Presented at
Behavior Representation in Modeling and Simulation (BRIMS)
May 2004

Behavior Design Patterns: Engineering Human Behavior Models

Glenn Taylor
Robert E. Wray, PhD
Soar Technology, Inc.
3600 Green Court Suite 600
Ann Arbor, MI 48105
734-327-8000
glenn@soartech.com, wray@soartech.com

Keywords:
Intelligent Systems, Human Behavior Models, Computer Generated Forces, Agent-based systems, Design Patterns

ABSTRACT: *This paper describes new research that attempts to address cost and development time bottlenecks in the implementation of high-fidelity human behavior representations. The proposed solution, behavior design patterns (BDPs), is an extension of design patterns in traditional software engineering. A crucial challenge for BDPs, and the fundamental limitation of existing design patterns, is capturing the unique computational properties of human behavior models (HBMs), which require parallel, goal-oriented execution to achieve autonomy, flexibility and adaptation in task performance. We demonstrate example patterns extracted from an existing HBM, TacAir-Soar, and describe how they can be applied in other HBMs. Although the results are preliminary, one advantage to the proposed approach is the development of solution catalogs, which can be used to improve development today, even while more powerful development tools are being designed and constructed.*

1. Introduction

Developing high-fidelity models of human behavior is a challenging endeavor, often resulting in high costs and long development times. High costs are, in part, the result of little formal reuse of software in building these models. Obviously, reuse is difficult when the modeled tasks are very different (e.g., armored cavalry vs. fixed wing aircraft). In addition, differences underlying implementation languages, cognitive architectures, and simulation environments can also contribute to the lack of reuse. Cognitive architectures attempt to capture some of the regularities in human behavior at the level of symbolic processing and subsymbolic tuning of behavior to the task [1, 2]. Our goal in this work is to extend reuse to encoding common patterns of human knowledge-level behavior. Because such a goal is obviously open-ended, we focus specifically on military domains, where training and common doctrine enforce and reinforce patterns of human behavior [3].

The inspiration for this work is in traditional software engineering techniques of design patterns that have proven useful in building large software systems. Design patterns capture and generalize successful designs for common problems. We are extending the notion of a design pattern to address the challenges of building human behavior representations. In doing so, we examine some of the qualities of intelligent systems that make them unique among software systems, and how these qualities might inform new ways to describe designs for intelligent systems. We present an analysis of an existing human behavior model, TacAir-Soar [4, 5], in an effort to demonstrate the prevalence of design patterns in implemented systems.

2. From Design Patterns to Behavior Design Patterns

This section describes the software engineering notion of design patterns and how they have been used in traditional software development. Patterns for human behavior representations (HBRs) must be different from traditional design patterns, because the computational support for traditional object-oriented systems is inadequate for the requirements of autonomous, robust, high fidelity HBRs. We introduce Behavior Design Patterns (BDPs), our proposed adaptation of design patterns to capture the different computational infrastructures needed for HBRs.

2.1 Design Patterns

In typical software engineering, design patterns represent concise solutions to common problems experienced in the development of large software systems – essentially, they embody a technique to foster reuse of software solutions [6]. Reuse can take many forms, and at different

Copyright 2004 Simulation Interoperability Standards Organization

levels of abstraction. These methods of reuse have been made possible by such features as language-supported encapsulation and functional interfaces. Aspects of reuse have been formalized into a pattern language that enables the description of object-oriented system behavior in uniform terms. The artifacts of using this formalism are design patterns. Design patterns have four essential elements:

1. **Pattern name**: a unique name for identification in a catalog
2. **Problem**: a clear definition of the problem the design pattern solves, including when to apply the pattern
3. **Solution**: a succinct description of the elements that make up the design, in terms of the pattern language primitives or other patterns, often including implemented examples of the solution
4. **Consequences**: the results of applying the pattern, and any tradeoffs in its use

Importantly, patterns must be described using a consistent format, with relationships to other patterns, and examples of their use. Without this consistency, reuse of the solutions is limited. A canonical example of a Design Pattern is the Model-View-Controller (MVC) pattern for describing user interface design [7]. This pattern describes a design where the underlying system (the model) is separate from the way to observe the model (the view), which is separate from the means for manipulating the system or its observation (the controller). This solution highlights the importance of decoupling major aspects of a system through encapsulation to reduce maintenance costs and encourage reuse. Design patterns span a wide range of problems and solutions, from object structure organization (Proxy Pattern), to low-level mechanisms for accessing elements in a set (Iterator Pattern), to patterns describing system-level designs, such as the MVC example given above. Any level of design throughout a software system can be described using design patterns.

