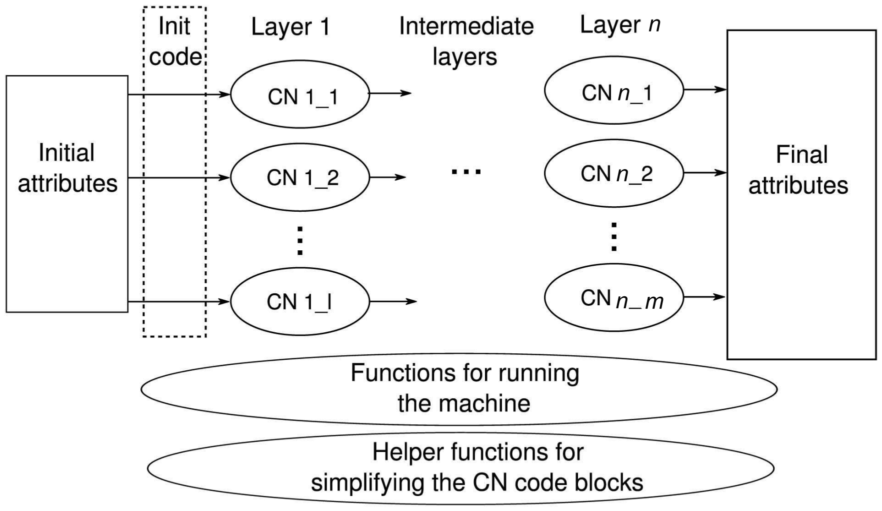


**Fig. 4.** The components of an AIM.

order of CNs is not fixed[5]. (By executing a CN we mean running its respective code block.) Because of this, the code of a CN must not depend on attributes altered on the same layer. It may only trust that the operations of the previous layers have been performed. Possible common initialization or input filtering code can be run before executing the CN layers. Feedforward artificial neural networks can be seen as special cases of AIMs.

The purpose of suggesting the use of several machines instead one is to simplify the overall AI construction by splitting it into logical pieces. The whole AI functionality is then achieved by chaining the AIMs, as shown in Fig. 5. The arrows pointing downwards in the figure represent the flow of execution. Passing the information from an AIM to another is accomplished by simply modifying the attribute data to which the machines have a shared access. Thus, defining any kind of complex input/output interfaces is not necessary.

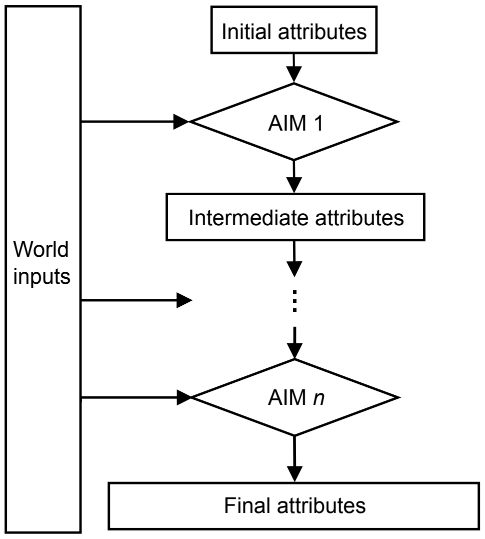


**Fig. 5.** Overview of the approach. A chain of machines is used to produce new data, like goals and actions to perform. The different rectangle widths depict possible growth of the number of the parameters.

Naturally, the usage of AIMs is not limited into this kind of single cascade structures. They are often viable and sufficient, but if needed, several AIM cascades can be run in parallel, or hierarchical AIM structures can be built.

The initial attributes can contain many kinds of data, including the personal traits and the moods of an agent. These parameters are simple key–value pairs. Data requiring more complex representation, like knowledge, other history-related data, and perceptions, can also be present, represented by suitable data structures. In every stage in the process, the existing attributes can be modified and new ones can be added, so the cardinality of the set of the final attributes may be greater than that of the set of the initial attributes.

The AIMs can be easily used to decide goals, generate action possibilities, and eventually produce a priority list of actions (or sequences of them) to carry out. Thus AIMs are a suitable tool for applying GOAP techniques. In Fig. 6, an example machine architecture, the basic GOAP system of the CAGE framework, is depicted. This architecture implements the general idea of Fig. 5 and uses AIMs at three levels. First, there is a machine for generating the goals based on the personality and the environment. The next machine generates tasks to be performed based on the most important goals given by the first machine. Finally, there is a separate machine for splitting the tasks into short-term actions. Effectively all the machines manipulate the importance values of their respective output parameters so that decisions can be made based on priority lists. For keeping the goals

[5] The purpose of not fixing the order is to make it possible to write parallel implementations easily.

