

Layered Specification of Intelligent Agents

Paul Scerri, Johan Ydrén and Nancy Reed

Department of Computer and Information Science
Linköping University, SE-581 83 Linköping, Sweden
pausc@ida.liu.se johan.ydren@meridium.se nanre@ida.liu.se

Abstract. Interactive simulation environments with large numbers of intelligent agents are becoming increasingly common. In general, knowledge of precisely what agents should do in the environment is not an agent developer's area of expertise, rather it is a *domain expert's* expertise. In this paper we present an approach to specifying agents that takes advantage of the domain expert's knowledge where possible, but still allocates difficult programming tasks to expert programmers. In particular, the task of specifying agent behavior is layered and tasks are allocated according to the relative amounts of programming and domain expertise required for each one. Results are presented for an implementation of the technique for RoboCup players. An interesting benefit of the layered specification we observed was that an efficient, parallel development approach emerged.

1 Introduction

Interactive simulation environments with large numbers of intelligent agents are becoming increasingly common. Such simulation environments often have agents playing the roles of humans within the simulation [17]. Examples of this type of environment are military simulations [16, 8], training [1], computer games [4] and RoboCup [10]. In general, knowledge of precisely what agents should do in such environments is not an agent developer's area of expertise, rather it is a *domain expert's* expertise. It follows that it is desirable to have domain experts as closely involved in the development of the agents as possible. Unfortunately in general, domain experts will not also be programming experts, rather they will be average computer users, i.e. they know *what* the agents should do but not necessarily *how* to get the agent to do the job. Easy-to-use specification tools are necessary for allowing domain experts to directly specify the behavior of the agents.

The ideal situation would be to have domain experts directly specifying all agent behavior using an off-the-shelf environment. Unfortunately that is a difficult, and perhaps unrealistic goal, take a smaller step we take an approach to agent specification which combines the desirability of non-expert agent specification with the practical limitation that programming agents is hard [3].

In particular we propose that the task of specifying an agent be broken into layers where the more difficult-to-program parts are assigned to expert programmers who are provided with appropriate expert programming tools, while the

domain knowledge intensive aspects of specifying the agent are assigned to domain experts with easy to use tools. In particular, agent expertise is required for specifying low level *skills*, an intermediate level of agent programming experience is required for developing *individual strategies* and primarily domain experience is required for specifying *scenarios*. Figure 1 illustrates this idea.

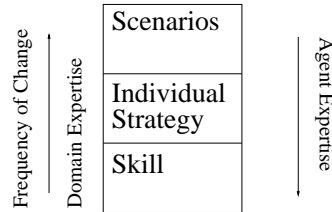


Fig. 1. The relative frequency of change and level of expertise needed in three abstract levels of the agent specification process.

At each layer of the design, a different tradeoff can be made in the specification tools between expressive power and ease of use. At the skill layer, expressive power is the major concern, at the individual strategy layer, both expressive power and ease of use have equal weight. Finally at the scenario specification level, ease of use takes precedence. Ideally no tradeoffs would be necessary but in reality they are required. Our approach is to acknowledge that tradeoffs will be made and to make them intelligently.

Many researchers have explored the idea of how to create agents composed of multiple layers. Usually the layers are selected to be appropriate to the kinds of agent reasoning required at different levels of abstraction (e.g. [7, 12]). In this paper we take a slightly different view on layering, i.e. we look at layering from the perspective of the specification tools a designer uses to specify agents. When separating an agent specification into layers, emphasis is on providing appropriate specification tools rather than on an appropriate runtime architecture (although clearly this cannot be ignored.)

Skill specifications need to be available to be used in the individual strategy layer and individual strategy specifications made available in the scenario specification. In effect, in the lower layers *building blocks* are specified which are used as the basis of specifications at the next layer. A critical aspect of making the layered approach work is that the building blocks exported from one layer support a good specification process at the next layer.