2.2 Behavior Design Patterns

Based on the demonstrated usefulness of design patterns in traditional software engineering, we are adopting this methodology to develop behavior design patterns (BDPs)—abstract patterns useful in the creation of human behavior models (HBMs). Like design patterns, BDPs describe successful designs for agent behaviors that appear repeatedly in agent system designs. We expect BDPs to share many of the features of design patterns, and to be similarly defined, including a description of the problem it the pattern addresses, and a detailed description of the solution.

While much of the content of a BDP is similar to that of a typical design pattern, differences between typical software systems and the software architectures used to build high-fidelity human behavior models necessitate differences in the definition of BDPs. For example, the design pattern literature typically assumes certain features in languages that can be used to implement the design patterns, such as inheritance and composition in object-oriented design. Furthermore, traditional software engineering assumes the standard linear execution paradigm of von Neumann architectures, which is an implicit assumption about execution in existing design patterns. HBR architectures such Soar [8] and ACT-R [1] do not assume serial threads of control. Instead, they include features such as parallel (or at least pseudo-parallel) execution, associative memory retrieval, least commitment (the ability to delay run-time decisions until absolutely necessary), memory management via belief maintenance [9], and a combination of goal-directed and reactive, situated behavior. These differences enable the autonomy, flexibility, and taskability available in HBRs. Because these intelligent system architectures present a different computational paradigm than typical object-oriented software systems [10], these uniquely agent characteristics require a novel look at design formalisms.

Common architectural structures and processes may underlie agent behavior execution, such as beliefs, desires, and intentions [11], and processes of commitment and reconsideration to these structures [12-14]. However, despite this unifying view, the details of these processes will differ from one behavior architecture to another. Because different architectures make different assumptions about memory or execution structures, a design description must be abstract enough to accommodate many different approaches. However, it may be unavoidable that in some cases the implementation of a behavior design pattern into a HBR architecture is non-trivial, or incurs different costs with different architectures and implementations.

3. Patterns in Existing Behavior Systems

We argued above that patterns of human behavior can be observed in a range of tasks. This section discusses an analysis of an existing HBM, and the potential generalization of the patterns we observed to other applications and domains.

3.1 Analysis of TacAir-Soar

We studied an existing behavior system, TacAir-Soar [4, 5], as an initial source for understanding and constructing general behavior patterns. TacAir-Soar is a model of military aircraft pilots that can execute most of the missions performed by fixed wing aircraft. High-level decision-

Copyright 2004 Simulation Interoperability Standards Organization

making in TacAir-Soar is generally organized in the form of a hierarchical task decomposition; e.g., engaging an enemy requires being within range of a weapon, which requires maneuvering into position with respect to the enemy, which requires knowing where that enemy is, etc. This structure is meant to capture the problem solving and execution process a human pilot demonstrates. However, task decomposition does not necessarily reveal the similarities among different tasks. With this analysis, we reorganized the behaviors into a generalization (is-a) organization [15] that better suggested commonalities between behaviors.