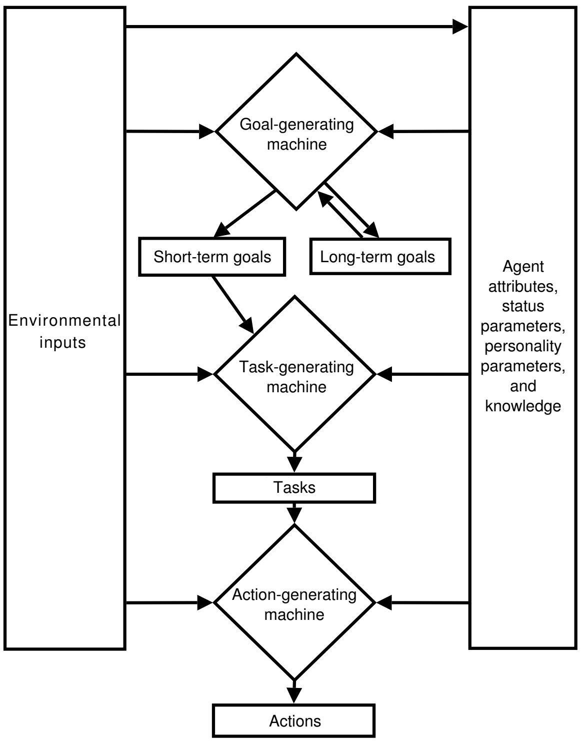


**Fig. 6.** Architecture of the CAGE GOAP system.

and tasks updated, the machine chain for an agent is run periodically, always when completing a task, and always when important environmental events occur.

## 4. STRUCTURE OF THE $F_1I_2A_3$ FRAMEWORK

For implementing and evaluating our approach, we defined the structure of the actual code so that the development process could partly be easily automated. The resulting high-level AI framework is called $F_1I_2A_3$. The name comes from "Framework for Individually Intelligent Autonomous Artificial Agents". The framework provides support for SMAs and AAIAs. It is also possible for an AAIA to command SMAs and other AAIAs belonging to its group using a simple message-passing mechanism.

The $F_1I_2A_3$ framework was obtained by extending and generalizing the existing AI framework of the CAGE game engine, so the language of choice is the scripting language Lua [9]. It is a dynamically typed popular language, which has been used in several games and industrial applications [8]. Lua is also very fast [3]. Moreover, Lua is an easy language to bind with other languages. This was an important factor when originally deciding the scripting language to be used in the CAGE AI framework. Thus, $F_1I_2A_3$-based AI implementations can be easily attached to different game engines or other "main programs".

### 4.1. Implementing state machines

The state machine implementation principles of $F_1I_2A_3$ are basically those of the original CAGE AI framework (although nowadays we use shared state machines actively). For using CAGE state machines, the application developer defines the state machine models consisting of state models, and connects SMAs to them. The state models include the state transitions on suitable conditions, and are written in Lua.

An example $F_1I_2A_3$ implementation of a state model is shown in Fig. 7. The modelled state is an attack state of a simple agent, which could be used, for example, in a first person shooter game. When the agent is in the attack state and sees a PC, it attacks. When there are no more enemies in the vicinity, the agent falls asleep. If injured severely, the agent will retreat.

In Fig. 7, the basic code-level structure of state models can be easily seen: they are implemented as Lua tables with three functions. One is called when entering the modelled state, one when executing corresponding state actions and triggering possible transitions, and one when leaving the state. Besides individual state models, state machine model implementations include functions for creating agent tables, setting agent properties, and running the modelled state machines for different agents.

### 4.2. Implementing advanced AI machines

Besides CNs, an AIM includes some metadata and functions for running and cascading the machines. The most important function is $run\_X(agent)$, in which $X$ is replaced by the machine name (that serves as a unique identifier (UID)). This function is used for running the code segments of the CNs in the order determined by the machine structure. Using the *agent* parameter, some suitable information about the agent for which the machine is meant to be run is passed for it. The parameter may, for instance, contain only a string containing the name or the UID of the agent, or it can be a pointer to a complex agent class with methods usable from the Lua code. There are also helper functions (achievable from the CN code blocks) for, e.g., sorting and handling priority lists.

An example implementation of a CN of the CAGE Goal Generating Machine is shown in Fig. 8. The CN is responsible for updating the hunger status parameters of the NPCs. Updating takes place each time the game time passes.

```
--Attack state of an agent
State_Attack = {}
State_Attack ["Enter"] = function(agent)
  agent.draw_weapon()
end
State_Attack ["Execute"] = function(agent)
  print("[Lua]: Executing the attack state")
  if agent.hp < 0.2 * agent.max_hp then --lots of damage taken:
    change_state(agent, "State_Retreat")
    return
  end
  if safe(agent) then --enemy probably dead
    change_state(agent, "State_Sleep")
    return
  end
  agent.attack()
end
State_Attack ["Exit"] = function(agent)
  agent.holster()
end
```

**Fig. 7.** An attack state model implementation. In this case, *agent* is a Lua table that stores necessary properties and functions of an AI agent using the state machine model.

```
-- gc_attributes is a table containing
-- attributes of NPCs, which
-- is indexed with an agent name

function calculation_node_1_2()
  an=agent:get_name()
  gc_attributes[an].hunger =
    gc_attributes[an].hunger +
    gc_attributes[an].hunger_step *
    gc_attributes[an].time_tick;
end
```

**Fig. 8.** An example calculation node implementation. The variable *agent* that has been given to the machine-running function as a parameter can be used inside the CNs. In this case, *agent* refers to a class instance with a method called *get_name*() callable from the Lua code.

## 5. TOOL SUPPORT

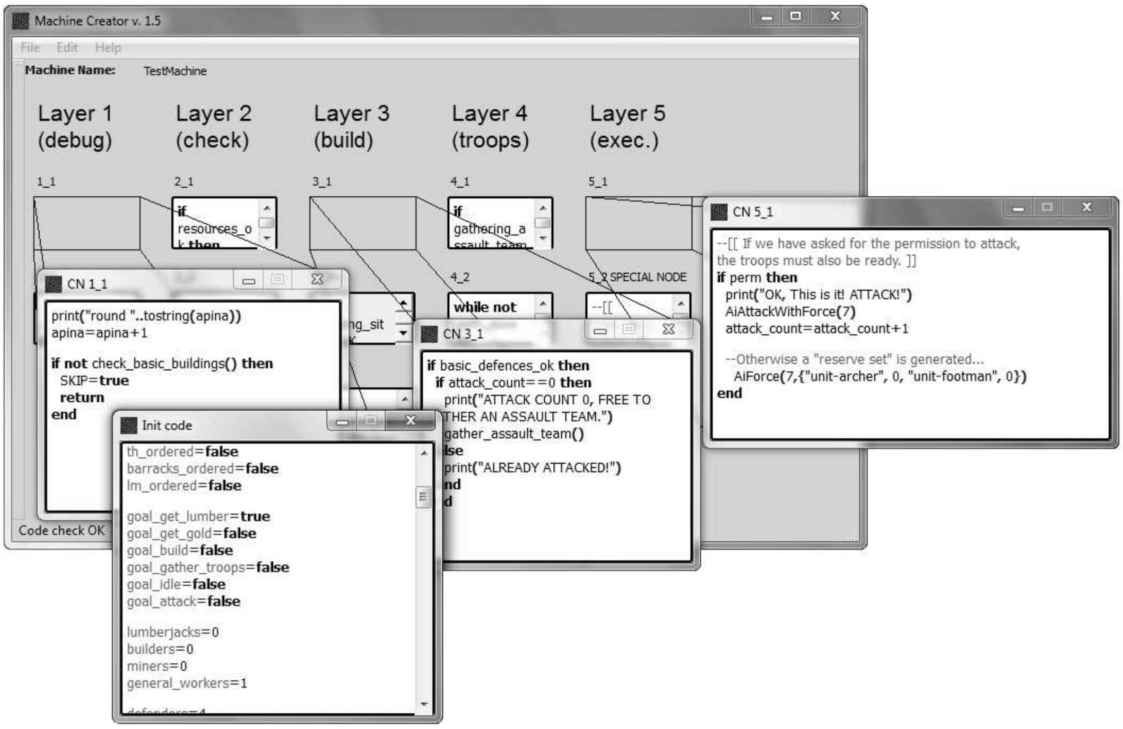
In the case of AAIAs, specializing the $F_1I_2A_3$ framework requires fixing the number and the layer structure of CNs and writing their respective code blocks. It is easy to automate the generation of the other parts of a normal specialized AI implementation. So, to help application developers in the creation of AIMs, we have implemented a software tool, called Machine Creator, for building them. Defining the layer structure of a machine and inserting the required script code for the CNs can easily be done via the graphical user interface (GUI) offered by it. A screenhot of the Machine Creator GUI is shown in Fig. 9.

After the structure of the machine under construction has been defined and the desired content of the CNs has been added, one can order the software to generate the AIM in $F_1I_2A_3$ format. In the Machine Creator version used in [7], the agent parameter value initializations had to be added to the script files by hand, but nowadays, an arbitrary initialization code can be added via the GUI. Everything else is generated automatically. The tool support speeds up the machine creating process considerably.

The Machine Creator tool has been implemented using the Qt framework [14] in order to get an easily portable tool for different platforms. It has been successfully used in Linux and Microsoft Windows environments.

Besides adding the possibility of giving the initialization code via the GUI, we have improved the Machine Creator tool in many other ways since [7] was published. The GUI has been enhanced visually, and alternative ways of performing actions have been added. Nowadays, the Machine Creator can also load the generated scripts from files for further modifications. Moreover, a syntax checking feature has been added: the tool can check the syntactical correctness of each code block. This makes debugging – and in the first place, generating bugless machines – considerably easier. Also Lua syntax highlighting support was added for making code-writing more pleasant and less error-prone.

Our AAIA approach would only have little use without the Machine Creator: GUI, syntax checking, and realization of our framework by automated code generation are the means for trying to achieve our goal of reducing the burden of AI developers. The time and



```
print("round "..tostring(apina))
apina=apina+1

if not check_basic_buildings() then
  SKIP=true
  return
end
```

```
if basic_defences_ok then
  if attack_count==0 then
    print("ATTACK COUNT 0, FREE TO
THER AN ASSAULT TEAM.")
gather_assault_team()
lse
print("ALREADY ATTACKED!")
nd
d
```

```
--[[ If we have asked for the permission to attack,
the troops must also be ready. ]]
if perm then
  print("OK, This is it! ATTACK!")
  AiAttackWithForce(7)
  attack_count=attack_count+1

  --Otherwise a "reserve set" is generated...
   AiForce(7,{"unit-archer", 0, "unit-footman", 0})
end
```

```
th_ordered=false
barracks_ordered=false
lm_ordered=false

goal_get_lumber=true
goal_get_gold=false
goal_build=false
goal_gather_troops=false
goal_idle=false
goal_attack=false

lumberjacks=0
builders=0
miners=0
general_workers=1
```

**Fig. 9.** The Machine Creator user interface.

work savings come from multiple sources; the main benefits of using the tool (and our general approach) are
- the possibility of implementing scripts naturally, based on temporal (decision-making etc.) flows;
- the easiness of locating the relevant pieces of code, when necessary;
- the easiness of implementing agent personalization and GOAP features;
- the savings obtained due to automation of implementing logics related to the execution order;
- helper functions offered;
- syntax checking in the code block level; and
- easiness of writing code to be executed truly concurrently[6].

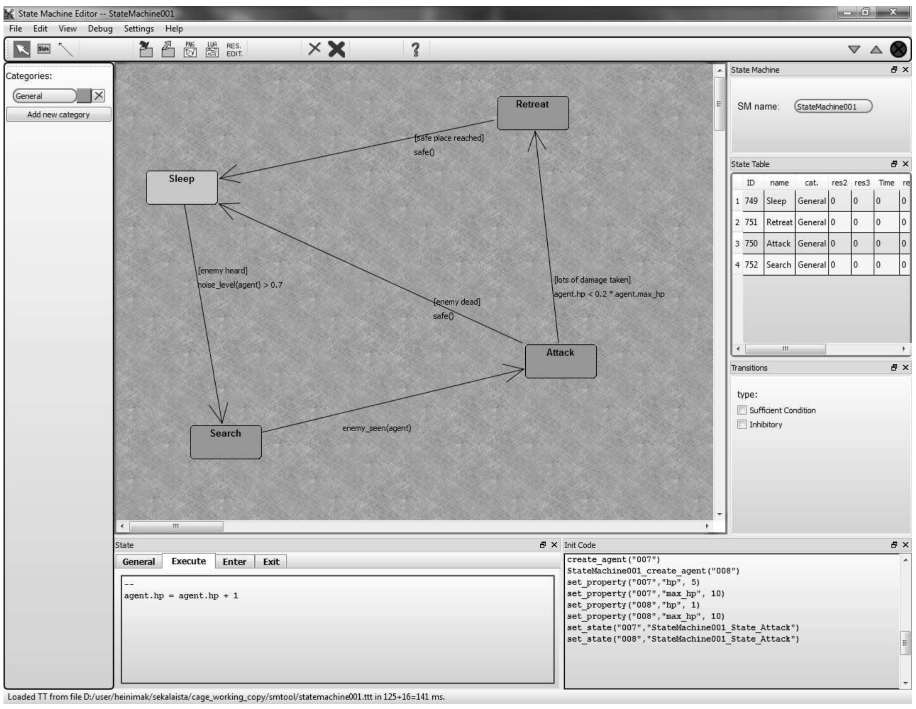
Of course, the actual savings in the amount of work or time used when implementing an AI are highly case-dependent. In some cases, our approach might not offer any benefit, while in other cases, the savings could be considerable. We do not try to make any numerical comparison against conventional scripting, as it is impossible to choose a generally applicable and fair baseline; if some AI feature was fixed to be implemented first conventionally and then with our tool, the results would not mean much, as there are many ways to solve a problem in either way. However, based on our experiments (see Section 6), we claim that at least in some cases using our AAIA approach and Machine Creator is truly beneficial.

The size of the script code overhead generated for structuring the AIMs and using them grows a (small) constant amount for each CN and a constant amount for each AIM. Compared to the size of the hand-written code, the size of the automatically generated code may be considerable with simple AI implementations with several AIMs and CNs and negligible with complex implementations with only a few AIMs and CNs. Ideally, all the code written by the AI developer should be “effective” and closely AI-related, but the exact number of the lines of the code needed for some implementation depends on the skills of the programmer and the chosen way to use our tool.

We also have implemented another tool, called State Machine Editor, for creating and modifying finite state machines via a GUI and for generating Lua implementations automatically. The GUI is demonstrated in Fig. 10. With this tool, our goals are to
- provide a visual view to the logics and thus help in the design process and
- speed up the development: the time savings gained by generating the state machine code automatically are substantial.

The Qt framework has been used also for implementing the State Machine Editor for the same reasons as it was used in the Machine Creator implementation. Also the State Machine Editor runs at least in Linux and Microsoft Windows environments.

[6] It is trivial to split the generated code to pieces to be run concurrently, and the GUI of the Machine Creator helps with visualization of the code parallelization.



| | ID | name | cat. | res2 | res3 | Time | re |
|---|---|---|---|---|---|---|---|
| 1 | 749 | Sleep | General | 0 | 0 | 0 | 0 |
| 2 | 751 | Retreat | General | 0 | 0 | 0 | 0 |
| 3 | 750 | Attack | General | 0 | 0 | 0 | 0 |
| 4 | 752 | Search | General | 0 | 0 | 0 | 0 |

```
--
agent.hp = agent.hp + 1
```

```
create_agent("007")
StateMachine001_create_agent("008")
set_property("007","hp", 5)
set_property("007","max_hp", 10)
set_property("008","hp", 1)
set_property("008","max_hp", 10)
set_state("007","StateMachine001_State_Attack")
set_state("008","StateMachine001_State_Attack")
```

Loaded TT from file D:/user/heinimak/sekalaista/cage_working_copy/smtool/statemachine001.ttt in 125+16=141 ms.

**Fig. 10.** The State Machine Editor GUI.

## 6. EXAMPLE SCENARIOS

We have implemented three example scenarios by specializing the $F_1I_2A_3$ framework to test our AAIA approach. The first one is a scenario called Gunslinger, in which the PC plays a role of a sheriff in the situation in which a group of bandits has stolen a chest full of gold. The purpose of this scenario was to verify the applicability of $F_1I_2A_3$ for its original intended task, creating NPC AI implementations. (The problem domain analysis considering the role of a sheriff is omitted here for brevity because it is not crucial given our goal.)

The second scenario is called Gomoku. It was implemented to demonstrate that it is possible to create also a board game opponent AI using the framework. The third scenario, Wargus AI, was implemented to test the applicability of $F_1I_2A_3$ for creating hierarchical manager systems. We created a simple strategy manager AI for playing a real-time strategy (RTS) game.

### 6.1. Gunslinger

In the Gunslinger scenario, initially there are five bandits guarding a chest full of gold. All the bandits use the same AIM, but differ in personal traits. Additionally, one of the bandits is the leader of the group. The leader can command the others, and the commandments are normally obeyed.

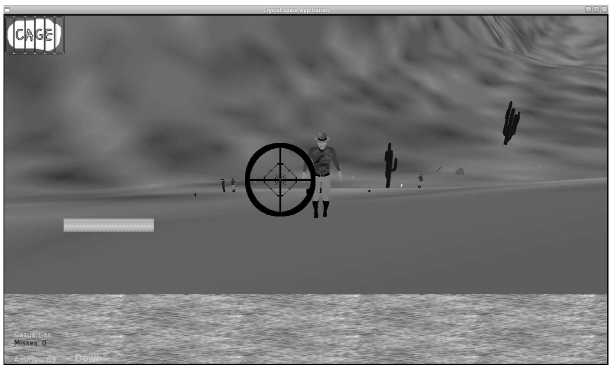
State machines could have been used for this basic operation of the bandits. However, the bandits are meant to model human beings, and human beings have different personal urges. For modelling these, AIMs are handy. So, if some personal need of a bandit to do something, for instance to flee, outweighs the authority of the leader, the bandit can also commit “rebellious acts”.

The personalities of the bandits are modelled by the following parameters: authority, bravery, alertness, tiredness, and greed. The tiredness value grows as time passes. The observations about the environment affect also: the PC can be seen, as can be the gold chest. Moreover, the bandits are aware of the casualties. In this simple scenario, there are five possible actions the bandits can perform: guarding, sleeping, attacking, alarming, and fleeing. In addition, the leader can command the others to perform these tasks.

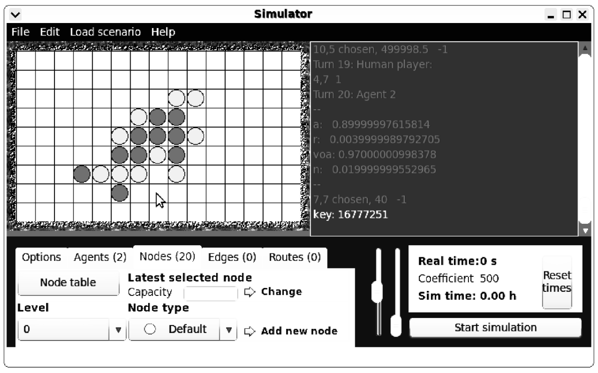
As the game engine (providing the graphics, the game loop, etc.) we used CAGE run in Ubuntu 10.04. Attaching the AIM to it was trivial and required only minor modifications as CAGE already was capable of running Lua scripts. A screenshot, in which the PC is attacking a bandit, is shown in Fig. 11a.

By using the Machine Creator tool, the implementation of the scenario, in which the individual characteristics of the bandits are clearly reflected into their behaviours, required roughly 40 hand-written lines of Lua code for the CN logics. There were a total of eight CNs forming an AIM used by all of the bandits. The scenario implementation clearly demonstrated that the Machine Creator tool and our approach in general could be successfully used for speeding up NPC AI implementations.

(a)



(b)



```
10,5 chosen, 499998.5   -1
Turn 19: Human player:
4,7  1
Turn 20: Agent 2
--
a:   0.89999997615814
r:   0.0039999989792705
voa: 0.97000000998378
n:   0.019999999552965
--
7,7 chosen, 40   -1
key: 16777251
```

Latest selected node
Capacity
Level: 0
Node type: Default
**Real time:0 s**
Coefficient 500
**Sim time: 0.00 h**

**Fig. 11.** Screenshots from example scenarios: (a) the Gunslinger scenario, (b) the Gomoku scenario.

## 6.2. Gomoku

The $F_1I_2A_3$ framework is meant to be a general-purpose AI framework. Thus, it should be applicable not only in creating the AIs for NPCs, but also in creating AIs that could be used as opponents or co-players. For demonstrating this, an AI for playing Gomoku (also known as Five-in-a-row) was implemented. This scenario was created for a self-made, general-purpose simulator software run in Ubuntu 10.04. A screenshot from a game between an AAIA and a human being is shown in Fig. 11b.

The basic implementation is simple: all the possible places for setting a counter are evaluated each round and the one with the best evaluation score is chosen. The scoring is based on the overall numbers of counters in the horizontal, vertical, and diagonal rows around the potential place and the lengths of continuous rows around it. Immediate victory moves and the rows of three counters with open ends are recognized as the special cases.

For making the AI behave like a human, the evaluation is affected by three parameters: *ac*, *r*, and *asvm*. Their values are evaluated for each ply of the game, using a single AIM, as depicted in Fig. 12. The "attacking coefficient" *ac* tells how strongly the decisions are biased towards considering only the agent's own counters. If the value of *ac* is low, the agent will play defensively considering mostly the opponent's possibilities of winning the game. Randomness coefficient *r* is used for making random mistakes in the counter placement, and the parameter *asvm* represents the ability to spot places leading to victory.

The base parameters for different player agents are absent-mindedness, interest in playing the game, tendency to get nervous during the game, aggression, and general pedantry. By altering these initial values, the playing style of an agent changes to reflect the given characteristics.

The AIM uses only four CNs, each having only one Lua line of code. In addition, some code was needed for binding the parameter values to the actual game-playing code. Still, the amount of the code and time for creating an opponent AI was minimal, and the resulting AI seems to work as expected.

Based on this scenario, the AI approach presented in this paper seems to be suitable also for non-NPC AI implementations. In this scenario, as well as in the Gunslinger scenario, the benefit gained by using the Machine Creator tool was obvious. Without using it, the corresponding AI features would have required much more time to be implemented.

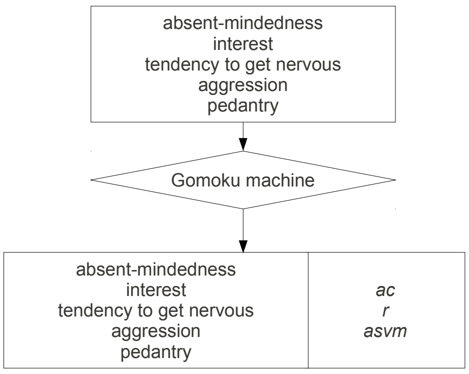


**Fig. 12.** Generating the playing parameters from the player agent characteristics using an AIM.

## 6.3. Wargus AI

RTS games are often seen as ideal test-beds for AI development. They offer a wide variety of challenges to be coped with. Wargus is a clone/modification of the well-known RTS game Warcraft II: Tides of Darkness (Blizzard Entertainment 1995). Instead of the original engine, Wargus runs on an open-source engine called Stratagus. The Stratagus engine has been previously used in different AI studies [13]. Therefore it seemed also to be a relevant test environment for our framework and we chose to implement a test scenario for it.

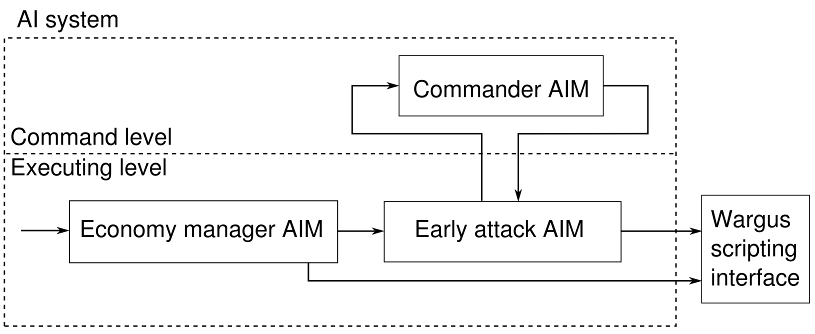


**Fig. 13.** The structure of the AI system implemented for Wargus using AIMs.

We tested cascading and hierarchical use of AIMs by creating an AI implementation for Wargus 2.2.6. It tries to beat its opponent by gathering an attack force and attacking the enemy relatively early in the game without giving the enemy much time to prepare; i.e. it does not rely on supremacy gained by developing technologies, but on speed. The structure of the AI implementation is depicted in Fig. 13.

The AI is two-tiered: there are command and executing levels. The executing level consists of two AIMS in a cascade. The first of them manages the economy, i.e. recruits peasants for workforce and gives orders about how to balance the efforts between gathering different resources, like lumber or gold. The second one issues orders related to constructing buildings, gathering army, and organizing attacks. This machine, however, is not allowed to work totally autonomously, but it must ask permissions to proceed. The command level consists of a single AIM that may give or deny these permissions.

The three AIMs needed were constructed using the Machine Creator. A total of 13 CNs were defined. The normal approach for implementing an AI for Wargus would have been writing a lengthy Lua script giving different orders concerning different things. We, instead, used such a normal AI script only for running our machines (which were implemented, of course, also as Lua scripts). Although the AI implemented was rather simple, it still took hundreds of lines of code. Writing and organizing it would have been more difficult without using the Machine Creator tool, which let us organize the code block execution with visual feedback, check the syntactical validity of each block separately, and split the total AI implementation into several simple machines without practically any additional work: while working already in a Lua scripting environment, the tool was able to generate all the code needed to run the machines automatically. Due to sharing the attribute data between the AIMs of an AAIA – in this case, our Wargus AI system – it is extremely easy to implement systems requiring message-passing between different components. This was also verified in implementing the Wargus scenario. A screen capture taken along the way of developing the Wargus AI using the Machine Creator is shown in Fig. 9.

## 7. CONCLUSIONS

In this paper we presented a general AI approach suitable for computer games and different kinds of simulations using intelligent, autonomous agents. The approach can ease the development of computer games containing large virtual worlds considerably. A crucial part of the contribution of this paper was introducing a new kind of artificial intelligence machines.

The approach was organized into a framework. Using the support offered by it, application developers can define their agents and create living virtual worlds with relatively little effort. The applicability of the framework was tested by creating tools for specializing it easily, and finally by implementing example scenarios. The preliminary results are promising: we were able to create believable and naturally behaving AIs quickly and easily.

### 7.1. Future work

The future work includes creating more different test scenarios and honing the framework structure and the Machine Creator tool based on the observations. We intend also to conduct tests with the State Machine Editor and demonstrate its usefulness. (This issue was left out of this paper, as the focus was on AAIAs.)

## ACKNOWLEDGEMENTS

This work was partly funded by the Academy of Finland, partly by the Graduate School on Software Systems and Engineering (SoSE), and partly by the Tampere Doctoral Programme in Information Science and Engineering (TISE). Also the Foundation of Nokia Corporation has supported the work.

## REFERENCES

1. Bourg, D. M. and Seemann, G. *AI for Game Developers*. O'Reilly Media, Inc., 2004.

2. Brandstetter, W. E., Dye, M. P., Phillips, J. D., Porterfield, J. C., Harris, F. C., Jr., and Westphal, B. T. SAI-BOTS: scripted artificial intelligent basic online tank simulator. In *Proceedings of the 2005 International Conference on Software Engineering Research and Practice (SERP '05)*. 2005, 793–799.
3. Buckland, M. *Programming Game AI by Example*. Wordware Publishing, Inc., 2005.
4. DePristo, M. A. and Zubek, R. being-in-the-world. In *Proceedings of the 2001 AAAI Spring Symposium on Artificial Intelligence and Interactive Entertainment*. 2001, 31–34.
5. Doulin, A. Scripting your way to advanced AI. In *AI Game Programming Wisdom*. Vol. 4. Charles River Media, 2008, 579–591.
6. Fairclough, C., Fagan, M., Mac Namee, B., and Cunningham, P. Research directions for ai in computer games. In *Proceedings of the Twelfth Irish Conference on Artificial Intelligence and Cognitive Science*. 2001, 333–344.
7. Heinimäki, T. J. and Vanhatupa, J.-M. Layered artificial intelligence framework for autonomous agents. In *Proceedings of the 12th Symposium on Programming Languages and Software Tools (SPLST'11)*. 2011, 102–113.
8. Ierusalimschy, R., Celes, W., and de Figueiredo, L. H. About. http://www.lua.org/about.html (accessed 20.09.2011).
9. Ierusalimschy, R., Celes, W., and de Figueiredo, L. H. The programming language Lua. http://www.lua.org/ (accessed 20.09.2011).
10. Khoo, A., Dunham, G., Trienens, N., and Sood, S. Efficient, realistic NPC control systems using behavior-based techniques. In *Proceedings of the AAAI 2002 Spring Symposium Series: Artificial Intelligence and Interactive Entertainment*. 2002, 46–51.
11. Lecky-Thompson, G. W. *AI and Artificial Life in Video Games*. Course Technology (CENCAGE Learning), 2008.
12. Millington, I. *Artificial Intelligence for Games.* The Morgan Kaufmann Series in Interactive 3D Technology. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2006.
13. Ponsen, M. J. V., Lee-Urban, S., Muñoz-Avila, H., Aha, D. W., and Molineaux, M. Stratagus: an open-source game engine for research in real-time strategy games. In *Papers from the IJCAI 2005 Workshop on Reasoning, Representation, and Learning in Computer Games*. 2005, 78–83.
14. Qt – cross-platform application and UI framework. http://qt.nokia.com/ (accessed 20.09.2011).
15. Shen, Z., Miao, C., Tao, X., and Gay, R. Goal oriented modeling for intelligent software agents. In *Proceedings of IEEE/WIC/ACM International Conference on Intelligent Agent Technology, 2004 (IAT 2004)*. 2004, 540–543.
16. Snavely, P. J. Custom tool design for game AI. *AI Game Programming Wisdom*, 2006, **3**, 3–12.
17. Sterling, L. and Taveter, K. *The Art of Agent-Oriented Modeling*. Intelligent Robotics and Autonomous Agents. The MIT Press, 2009.
18. Vanhatupa, J.-M. and Heinimäki, T. J. Scriptable artificial intelligent game engine for game programming courses. In *Proceedings of Informatics Education Europe IV (IEE IV 2009)* (Hermann, C. et al., eds). 2009, 27–31.
19. White, W., Demers, A., Koch, C., Gehrke, J., and Rajagopalan, R. Scaling games to epic proportions. In *Proceedings of the 2007 ACM SIGMOD International Conference on Management of Data, SIGMOD '07*. ACM, New York, 2007, 31–42.

## Tehisintellekti rakendus: üldine lähenemine koos tarkvaratoega

Teemu J. Heinimäki ja Juha-Matti Vanhatupa

Arvutimängude realiseerimisel on tähtis genereerida virtuaalne maailm, keskkond, kus tegutseb suur hulk mitte-mängijatest tegelasi. Nende tegelaste jaoks loodav tehisintellekt (TI) teostatakse tavaliselt skriptidena (stsenaariumidena), mille koostamine on mängu realiseerijatele lisakoormuseks. Käsitsi kirjutatavate skriptide hulga vähendamiseks pakume käesolevas artiklis lähenemise ja vastava tarkvaratoe TI funktsionaalsuse projekteerimiseks. Meie lähenemise abil võib mänguväljale luua näiteks erinevate omadustega loomulikul viisil käituvaid ja eesmärgile orienteeritud autonoomseid agente. Vajalikud mõisted spetsifitseeritakse vastavas skriptikeeles ja kohe seejärel saabki agentide konfiguratsioone kiiresti testida. On välja arendatud TI keskkond ja skriptide kirjeldamise raamistik, mis sobib ka üldotstarbelise tehisintellekti loomiseks ning vastavate skriptide automaatseks genereerimiseks. Artiklis on lähenemise rakendatavust näidatud kolme näite varal.