The precise types of tools required for each specification layer will vary depending on the agent's destination environment. For some environments the specification tools will need to support formal verification, while for other environments supporting rapid prototyping or enabling specification of physical distribution of the agents will be critical. The particular layering breakdown

presented here is tailored to a specific type of domain with the following characteristics:

- There exists a large amount of expert domain knowledge that is not general knowledge, e.g. how to fly a combat aircraft or soccer strategies.
- At higher levels of abstraction the required agent behavior changes more often than at lower levels of abstraction.
- A relatively large number of similar agents are immersed in the same environment.
- Reasonably “intelligent” performance is expected of the actors. E.g. if the agents have only very simple strategies then the individual strategy layer may be superfluous.

Despite the apparently strong constraints listed above on the domains, a wide range of entertainment and training applications do meet these requirements. Furthermore, it should be straightforward to adapt the ideas to use with other types of applications.

For RoboCup 1999, we developed a layered specification system for a simulation team called the Headless Chickens III (HCIII) [14]. Low-level skills were written in Java, the individual strategies were created with a graphical system for developing behavior-based agents and the team strategies were built using a team-level strategy editor based on the idea of a coach’s white board.

Three important observations were made while using this system. Firstly, after an initial start up time development was able to proceed at each layer almost completely in parallel. Secondly, the rapid scenario development system allowed extremely rapid, yet dramatic and effective, changing of team behavior. This enabled us to quickly adapt the HCIII strategy for different opponents during the 1999 World Cup. Finally, the runtime architecture layers were found to be relatively interchangeable. In fact for the 2000 RoboCup World Cup, the team scenario specification environment is being reused with two different individual strategy and skill runtime architectures (in different teams) and specification tools.

2 Layered Development Model

When beginning to develop agents for a new domain, appropriate development tools must first be designed/selected. Some analysis of the domain will need to be done to establish what properties the tools and resulting agents should have. The specific specification tools chosen for each development level depend on the requirements of the domain, e.g. reactive agent or formal verification. An important constraint is that the tools must export a specification in a form that the layer above can interpret. Once the tools are designed (or in practice probably simultaneously) a mapping to a runtime architecture needs to be developed. Next, skills, individual strategies and scenarios are developed and tested in parallel. There will be significant feedback between the specification layers, especially early on, but over time the bulk of the work will be centered in the

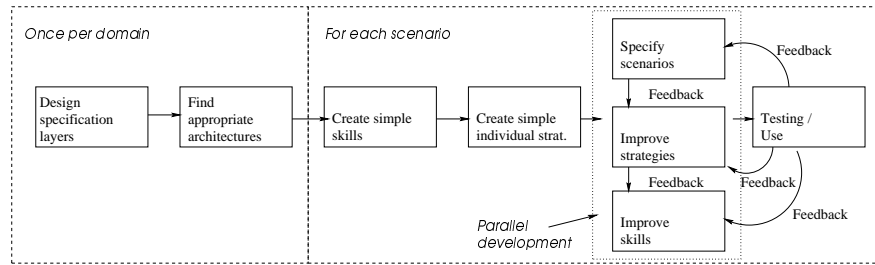


Fig. 2. The agent development process.

scenario specification tool. Ideally a stable set of skills and a library of individual strategies will be developed and most effort will go into creating and modifying scenarios, hence most work will become the domain expert’s sole responsibility. See Figure 2 for an overview of the proposed development process.

At the skill layer the basic abilities of the agents are defined. For example in the air-combat domain skills would involve basic maneuvers such as landing and ground-avoidance. The individual strategies will be developed with a specification tool tailored to the type of agents required in the domain, e.g. a tool for specifying reactive rules or a tool for specifying sequences of actions. The individual strategies use the skills as their basic building blocks. For the air-combat domain individual strategies may include different flying formations and different opponent engagement strategies. Finally, at the top level, entire scenarios potentially consisting of many agents, or simply a particular “mission” for one agent, are defined. The scenario level strategies should be specified in an environment particularly suited to the domain and matching the domain expert’s ways of explaining strategies.

Notice that the layering breakdown refers only to divisions in the tool support, not to where the layers in the runtime agent will be. The runtime architecture of the agent may not be layered in the same way as the tools (or not layered at all). The tools may simply be different ways of specifying things for the same runtime architecture or, map to possibly a seven layer architecture. The HCIII mapped the three specification layers to two runtime layers (see Figure 3) while Headless Chickens IV (HCIV – our RoboCup 2000 entry) maps to a single runtime layer. Although in the actual agent the same layered structure may not exist, it is important to preserve the illusion that it does for the benefit of the user of the specification tools. The usability of the tools will decrease if, for example, at the scenario specification level the designer needed to consider how the compilation process for the upper two layers works.

2.1 Skills

Skills are simple, self-contained pieces of agent functionality. Examples of skills include *landing* for aircraft simulation and *dribbling* for RoboCup players. From

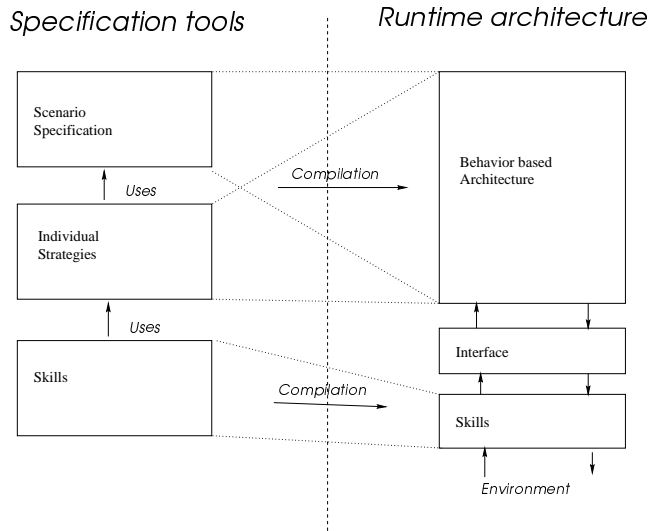


Fig. 3. An example mapping between specification tools and the agent architecture.

the perspective of the user of higher specification layers a skill is turned on and continuously attempts to achieve some simple task until being turned off.

The skills of agents are not expected to change much over the lifetime of the simulation or to vary from scenario to scenario. For example the way a pilot lands his aircraft is fairly independent of the mission in which he was involved. As the skills will often be fairly complex control routines, low level general purpose languages and associated general purpose programming environments will often be well suited. Furthermore in many simulation environments at low levels of abstraction a domain expert's knowledge may not be relevant due to the design of the agent to environment interface. For example the interface between a RoboCup player and the SoccerServer for a kick is a string "kick power direction" which makes the domain experts knowledge of muscle movements required to dribble quite irrelevant. Likewise, the particular movements of the control stick to land an aircraft are probably not relevant when specifying the skill for a simulated pilot to land a simulated aircraft.

Because of the relative infrequency of the required changes to the skills and the difficulty in encoding the knowledge it is reasonable that expert programmers be employed to create efficient, robust skills using languages and tools most appropriate for the task. The specification tools at the skill layer should support an expert programmer in efficiently defining skills. The emphasis in the tool should be on providing an effective means for specifying a low level skill in a machine interpretable format.

HCIII skills were written in Java, i.e. they were completely in the realm of an expert programmer. Most skills were parameterized by a field position, for example the position to kick to, dribble to or run toward.

2.2 Individual Strategies

The individual strategies are the abstract behaviors of a single agent. For example in air combat simulation an individual strategy may be a certain opponent engagement strategy. As a whole the individual strategies of the agent define a sort of agent “template” which will be instantiated for a particular scenario. From the perspective of the user of the scenario specification layer individual strategies are the abstract, parameterizable behaviors that the agent has at its disposal. Some mechanism needs to be developed that allows the team level strategy editor to use the individual strategy “templates” and to allow the individual strategy editor to use the skills.

For the purposes of the scenario specification layer the behaviors will be atomic. The individual strategies of an agent will change more often and more markedly over the lifetime of the simulation than will the agent’s skills. On the other hand, for the targeted domains, the individual strategies will change less often than the scenarios. For example in RoboCup, an individual’s strategy for marking another player will change infrequently, while the skill of dribbling will almost never change and the team formation (i.e. scenario) will change the most often. Because of the slightly higher rate of change, and more importantly because there is more domain knowledge relevant to individual strategies, domain experts need to be closely involved in specifying individual strategies. However because of the potential complexity of the strategies the problem of encoding the strategies will often not be possible by a domain expert alone (at least with the current state of the art). When designing tools for specifying individual strategies, a tradeoff between ease of use and expressive power should be made more towards the side of ease of use than for skills, though more towards expressive power than for scenarios.

For HCIII individual strategies are specified in a graphical editor – no code needs to be written. The tool allows relatively fast development of fairly complex behavior-based agents. However the complex concepts underlying these systems mean that the average end-user (i.e. domain expert) cannot do significant development work unassisted. To allow the team level specification system to “understand” individual strategies a simple “language” is embedded into the specification system (for HCIV this interface is in XML). When a team level strategy is created, the behavior specification is “compiled” into separate players, i.e the scenario and individual strategies are combined together into a single runtime layer (see [14] for details).

2.3 Scenario Specification

The scenario specification tool is the most important of the specification tools. Over the lifetime of the agents it is the scenario specification system that will

be used the most often. The scenario specification is used to design the high level “organizations” or “missions” of the agents. For example in RoboCup it is used to specify team formations and tactics, in air-combat simulation it is used to specify missions and for computer games to specify the initial positions and movements of the “characters”.

At the scenario specification system’s high level of abstraction expert knowledge is crucial – even more so than for the lower levels. The scenario specification tool should support the expert in their job of developing appropriate behavior for the application. There is a subtle but important difference between the design emphasis for the scenario specification tool and the lower level tools. At lower levels the specification tools were chosen or designed to simplify the designers task of translating their knowledge into a computer understandable format. At the scenario level there is far more experimentation, iteration etc., so the emphasis moves from providing means to effectively encode well understood activities to creatively creating specialized scenarios. In particular it is desirable that a scenario specification system have the following characteristics:

- A specification method that allows domain experts, rather than system experts, to interact easily with the system.
- A specification method that is natural to the domain expert. Most desirable is to draw on explanation methods from the domain for inspiration.
- Very rapid development abilities.

To integrate with the rest of the system the scenario specification tool needs to be able to “understand” the templates provided by the individual strategy tool. Additionally it needs to instantiate the provided templates into specialized agents in some way.

The scenario specification system of HCIII is based on the idea of a coach’s white board, as shown in Figure 4. In other words we took a medium for expressing strategies that a domain expert was used to using and attempted to reproduce that medium in the computer. The “white board” provides different panels for each of the modes of play that the agents know about. The known modes are determined by looking for a special keyword in the agent template (imported from the individual strategy specification tool). The domain expert (i.e. a soccer coach) places the players, represented by circles, on a diagram of the ground and then indicates directions they should dribble and pass in each of their modes. The tool allows coaches to express strategies in a way that they are used to as well as encouraging creativity and experimentation. For the RoboCup domain there was a clear real-world analogy between the white board and the specification system. For other domains the scenario specification method may appear much different. For example air-combat tactics may be written as a set of responses to possible situations, using a specification system styled after they way tactics are normally written down.

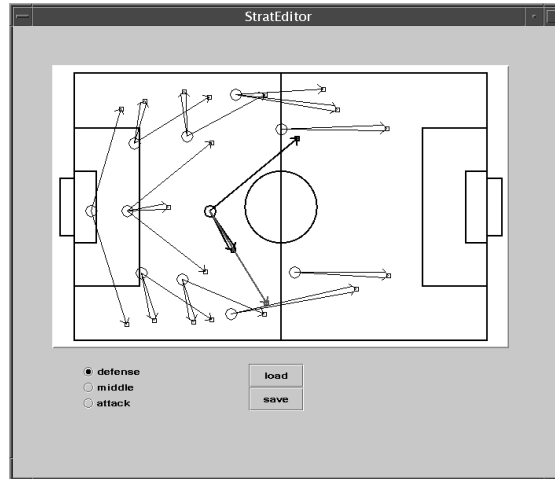


Fig. 4. A screen shot of a defensive team formation in the HCIII scenario editor.

3 Discussion

Developing a layered model like the one described in this paper will often require significant development effort for a new domain. For example, the development of the team strategy editor for HCIII was far from a trivial exercise. However in the RoboCup case it was decided the complexity of designing team strategies and the advantages to be gained by being able to quickly change strategies outweighed the extra effort required. This will not always be the case for a domain. Sometimes the relative simplicity or infrequency of creating scenarios will not justify the development time of a scenario editor. In all cases, the significant cost of creating domain specific, easy-to-use scenario specification environments can be largely offset by the removal of the expert programmer from the scenario design and testing loop.

A number of specification environments allow specification across a very broad range of abstraction levels, i.e. skill through scenario, e.g. [13, 5]. Such a specification means that the “artificial” breakdown into layers of our layered model need not be made. This in turn avoids some potentially inelegant designs forced by the layering. On the other hand using the one specification environment for all aspects of development means that the same tradeoffs between ease of use and expressive power exist at all levels. Layering allows tradeoffs to be made on the basis of the type of specification to be done at a particular specification level. For example more flexible and complex low level (and hence generally harder to use) tools are advocated for skills while easy-to-use (and hence less flexible) tools are used for scenario specification. If the same tool is used across all levels, the same tradeoff must be made across all levels perhaps resulting

in lack of expressive power at the lower levels (e.g. AgentSheets [13]) and/or undesirably low usability at higher levels (e.g. dMars [5]).

Some development environments, such as JACK [9] and AgentBuilder [15], provide different mechanisms for specifying individual and team behavior. However, unlike in the tool specification editor presented here, the tools provided for specification of team behavior in both Jack and AgentBuilder require an solid understanding of programming and of agents. However for the types of domains and applications these tools target, e.g. integrating legacy systems in an intelligent way, this is a perfectly reasonable design decision as the *domain expert* in this case is a programmer! If the tools were targeted to domains where the domain experts were not computer experts a different approach would probably need to be taken.

In layered agent architectures, such as 3T [2] and Raps [6], agents usually have three or four architectural styles in different layers at runtime. The rationale is much the same as the rationale for layered specification tools, i.e. to provide appropriate tools for the job at different layers of abstraction. In the case of tools the job is specification, in the case of agent architectures the job is action selection. However specification layering is not simply a translation of the layered idea from the computational side of agent development to the specification tool side. Firstly most layered architectures predefine the structure of the layers before examining the domain, i.e. *general* layered architectures are developed then applied to a particular domain. On the other hand our model advocates selection/development of tools based on the requirements of the domain. By analyzing the domain before choosing the layers the appropriateness of the overall system to the domain is likely to be better. The improved suitability comes at the cost of having to build new tools for each domain. Secondly, the layers for layered agent architectures exist both at design time and at runtime, whereas for layered specification the layering may only exist at design time. For example in the RoboCup system presented here the scenario specification and individual strategies are compiled together so there exists only two layers at runtime.

The abstract model was used to develop a team that competed in the 1999 RoboCup World cup and finished 5th out of 35 teams. However it was not the quality of the agent behavior that was interesting rather the development process used to create the team. In the last weeks leading up to the competition there were three separate “groups” working almost completely in parallel to develop HCIII. One expert programmer worked at the lowest level of abstraction constantly trying to improve the skills of the team. Another programmer worked on the individual strategies of players. Finally, a group of students, who joined the project in the last months worked exclusively on the team strategies. Whenever significant improvements were made in either the skills or individual strategies the changes were passed to the team strategy group.

Feedback down levels of abstraction occurred frequently. Usually feedback from the team to individual player to the skill layer was in the form of comments like “Players need to be more keen to shoot when around the goal” or “They don’t follow the ball far enough from their position”. Feedback from the individual

strategies to the skills was usually something like “Can we make them shoot harder” or “They should look in the area where we tell them to pass to make the pass hit, rather than just kicking to the exact point”. The critically important factor of the development process was that all specification layers could work easily in parallel allowing a large team to work relatively independently yet still effectively.

Although the RoboCup implementation of the layered model was successful and useful it was at least partially due to the characteristics of the domain and the specifics of the architectures chosen, rather than just a good model. All layers, both on the specification and runtime side, were very reactive. The reactiveness allowed layers to “make decisions” independently of previous actions drastically simplifying specification system design. Even more critically it made the interfaces between the tools, e.g. the way a skill specification was imported into the individual specification tool, simple to design and build. The reactive nature of the layers also meant that the only communication between layers was in the form of the upper layer setting the new group of behaviors or skill at the next layer down – the environment being used to provide feedback about the effects of that action. The design of scenarios which involve communication protocols may be fundamentally more difficult and not possible for genuine end users.

4 Conclusion and Future Work

Experience with RoboCup using an instantiation of the specification model presented here supports the idea that layering the specification task provides benefits with respect to other specification methods. In particular appropriate tools were available at the different levels to make the specification task as easy as possible. A further, somewhat surprising, result was that the layering led to a very efficient and practically desirable development process.

There seem to be a variety of other domains/environments where such an approach to designing specification tools would be appropriate. MissionLab [11], for example, successfully uses a similar layered structure for specifying multi-robot missions. Hopefully more agent designers will consider layered specification approaches when examining possibilities for deploying agents in new domains.

Acknowledgments

This work is supported by Saab Corporation, Operational Analysis Division, the Swedish National Board for Industrial and Technical Development (NUTEK) under grants IK1P-97-09677, IK1P-98-06280 and IKIP-99-6166, and the Center for Industrial Information Technology (CENIIT) under grant 99.7.

References

1. R. Bindiganavale, W. Schuller, J. Allbeck, N. Badler, A. Joshi, and M. Palmer. Dynamically altering agent behaviors using natural language instructions. In *Proceedings of Fourth International Conference on Autonomous Agents*, 2000.
2. R. Bonasso, R. Firby, E. Gat, D. Kortenkamp, D. Miller, and M. Slack. Experiences with an architecture for intelligent reactive agents. *Journal of Experimental and theoretical Artificial intelligence*, 9(1), 1997.
3. J. Bradshaw, M. Greaves, H. Holmback, T. Karygiannis, W. Jansen, B. Silverman, and Alex Wong. Agents for the masses. *IEEE Intelligent Systems and their applications*, 14(2):53–63, 1999.
4. Johanna Bryson. Creativity by design: A character based approach to creating creative play. In *AISB Symposium on AI and Creativity in Entertainment*, 1999.
5. M. d’Inveron, D. Kinny, M. Luck, and M. Wooldridge. A formal specification of dMars. Technical report, Australian Artificial Intelligence Institute, Melbourne, Australia, November 1997.
6. James Firby. Task networks for controlling continuous processes. In *Proceedings of the Second International Conference on AI Planning Systems*, June 1994.
7. Erann Gat. On three layered architectures. Online Publication, May 1997.
8. R. Hill, J. Gratch, and P. Rosenbloom. Flexible group behavior: Virtual commanders for synthetic battlespaces. In *Proceedings of Fourth International Conference on Autonomous Agents*, 2000.
9. A. Hodgson, R. Rönquist, and P. Busetta. Specification of coordinated agent behavior (the SimpleTeam approach). Technical report, Agent Oriented Software, 2000. <http://www.agent-software.com/>.
10. Hiraoki Kitano, Minoru Asada, Yasuo Kuniyoshi, and et. al. RoboCup: A challenge problem for AI. *AI Magazine*, 18(1):73–85, Spring 1997.
11. Douglas MacKenzie. *A design methodology for the configuration of behavior-based mobile robots*. PhD thesis, Georgia Institute of Technology, 1996.
12. K. Pfleger and Barbara Hayes-Roth. Using abstract plans to guide behavior. Technical Report KSL-98-02, Knowledge Systems Laboratory, Stanford, Jan 1998.
13. Alexander Repenning and Andri Ioannidou. Behavior processors: Layers between end-users and java virtual machines. In *Proceedings of VL’97*, Capri, Italy, September 1997.
14. Paul Scerri and Johan Ydrén. *RoboCup-99: Robot Soccer World Cup III*, chapter End User Specification of RoboCup Teams. Springer, 1999.
15. Reticular Systems. Agent builder: An integrated toolkit for constructing intelligent software agents. Technical report, 2000. <http://www.agentbuilder.com>.
16. M. Tambe, K. Schwamb, and P. Rosenbloom. Constraints and design choices in building intelligent pilots for simulated aircraft. In *AAAI Spring symposium on "Lessons Learned from implemented software architectures for physical agents"*, 1995.
17. Milind Tambe, W. Lewis Johnson, Randolph Jones, Frank Koss, John Laird, Paul Rosenbloom, and Karl Schwamb. Intelligent agents for interactive simulation environments. *AI Magazine*, 16(1):15–39, Spring 1995.