A first challenge in this task was in deciding what should be called a behavior. Because there are many different ways we could have approached pattern extraction, we took a user-centered point-of-view and focused on tactical level decision-making, rather than low-level, behavior-engine-specific details. The tactical decision-making focus, allowed us to abstract away from many of the low-level behaviors. We grouped the remaining behaviors from TacAir-Soar into eight general categories:

· **Communications**: How and when to talk to other agents, and how to interpret incoming communications
· **Missions:** Mission-specific behaviors, such as for air-to-ground missions or air-to-air missions
· **Control**: How to direct other units to take action
· **Coordination**: How to work with other friendly agents
· **Flying:** How to fly a plane
· **Navigation**: How to decide where to go and how to get there
· **Situational Awareness**: How to manage the information in the environment
· **Action**: Atomic interactions with the simulation platform

This analysis demonstrated one of the fundamental limitations of design patterns (and traditional object oriented languages), mentioned previously. Many behaviors for autonomous systems are cross-cutting and cannot be cleanly, completely encapsulated from other agent behaviors. Situational awareness, for example, is a process that happens nearly continuously, and is informed by and informs other behaviors. Communication is another example of a crosscutting behavior. Certain aspects of communication—sensing incoming messages, parsing them, connecting them to the behavior contact, or constructing a message from the context and delivering it via a medium such as radio—occur in every communication exchange, at many points during the course of a mission. The context changes for message interpretation and generation, and the content of the messages changes accordingly, but the basic process is the same. The work on generic communication

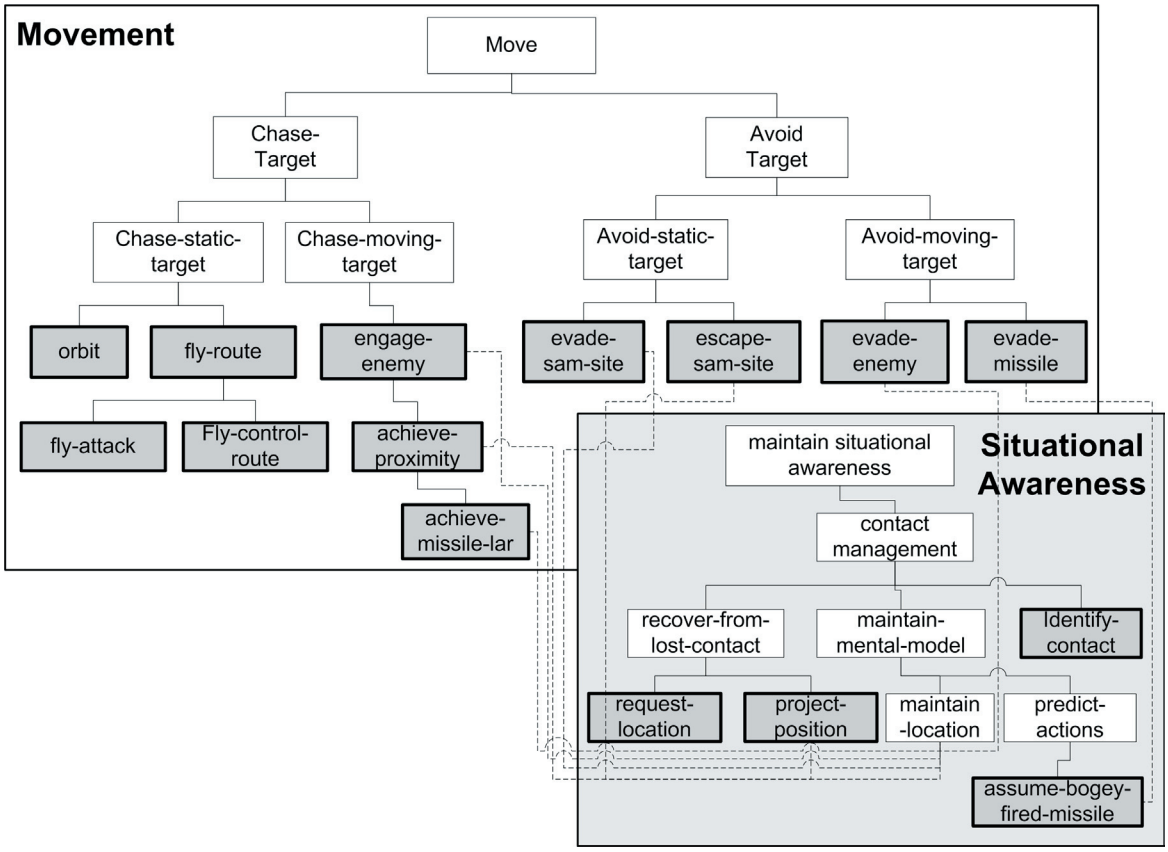

Figure 3.1: Example behavior taxonomy and class interdependencies

Copyright 2004 Simulation Interoperability Standards Organization

behaviors described by [16] does not explicitly call this a pattern, but their presentation of the process is a definite precursor to this work. The outstanding challenge is to determine how to incorporate elements that are more crosscutting than communications (aspects of which can be encapsulated straightforwardly), such as coordination, teamwork, and error handling. Cross-cutting issues are the subject of current research in software engineering [e.g., 17], and may provide some potential solutions.

Figure 3.1 illustrates a subset of TacAir-Soar's behavior taxonomy. The two large boxes represent some of the major categories of behaviors: Movement and Situational Awareness. The small shaded boxes represent behaviors taken directly from TacAir-Soar; the white boxes represent the taxonomic structure introduced in analysis. The solid lines reflect the is-a relationships between elements (*avoid-static-target* is a specialization of *avoid-target*). The dotted lines between boxes illustrate the data dependencies between behaviors, and highlight the crosscutting nature of, in this example, situational awareness.

A fairly simple example BDP derived from this TacAir-Soar analysis is shown in Figure 3.2. The Move-to-Target BDP captures regularities in a certain class of movement behaviors. For example, three behaviors typical in military domains are chasing an enemy, following a route, and maintaining formation in a group. Basic route following consists of moving from one point to another. Maintaining formation consists of keeping a position relative to a moving lead vehicle. Both of these behaviors are composed of even simpler building blocks such as changing speed and orientation, and sensing other objects in the environment (waypoints or vehicles), and, at a sufficient degree of abstraction, are all kinds of target chasing: for route following, one chases sequential stationary targets (waypoints); for formations, one chases a single moving target (the lead); in engaging an enemy, one achieves some range from the target in order to employ weapons. Furthermore, both aircraft and tanks perform these activities, albeit constrained and influenced by details about the different physical platforms and their capabilities, and the other agents and objects they sense in the environment. This BDP encapsulates each of these behavior components, describe how and when they are useful, and specifies how they can fit together with other modules to define more complex agent behavior.

The structure of the proposed solution is also given in the diagram. Here, we offer a simple hierarchical goal decomposition of what it means to move toward a target: changing speed, or changing orientation, which itself may include changing pitch, roll, or heading. Important to the definition of the pattern is that points of variation are identified and isolated. For example, movement in the ground domain is different than in the air domain due to the degrees of freedom allowed in each: pitch and roll orientations are not typically specified in a tank. By isolating these differences, and giving them a place in the pattern, the similarities can be exploited to cover a wide array of behaviors. Included in the definition of the pattern are a description of the problem it solves, and the consequences (positive and negative) of using the pattern. The pattern also identifies examples of use, and other patterns that

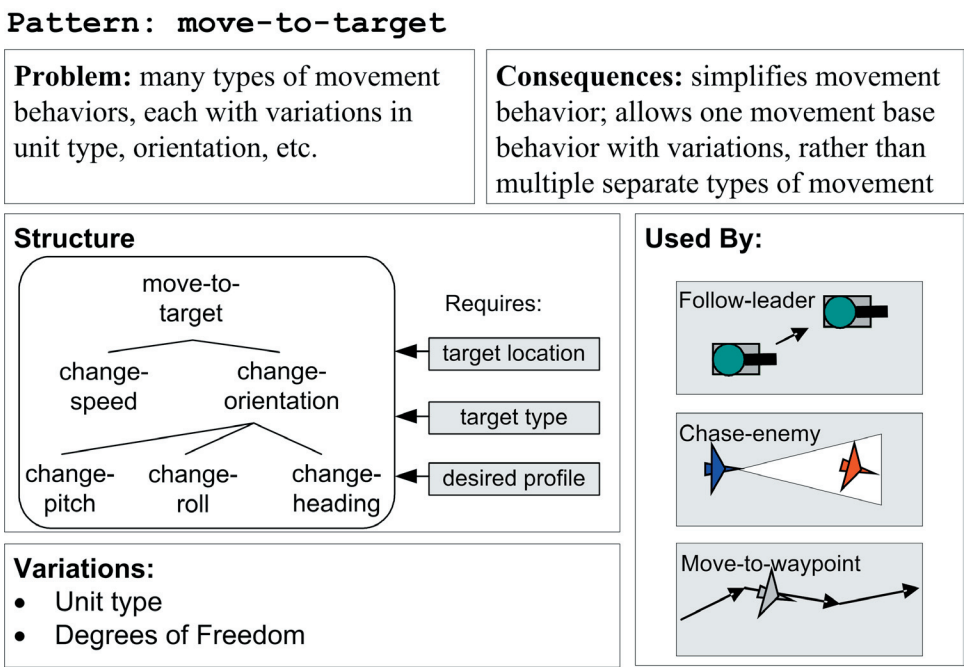

Figure 3.2 : The Move-to-Target Behavior Design Pattern

Copyright 2004 Simulation Interoperability Standards Organization

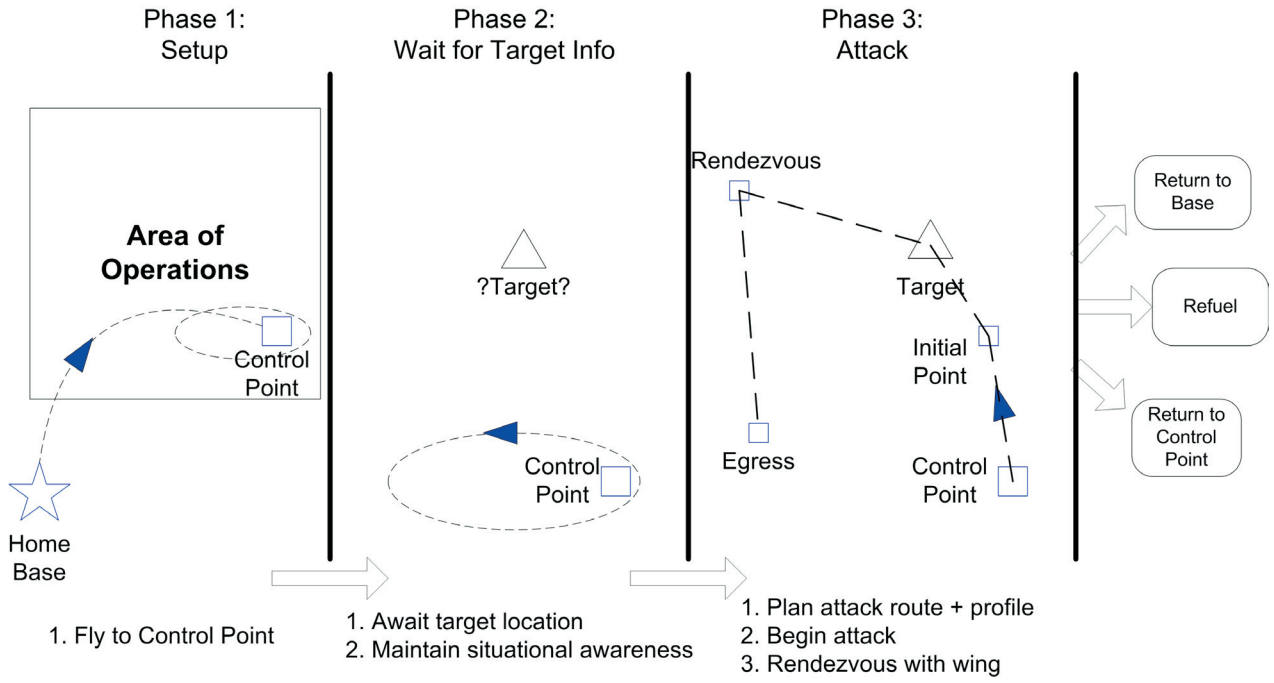

Figure 3.3: An overview of a generalized fixed-wing Ground Attack mission

might be involved in the use of this pattern.

3.2 Ground Attack Example

As mentioned previously, patterns can describe designs at all levels of a software system. In TacAir-Soar, a high-level behavior is the execution of an entire mission, which includes a multitude of goals and sub-goals, coordination with other entities, and maintaining situational awareness over a long time period. Our experience developing TacAir-Soar has shown many common features between different mission types, and has allowed us to abstract some of these features.

For example, Close Air Support and Strike missions share a great number of characteristics, including similar data requirements and phase structures. We have generalized these missions into a generic Ground Attack mission. Figure 3.3 presents an overview of this mission's execution, divided into three phases: (1) the aircraft takes off and flies to some control point, (2) awaits target information from the target controller, and, once received, (3) executes its attack run. (The mission planning, in Phase 1, is not shown in the figure). Following the execution of the attack, the agent has a number of explicit options it can consider for transitioning out of the BDP (including a recursive execution of the ground attack mission by going back to the control point). With some understanding of specific missions, it should not be difficult to see how this generic mission can be instantiated to describe the behavior of a particular strike aircraft executing a close-air support or strike mission.

The pattern for this generalized mission (shown in Figure 3.4) gives a state chart view of the Ground Attack behavior, reflecting the different phases and transitions between the phases. Thus, the state chart diagram provides a script-like representation of the behavior pattern, easy for a user to interpret. However, it is important to realize that the actual execution of the behavior includes reactivity and complex decision-making, resulting in rich, autonomous behavior, which is difficult to capture in the state chart diagram. In TacAir-Soar, the Ground Attack behavior is only one of many simultaneously active, executing behaviors (as indicated by the "Interacts With" box in Figure 3.4). The additional behaviors shown here involve maintaining and updating situational awareness, coordinating with other entities such as teammates and controllers, etc. Just as the Ground Attack BDP indicates transitions out of the behavior, the situational awareness behaviors can also trigger transitions not explicitly encoded in the BDP itself. For example, if the strike package encounters enemy aircraft, the situational awareness BDP may trigger a transition to intercept or evasion behaviors. These interacting behaviors may themselves be defined using BDPs (e.g., Target Identification Pattern, Situational Awareness Pattern; Coordination Pattern), where the Ground Attack pattern is defined from a combination of well-defined BDPs.

This example highlights the potential advantage of

Copyright 2004 Simulation Interoperability Standards Organization

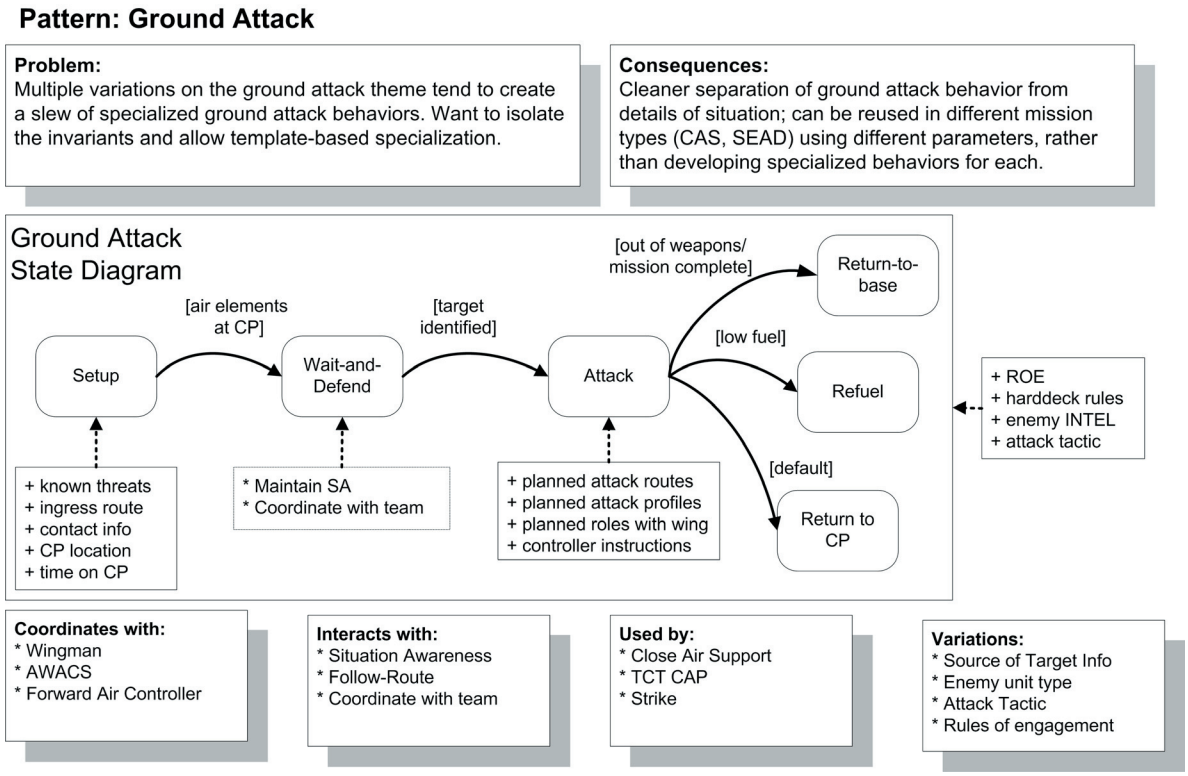

Figure 3.4: State-based view of the close-air support Behavior Design Pattern.

developing BDPs for HBRs. By identifying and specifying the general air-to-ground attack pattern, variations of this pattern could be easily developed from it. For example, the AC-130 performs air-to-ground attacks by circling a target, rather than by delivering a bomb on a target. Because the BDP captures the general pattern of ground attack, a knowledge engineer can focus on encoding the differences between the specific, desired behavior and the pattern. This is in contrast to current methodologies, where an engineer would either have to implement the behavior from scratch, or try and identify pieces from existing implementations of close-air support, strike, time-critical-targeting and possibly other behaviors.

3.3 BDP Notation

We have not yet settled on specific diagram formats and notations. As demonstrated in the Universal Modeling Language [18], multiple diagram forms are required to capture the full details of a complex software system. While we have attempted to use the UML in its various forms to diagram the behavior design pattern, it is clear that the standard forms of the UML, because they are based on the assumptions of a serial execution model, are not sufficient to cover the reactive, parallel models in many agent systems. For example, as drawn in Figure 3.4, they imply a serial, fixed-order execution, which is not at all the case in intelligent HBMs like TacAir-Soar.

The agent-oriented programming community has begun to address some of the limitations of the UML with respect to agent-based systems. There is some debate within that community whether the existing UML notation is sufficient for their purposes, or if additions are required, such as described in Agent UML (AUML) [19]. AUML introduces elements such as goals and multi-agent interaction through the UML's provided extensions, but so far these have been demonstrated on only simple agent systems, which exhibit few of the complexities that are unique to HBRs. Thus far, AUML is not sufficient for knowledge-intensive HBRs. Thus, we are still exploring useful means to convey the precise patterns of behavior and execution in these complex behavior systems, and, in particular, the complexities of reactive decision making and parallel execution.

4. Relationship to Other Work

There is a growing body of literature that describes design patterns specific to agent-based systems [19, 20]. However, it has focused largely on simple software agents, and on multi-agent interactions. Predicting and controlling these interactions is important, and some of these patterns will be useful in developing muti-agent HBMs; however, BDPs for high-fidelity HBMs must also provide a design formalism that captures the complexities of the agents themselves. [21] offers a description of design patterns for

Copyright 2004 Simulation Interoperability Standards Organization

CGFs, but focuses on the simulation architecture rather than on the behaviors of high-fidelity intelligent entities in the simulation. We adopt the view that intelligent systems represent a fundamentally different paradigm from traditional object-oriented software engineering, and even from the simple agents described in the mobile software agents literature. While some of the behaviors of intelligent systems may be shared with other types of applications (such as iteration over a data set), there will be a large class of problems and solutions unique to intelligent systems and human behavior models.

Outside of design patterns, a few areas of research are worth mentioning. Generic Tasks (GTs) [22] describe problem-solving methods for intelligent systems, and the requirements of those methods in terms of inputs, outputs, and expected knowledge forms. GTs focus primarily on weak methods such as the estimation of actions in a state, matching hypotheses to data, and hierarchical classification. This work is certainly a precursor to behavior design patterns, but because much of our work deals with entity-level behavior in a simulation environment, we have so far examined BDPs that are grounded in entity-level tasks such as movement and communication, and higher-order constructs such as missions. Also related is the behavior taxonomy literature, most recently illustrated by Feinberg [3], who created a taxonomy explicitly meant for building synthetic forces. While this taxonomy is far-reaching and highly detailed, its behavior definitions seem too generally defined help to derive patterns.

5. Conclusion and Future Work

We have presented initial work adapting traditional design pattern methodology to the practice of engineering human behavior models. We analyzed an existing fixed-wing behavior system, TacAir-Soar, and identified a set of abstract behavior categories, some of which map well to behavior design patterns. For example, we identified existing mission-level behaviors for fixed-wing air-to-ground attacks and generalized them into an example behavior design pattern. Also, we generalized several movement behaviors into a design pattern that encapsulates and exploits their similarities while isolating their differences.

The TacAir-Soar analysis showed that categories (and a few examples) of patterns can be found in at least one knowledge-intensive behavior system. Our assumption is that these patterns will transfer to other parts and extensions of TacAir-Soar, other fixed-wing air HBRs, and, importantly, to other HBR domains. We have begun to use these patterns in other CGF systems, including RWA and Dismounted Infantry[23]. Optimistically, this analysis suggests that, given a pattern design language and descriptions of existing behavior patterns in that language, much can be re-applied and reused for new applications.

We are currently in the process of defining a language for specifying behavior design patterns for human behavior models. As mentioned previously, the primary challenge is designing a language that can express the essence of reactive and goal-directed agent behavior. While typical software systems have a very serial, easily described execution process, which can be illustrated with traditional flow charts or state charts, most HBR systems do not execute in this manner. So, while it is possible to describe possible threads of agent execution in a serial way, with serial diagrams, in actuality, a behavior may be interrupted at any point, and goal contexts may switch. One of the challenges of good design is to develop abstractions that can begin to communicate the actual complexities and capabilities of autonomous systems.

Behavior design patterns will reach their maximum potential when agent development tools are developed that can translate the diagrams in a pattern into source code in a selected agent language. (Tools such as Prometheus [24] have begun to address this for some types of UML-like diagram forms, but primarily at a multi-agent system level, rather than for detailed single-agent design.) Until then, much work remains in formalizing a pattern language for describing BDPs for complex human behavior models, and then describing known patterns. The traditional design patterns literature has improved software engineering practice as a whole simply by providing a catalog of well-described solutions to common problems, allowing even junior engineers take advantage of tested design solutions. As the HBR community matures, and we collectively learn more about building systems of this nature, we expect similar catalogs to improve the practice of building less expensive, more robust and reliable HBRs.

6. Acknowledgements

This work was supported by the Defense Advanced Projects Research Agency, contract number DAAH01-03-C-R242.

7. References

[1] Anderson, J. and C. Lebiere, The Atomic Components of Thought. 1998: Lawrence Erlbaum.
[2] Newell, A., Unified Theories of Cognition. 1990, Cambridge, Massachusetts: Harvard University Press.
[3] Fineberg, M.L., A Comprehensive Taxonomy of Human Behaviors for Synthetic Forces. 1995, Institute for Defense Analyses: Alexandria, VA.
[4] Tambe, M., W.L. Johnson, R.M. Jones, F.M. Koss,

Copyright 2004 Simulation Interoperability Standards Organization

J.E. Laird, P.S. Rosenbloom, and K.B. Schwamb, Intelligent Agents for Interactive Simulation Environments. AI Magazine, 1995. 16(1): p. 15-39.
[5] Jones, R.M., J.E. Laird, P.E. Nielsen, K.J. Coulter, P.G. Kenny, and F.V. Koss, Automated Intelligent Pilots for Combat Flight Simulation. AI Magazine, 1999. 20(1): p. 27-42.
[6] Gamma, E., R. Helm, R. Johnson, and J. