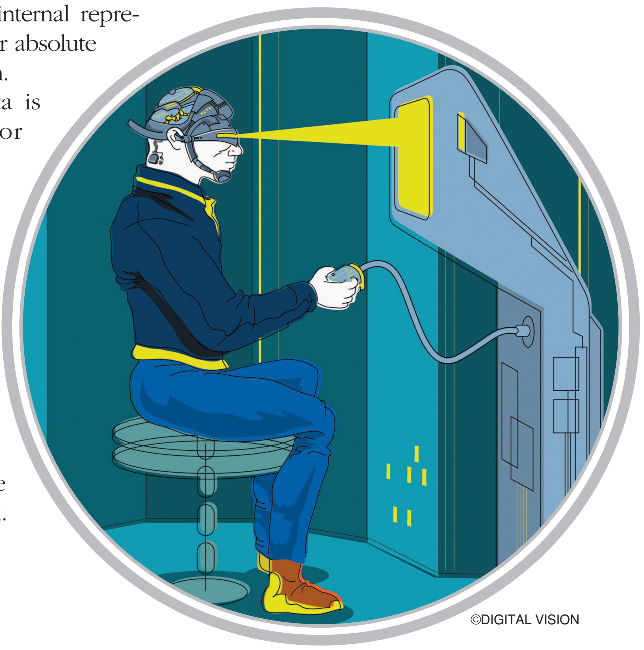
THE APPLICATIONS FOR REAL TIME autonomous software agents include intensive simulations such as interactive computer entertainment software. The layered artificial intelligence (AI) architecture presented is based on the subsumption architecture, a proven control system for mobile robots that consists of parallel levels of independent behaviors. Higher-level behavior modules can subsume lower-level ones through the limited number of inputs and outputs. This AI system can be highly optimized and extended in various ways. A C++ program is presented, demonstrating simple three-layer AI agents. Experiments with this program show that relatively little memory and processor time is required per agent; furthermore, a modern computer game would be able to support thousands of simultaneous agents with this behavior system.


©DIGITAL VISION

Behavioral software agents for real-time games

Samuel H. Kenyon

Layout of subsumption architecture

The subsumption architecture consists of layers of modules that are all networked together. The modules are finite state machines (FSMs) that take inputs from sensors and determine the appropriate actuator outputs. Modules in one layer can subsume the modules in a lower layer by inhibiting or suppressing inputs or outputs. The subsumption architecture is proven to work for real-time mobile robots in unmapped, dynamic environments.

The same attributes that make the subsumption architecture work well for autonomous behavioral robots make it equally ideal for autonomous software agents:

- The designer specifies the desired behaviors.
- It allows for a design-test loop in which new layers are easily added on top of working ones.
- Simple deterministic modules result in reactive, real-time control of an agent.
- It does not need internal representation of the world, or absolute map location information.
- All necessary data is routed through sensor inputs.
- It always has the most recent data directly from the outside world.
- Multiple missions and/or requirements can be carried out simultaneously.
- Cognitive behavior, at least on a primitive level, emerges from the simple modules in parallel.
- The resulting intelligence can survive unknown and changing environments and situations.
- It can still survive to some degree when sensors have errors.

Subsumption-based AI results in agents that are very suitable for soft, and possibly hard, real-time systems. They have no internal representation (the world is the model) and base their actions directly off limited inputs. A corollary of this is that domain knowledge, however small that may be, is always up to date. Plus, the agent is situated in the domain and, as such, gets immediate feedback on its actions.

In a layered system, higher-level behavior can be implemented simply by adding more layers on top of the existing, more primitive behavior layers. A topmost layer can be removed just as easily because the layers below it do not depend on it.

Complex behavior can result from simple sets of rules, especially stimulus-response rules and the use of those as implemented in a subsumption architecture. The purpose of the proposed system of subsumption-based software agents is to develop virtual entities that react intelligently, but quickly, within the tight constraints of memory usage and available execution time on a single processor.

Unlike a nonlayered system of reactive-behavior rules, this system results in what appears from the outside to be cognition on the parts of the agents during any and all circumstances. Since the reactive-behavior rules, embodied in FSMs, are layered, there is an intrinsic conflict-resolution strategy between the rules. A higher-level layer can inhibit (delay or stop) or suppress (override) the action that a lower-level layer would choose. The layers are still deterministic, but in a complex environment with many random variables—especially the interaction with human players—it will seem not only intelligent but unpredictable.

The proposed subsumption system for the agents (the word "agent" is not to be confused with the independent internal modules of the autonomous entities) is similar to the layered character behavior system explained by Funge, in which there is a cognitive layer of goal-directed behavior, a lower layer of predefined behavior, and an arbiter. However, the system described in this article eliminates the entire cognitive model and the costly planning actions. Each agent will have minimal, if any, domain knowledge. Each agent will have some state but no persistent model of the world, at least not yet.

Autonomous agents in virtual worlds

A rigorous test environment in which to prove the efficiency and

0278-6648/06/$20.00 © 2006 IEEE

robustness of these software agents would be an interactive simulation such as a modern computer game that contains a 3-D virtual world. Layered-AI agents are ideal for real-time computer games because of their ability to seem more intelligent with less resources. Most current game AIs use regular and fuzzy FSMs combined with scripting. Layered AI has begun to be used but mostly in nonreal-time situations, such as strategy games, where the AI layers make plans and otherwise use a lot of processing time. Yet future layered AIs, such as the simple system proposed in this article, could be easily scalable and could seem more sophisticated with less scripting.

The layered-AI agents could assume the role of either "bots" in a multiplayer game or the artificially intelligent enemies in a single-player game. The behavior of the opponents could range from low-intelligence drones to high-intelligence virtual humanoids, depending on how much interaction a user has with them. As with all the other simulations in a computer game, the intelligence is a gross approximation of the real thing and will always be less than realistic when examined closely enough. However, fully autonomous agents will add a new unpredictable element regardless of the environment in which they are released.

Traditional game AI

Traditional AI opponents in virtual worlds, even if driven by state machines, merely respond to a set of rigid rules. As scripting became more prevalent in the past decade, the enemies and nonplaying characters (NPCs) would seemingly act intelligent but only in specific circumstances and always in the exact same way. Scripted events and cut scenes are prepackaged and naturally reveal their inability to change or adapt. They also reduce the interactivity, which is a necessary part of any game.

This system separates the AI of the game itself from one central AI to the appearance of a separate AI for each entity.

The vertical layers of increasing high-level specialization simulate real life in a desired way. For instance, it allows the AI to make mistakes just like a person in real life would do. The subsuming of lower levels of basic survival makes it possible for the agents to have flaws. For example, an antagonist might subsume its natural tendency to avoid dangerous and painful objects to attack the protagonist and end up killing itself in the process.

Details of survival

This subsumption system allows an easier way to program details of survival. Unless they are main elements of the game play, most game entities besides the main character (the user) don't need to eat, sleep, exercise, excrete waste, or procreate. They don't even have higher-level human needs or desires. Now they can have a full range of low- and high-level behavior and break the barrier between the needs and special treatment of AI characters versus human players.

Considerations for applying real-time behavioral agents

Reduction of inputs

A typical game AI has all the knowledge that there is to have, in other words, the AI can see the entire map and position the agents appropriately. Consideration is made in a rudimentary way for what each agent can see individually, but they still have knowledge in a nonrealistic omnipresent sort of way from the controlling AI. Using a layered AI would also involve reducing the number of inputs each agent has; each agent is allowed only a certain number of virtual sensors that may be analogous to senses in general or broken down further into relevant types of visual indicators and tactile feedback. All data in the virtual world is filtered through these minimal-amount inputs. However, if the layered AI cannot handle the processing quickly or does not have a layer that deals with a particular virtual sensor, the programmer simply cuts off that input so there are as few in number as possible.

Plans and strategies

Butler et al. report a subsumption architecture implemented for AI agents of a turn-based strategy game that makes use of a short-term memory/communication system to link the layers and the game world together. This system involves plans of actions that each layer produces, along with possible inhibiting signals, and an arbiter that executes the chosen plan.

The system presented herein is simpler and more reactive and as such works well for mass instantiations of real-time autonomous agents. There is still plenty of room to build upon the system to add more intelligent behavior. It depends on the application as to whether various agents need complete plans that the higher-level layers are trying to complete, strategies that can be combined in different ways as the agent sees fit, or by simple reactive rules as it has now. It is not likely that the real-time agents would be developing plans at any time. Plans would have to be the result of combining precoded rules, strategies, or scripts together over time because these agents do not solve the problems of their situation in the form of a plan that can be summed up at any point in time.

Emergent nondeterminism

This subsumption system does not produce explicit plans of action beyond simple goals. However, not only are there emergent strategies already built into layers (described by the FSMs), but these strategies could be modified on the fly. The agent could alter its own FSMs, thus resulting in behavior using a time-distributed plan of action. This would, could be implemented as a special self-mutation actuator.

Self-mutation in this manner is actually a type of learning and also would result in nondeterminism over time. The partial nondeterminism should complement the goal-directed behavior of the higher-level AI layers.

Agent flocks as evolutionary algorithms

If all the agents are self-mutable and do so frequently, then a flock of them would actually be demonstrating an evolutionary algorithm. The random element comes in through the slight randomness of their internal and external starting states and/or the myriad number of situations any agent could experience in a large world, not only by interacting with other AI agents but also with live human players. Only the most competitive versions of agents would survive to the next round as defined by time period or level. If the agent hordes are being deployed by a master AI, this evolutionary algorithm would determine what type of agent is best suited to defeat the enemy. The master AI could then deploy a whole army of agents cloned from the one who best competed previously, and the process would continue.

A flock of subsumption agents could easily display intelligent swarm behavior, especially if they operate predominantly on lower AI layers, which could be controlled dynamically. For instance, the aggregation of a large crowd would result in less-intelligent individuals as the AI system limits the amount of computation for each agent, which is necessary anyway so that the processing of the scene does not slow down. Of course, the crowd will then exhibit its own behavior. Limiting the behaviors within a crowd and having a combination of higher-intelligence, goal-oriented agents with lower-intelligence reactive agents in the same virtual world has been called behavior culling.

Decreasing the processing time

If time goes along the horizontal axis, then making the subsumption structure vertically higher only increases program size, not execution time, within reason. This assumption only works, however, if most layers at any given time are not only being subsumed but are being restricted from executing at all that way only the minimal amount of processing is done. More on this concept will be explained later.

An application such as a computer game appears to be real time to the user, which is at least 30 frames/s. The state of the world as seen from a particular point of view at the current point in time has to be completed in a minimum of about 0.033 s. If the entire AI portion of the program is only allowed 10% of that time, only about 3.3 ms of CPU time is available. The latest trends actually indicate that the AI is allowed up to 20%.

The time spent on the subsumption system for each agent can be greatly decreased simply by not executing the whole system every iteration. Even if an agent represents some supercreature that can make intelligent decisions and adjust its actions accordingly say 30 to 60 times per second, it would probably not change its behavior radically at every iteration. In fact, it would most likely spend several seconds in a general behavior before transitioning to another clearly different one, depending on the stimuli.

To determine an agent's behavior between the full processing points in a smooth manner, the program would continue the current behavior based on the recent history of behavior. It can do this because the actuator variables would also have corresponding variables that would always contain the average over the last x number of iterations. Whether the behavior modules should have access to these variables and possibly to fluctuation calculations,has yet to be determined.

This scheme could be further optimized for applications with a large number of agents by offsetting each agent's particular iteration for full processing so that every iteration has an approximately equal amount of AI processing.

Simulating the parallel processing of the layers

This AI system is designed to run on one physical processor and use less than 20% of the total CPU cycles available. The problem when simulating a subsumption architecture on one CPU is this: How can the different layers be processed in parallel while being mostly independent of each other?

One solution would be to simply give each module its own thread. This would not be a viable solution for a game with many instantiations of subsumption systems, however, unless they were all sharing the same threads; e.g., one thread that would loop through each agent to process one module, with another thread to handle a different module and so on.

Another solution is to multiplex/timeshare the parallel modules. However, an application of several instantiations of a subsumption architecture makes this less optimal. It is not an appropriate place to use interrupts, either.

A better way that would be much faster and use less resources would require a modification to the subsumption architecture:

1) Each layer is reduced to having only one module.

2) Only one layer is processed at a time.

First, this means that the FSM network is simplified immensely, which has the advantage of decreased processing time but the disadvantage of limited behavior complexity. However, a module could be internally expanded to have a network of FSMs inside, which contribute to whatever outputs that layer/module has.

Second, this means that only one layer gets executed in a given iteration, completely subsuming all layers below it and ignoring all layers above it. However, in this new scheme, since the highest layer has the highest priority, the only way for a lower level to have any effect is if a higher layer redirects control downward. This would not be too difficult if the states in the first FSM of a given layer directly correspond to the lower layers, and, after the state is determined, the control jumps directly to the appropriate lower layer or continues in the current layer.

A slightly different way to think of this optimization is: If this layer is not going to subsume the layer below it, then stop processing this layer and process the layer below it instead.

So that this scheme does not increase the length of time for the subsumption system and outweigh the benefits of not processing all layers every iteration, every layer from the second-to-the-top down must have no more than N-1 FSMs, given N FSMs in the layer above a given layer, and assuming that all the FSMs process in average time T, which can be controlled by limiting their calculations and number of states in the first place. Of course, many modules might only have one FSM anyway. Thus the subsumption system as instantiated in an agent, with proper constraints on programming the specific state machines, will always execute within a constant total time limit. Not only will this scheme work for many concurrent instances due to its speed, but the application will be able to tell how many instances it can safely handle based on the known constant time for an individual agent's AI execution.

It may prove necessary, however, to have multiple modules run completely instead of just one, for the behavior to display low-level survival simultaneously with higher-level objectives. If this is the case for a particular application, then the behavior systems may have to be designed so that a merging of values takes place on certain output connection nodes as opposed to complete suppression or inhibition.

Although time-based inhibition/suppression might work, it may be appropriate, especially for the movement of an entity in a virtual world, to use vector addition of multiple outputs in order to generate the resultant single output.

Confusion points

Each level of a layered AI should have plenty of opportunity to become confused, at which point it effectively shuts down and does not subsume any lower layers. This confusion point would actually assist in making the AI routines as fast as possible by skipping the execution of an entire layer if the processing would take too long—i.e., the AI cannot think straight and resorts to more primitive behavior.

The actions

Actions must be of a level appropriate for each layer of the subsumption system, whether scripted, hard coded, or a series of actions.

In Connell's book, the module outputs are not actual motor currents, they are indexes into a lookup table that holds standard motion patterns. In an analogous fashion, an agent's actuators—especially those for higher-level modules—will correspond with several different actual actions. Just like a living animal or human being, the agent does not normally have to actuate every muscle and joint involved in walking and the corresponding balance-feedback system for staying upright. The agent merely actuates "walk" with a direction. A higher-level module may have access to activate more complex series of actions, which could be scripted and may have temporal factors, such as "get into that vehicle and drive away from pursuers" or "hide and wait for the target to be in position."

Memory

These subsumption agents may have an intrinsic way of remembering events, situations, and other entities. Each layer has at least one memory point: the state in which it currently is. Since multiple state machines are allowed, there is an exponential amount of binary information being stored: a 2-FSM system stores a minimum of 2^2 combinations, a 3-FSM system stores a minimum of 2^3 combinations, and so on. These combinations could be considered metastates. In addition to the metastates, each FSM might be augmented with additional memory in the form of temporary variables. How this metastate memory can be purposefully used remains to be seen.

Communication

Not only is communication between the modules running in an agent limited to suppression, but communication between agents is nonexistent. Eventually, it will be necessary to implement a message-passing system between the agents in some way that does not affect what a virtual reality is simulating. For instance, in a game with two teams, the members of a team could talk via radio to each other but not to the other team. A possible solution is given by the dynamic subsumption architecture for agents shown by Nakashima and Noda.

A simple implementation

An example program was written in C++. To keep the implementation as simple as possible, the programmed generation of the AI agents herein do not suppress or inhibit inputs between layers; they only suppress or inhibit outputs (see Fig. 1). This experiment used agents with three-layer, three-module (one per layer) subsumption networks. One could argue that this is not a reasonable test because a real application would need at least 50 modules for its behavior to even approach insect-level intelligence, as is the case in real-life mobile robotics. The point in this case, however, is not to simulate one robot in excruciating detail but to run hundreds or thousands of agents in real time. In a virtual world, the developer can abstract away some of the low-level operations that subsumption architectures for physical robots normally have to handle. For example, if one extended this experiment to have three-dimensional animations of a robot walking to correspond with the agent AI walking, it does not need to handle how it actually walks—that can be done automatically in the graphics engine. If one did want to make a more complicated 50-module or 100-module network of FSMs for some of the agents so that they can explicitly control every virtual joint and body part, that could be done, but it would reduce the number of simultaneous agents of that kind from thousands or millions to hundreds or less and make optimization even more critical. Obviously, the application can mix and match several different types of agents, as long as significant overhead is not introduced for each individual type.

Notice that there is a one-to-one relationship between the layers and the modules, as shown in Fig. 1. This isomorphic setup simplifies the connections between modules and their layer identity.

Note also that the modules do not necessarily have to be made up of FSMs. Layer 0 of AI_Agent is BM_Avoid, which has only one state and consists of merely a series of conditionals to determine what to output. However, the other two behavior modules in use are FSMs. BM_Search on layer 1 is a six-state machine as shown in Fig. 2, and BM_Attack on layer 2 is a two-state machine as shown in Fig. 3.

The first test application is an AI-driven game that looks similar to an ALife (Artificial Life) program. The game consists of a two-dimensional world that is represented with ASCII characters on a screen. A configuration file that is loaded at run time defines the size of the world matrix and the starting positions of the AI agents, who can either be of the "A" team, or the "B" team. Figures 4–6 represent the "A" team with blue squares, and the "B" team with red squares. The objective of the game is to

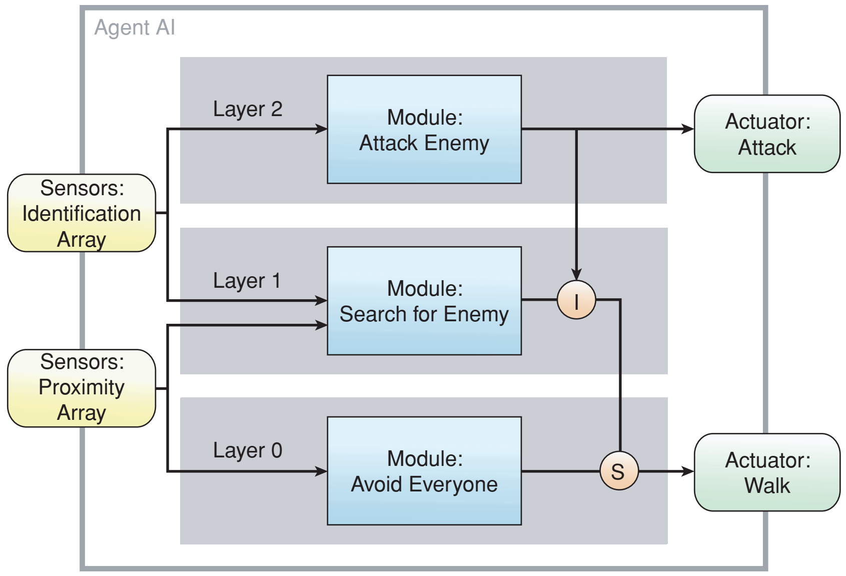


Fig. 1 Agent AI subsumption diagram. An I node indicates inhibition; an S node indicates suppression.

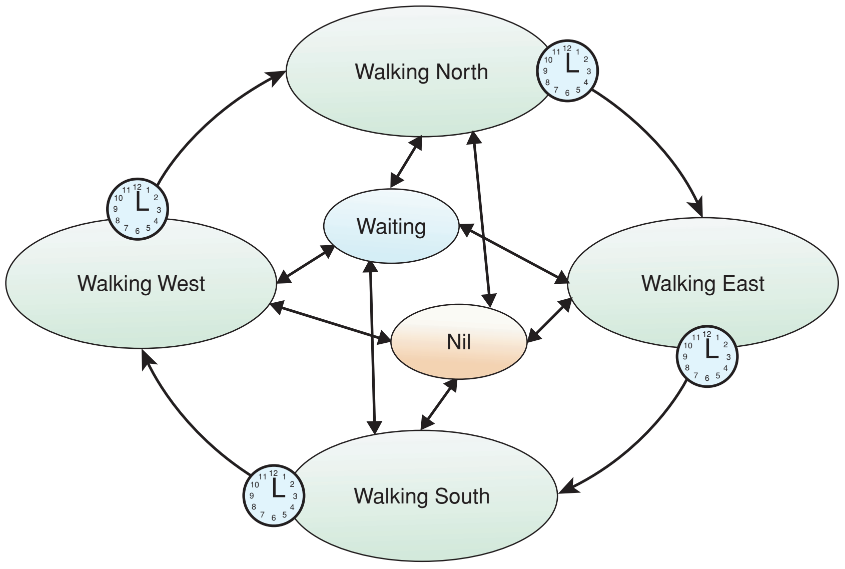


Fig. 2 Finite state machine for the BM_Search behavior module. The clocks indicate timers that trigger state changes. The NIL state is used to indicate when this layer is not choosing an action and thus not suppressing the output of layer 0.

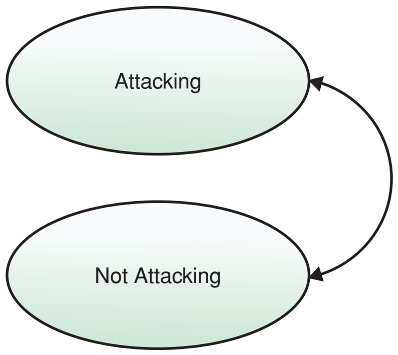


Fig. 3 Finite state machine for the BM_Attack behavior module

destroy members of the opposite team. The agents can only actuate and sense on orthogonal lines, not on any diagonals.

The AI system uses domain knowledge to assign values to the virtual sensors, but none of the domain knowledge is directly transferred to the behavior modules. There are only two virtual sensors available to the modules:

1) proximity, an array of four inputs corresponding to the four sides of an agent, indicating the presence of another agent adjacent to a side

2) identification, an array of four inputs that indicate the ID of an adjacent agent, i.e., what team it is on.

As implemented, proximity and identification are actually the same four-element array, used differently by each layer.

There are only two virtual actuators available to the modules:

1) walk (in specific orthogonal direction)

2) attack (in specific orthogonal direction).

These are the behaviors:

1) Avoid Everyone: If an agent is adjacent to another agent (ignoring diagonals), regardless of what team, it will move either Up, Down, Left, or Right (directions are absolute and correspond to cardinal directions, so Up is really North, etc.) based on deterministic conditional rules (see Fig. 4).

2) Search for Enemy: Before an agent can attack, it must first find an enemy. This behavior causes the agent to walk in one direction for a certain amount of time, then switch to the next direction, and so on, until it finds an agent of the other team, upon which it stops and waits (see Fig. 5).

3) Attack Enemy: If the agent has found an enemy, this behavior activates the attack actuator. In this simple game, it only takes one attack to destroy an opponent, and whoever happens to initiate the attack first wins the battle (see Fig. 6).

Timing, of course, is approximate, and will vary depending on the computer system. However these times would be very similar, and probably faster, on any 2002–2003 gaming system using either a console or PC.

The timing, which includes the entire execution of one step in the world, and the world size itself does affect the execution time regardless of the number of agents. Therefore, the time per agent is actually slightly less. In a more advanced 3-D world, the world size and sizes of the agents would matter much more depending on how the agent sensors were tied into the world and how efficient the scene graph/hierarchy was.

The modest memory usage can be optimized even as it is. The overhead for a more complex AI (in terms of virtual world embodiment) would of course be larger, but it also would be part of the data structures used by the physics and graphics modules of a more complete simulation or game engine.

This test program was successful both in behavior and in the overall

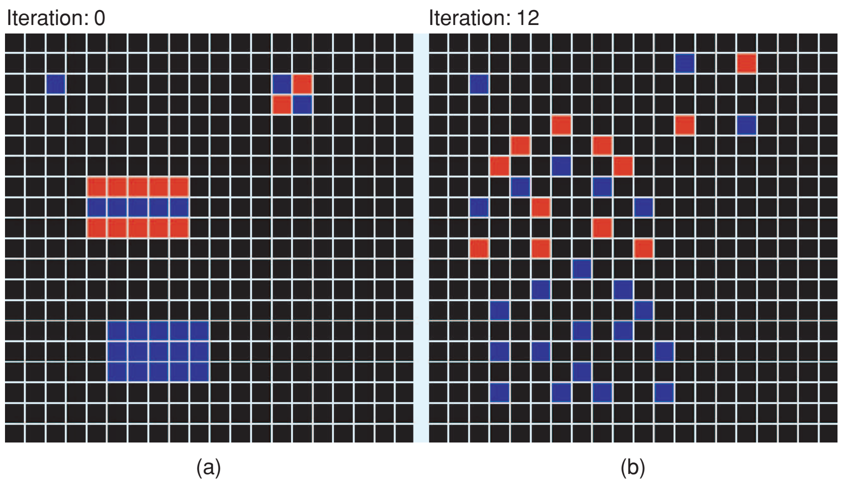


Fig. 4 (a) Agent starting positions and (b) the resulting positions due to Layer 0 behavior. Blue boxes are "A" agents; red boxes are "B" agents.

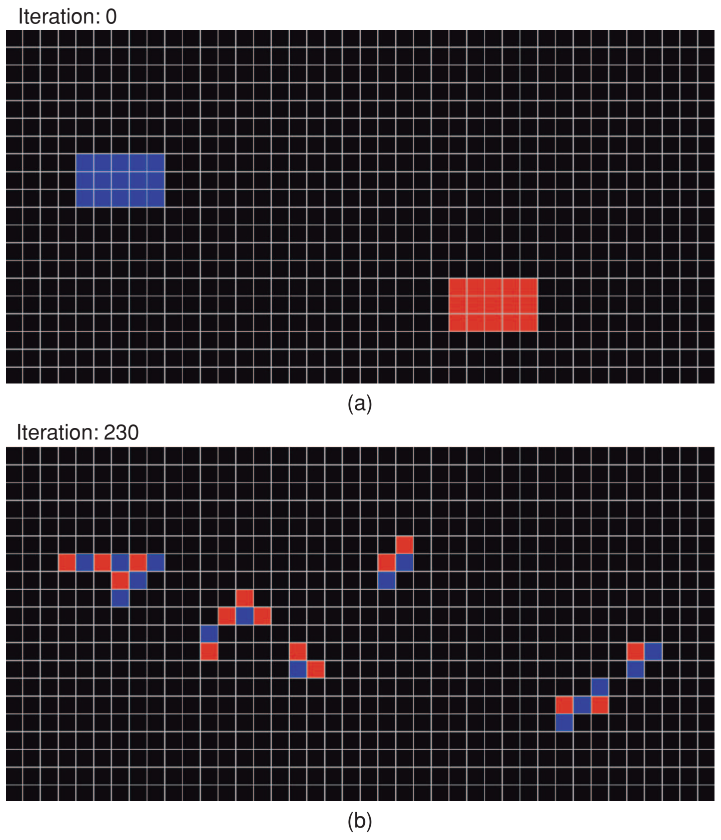


Fig. 5 (a) Agent starting positions and (b) the resulting positions due to Layer 1 + Layer 0 behavior after several iterations. Blue boxes are "A" agents; red boxes are "B" agents.

approximate time spent processing the subsumption AI for each agent; and although it was programmed for simplicity, it could be easily incorporated into a more rigorous interactive simulation.

Performance

Many timing experiments were done using the experimental agents described in this article. As shown in Table 1, the average time needed for each agent at each iteration was 118.7195 ns on the test computer (CPU was 1.6 GHz AMD Athlon XP 1900). For a real application, such as a game, the maximum time to process each frame is 0.03333 s if the video is updated at 30 frames/s. If 20% of that time is dedicated to the AI system, then that gives 0.006667 s per frame for the agents. The maximum number of simultaneous agents (assuming 1 iteration per agent per frame) is 56,154.

The memory required for each agent is 16 B, plus overhead of 3,224 B to the program. Thus, for the maximum number of simultaneous agents, the total memory usage will be 901,688 B which is less than 1 mb.

Future modifications

One possible modification is that of interagent communication in a way that does not cheat the rules of the world containing the users and agents. Also,

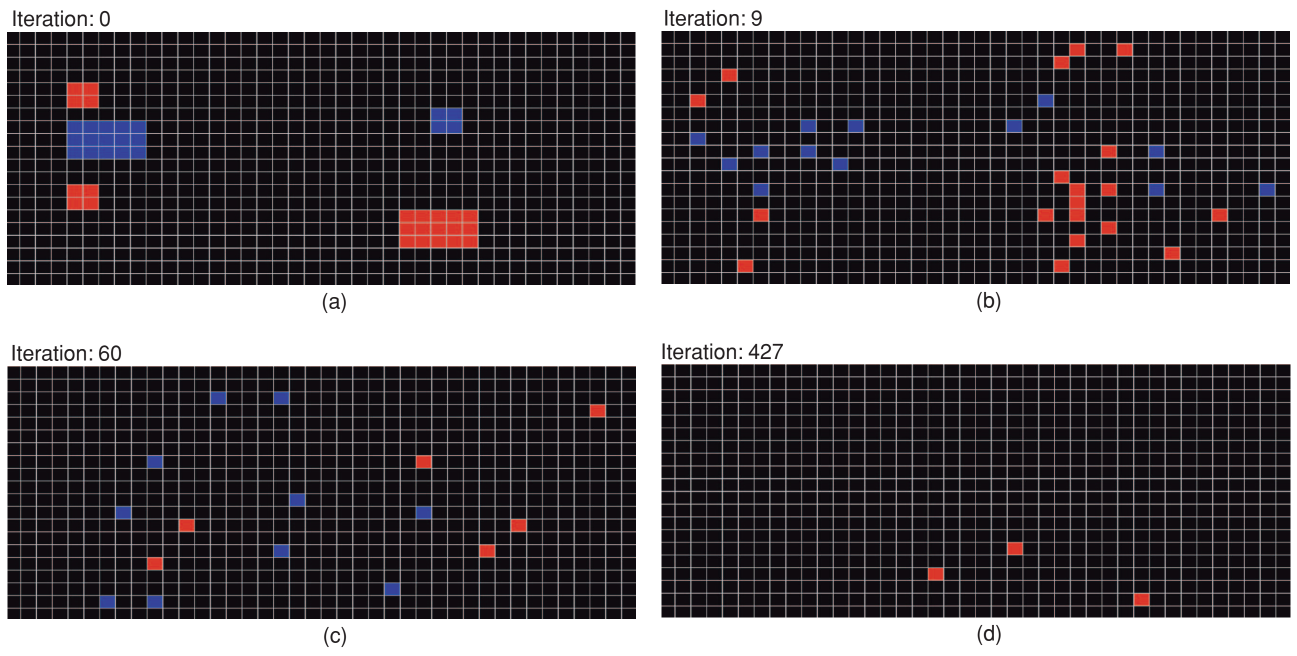


Fig. 6 Agent positions due to Layer 2 + Layer 1 + Layer 0 behavior: (a) starting positions, (b) after nine iterations, (c) after 60 iterations, and (d) after 427 iterations, in which the "B" team has defeated the "A" team; the agents will now remain in that position indefinitely. Blue boxes are "A" agents; red boxes are "B" agents.

predefined action scripting would be a major advantage. In fact, the entire system would benefit from being completely script based so that at run time the configuration or script files are loaded, filling in the FSMs, and defining the necessary sensors and actuators used by each module. The limitations of the narrow and vertical (in abstract design space) subsumption structure may be alleviated with various modifications as long as the requirement of real-time capability is met for the given application. An interesting variation would be the layers spread out among modules of different priorities in different subsumption structures, or behavior layers that merge, split, or cross over each other. Switched at some point by the agent itself or a governing AI arbiter, the selective groups of behaviors would give the agent the ability to change tactics and appear even more intelligent.

Table 1. Timing results.

World Size	Number Agents	Number Iterations	Time (s)	Time/Iteration (ns)	Time/Iteration/Agent (ns)
40 × 20	42	500,000	2.36	4,720	112.3810
40 × 20	42	5,000,000	13.08	2,616	62.2857
40 × 20	42	10,000,000	34.77	3,477	82.7857
80 × 40	702	500,000	51.63	10,3260	147.0940
80 × 40	702	5,000,000	582.43	11,6486	165.9345
80 × 40	702	10,000,000	995.69	99,569	141.8362
				Average Time/Iteration/Agent:	118.7195

Conclusions

A simple version of the subsumption architecture is presented, in which the one-per-layer behavior modules are not networked except for strict downwards subsumption. The system allows for future expansion to series of FSMs, or other deterministic code, within each module. The point is to limit the number of modules actually processed in a row to preserve precious execution time, which would be more difficult with combinatorial networks of asynchronous modules that have variable times from initial input to final actuator output.

The described behavioral autonomous agents are ideal for real-time applications where large numbers of seemingly intelligent entities interact simultaneously with tight resource/processor restraints, especially computer games. The high performance of agents, which each have three behavior layers, is validated with an experimental program. There are many different ways this type of agent system could be used successfully, from individual game opponents to simulating packs of wild animals or swarms of insects.

Read more about it

• R.A. Brooks, *Cambrian Intelligence: The Early History of the New AI.* Cambridge, MA: MIT Press, 1999.

• J.L. Jones and A.M. Flynn, *Mobile Robots: Inspiration to Implementation.* Wellesley, MA: A.K. Peters, 1993.

• J.D. Funge, *AI for Games and Animation: A Cognitive Modeling Approach.* Natick, MA: A.K. Peters, 1999.

• S. Woodcock, "Game AI: the state of the industry 2001–2002" *Game Developer*, vol. 9, no.7, pp. 26–31, July 2002.

• G. Butler, A. Gantchev, and P. Grogono, "Reusable strategies for software agents via the subsumption architecture," in *Proc. Sixth Asia Pacific Software Engineering Conf.*, Dec. 1999, pp. 326–333.

• J. Blow, "Toward better scripting: Part 1," *Game Developer*, vol. 9, no. 10, pp. 14–17, Oct. 2002.

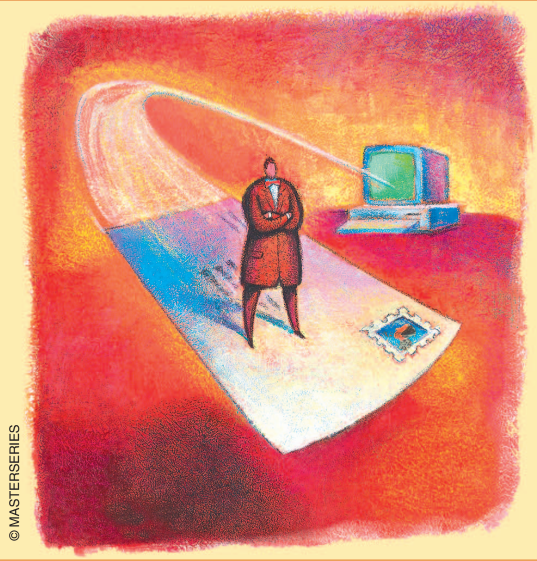
• J.H. Connell, *Minimalist Mobile Robotics: A Colony-Style Architecture for an Artificial Creature* San Diego, CA: Academic, 1990.

• H. Nakashima and I. Noda, "Dynamic subsumption architecture for programming intelligent agents," in *Proc. Third Intern. Conf. Multi Agent Systems*, July 1998, pp. 190–197.

• C.G. Langton, "What is Artificial Life?" Zooland, 2004 [Online]. Available: http://zooland.alife.org/

About the author

Samuel H. Kenyon (ksamuel@coe.neu.edu) obtained his B.S. in computer engineering technology in 2004 from Northeastern University, Boston, Massachusetts. He now works for iRobot Corp. and takes occasional graduate classes at MIT.



Tell us what you think; the sky's the limit

Every issue of *IEEE Potentials* features articles that are on the cutting edge of ideas with subject matter that often involves new technology, new applications of existing technology, or the results of new research. Such new ideas are likely to provoke questions and discussion among colleagues and friends. They may also generate controversy.

Whatever your reaction to an article, essay, IEEE program, or general state of the world, *IEEE Potentials* would like to hear about it. Instead of venting in obscurity or only to those within earshot, put your thoughts into an e-mail and send it to <e.m.smith@ieee.org>. Remind your friends and colleagues to do the same.

E-mails that are received may be published in the next available issue of *IEEE Potentials* in our "Letters to the Editor" column. Letters may be edited for publication.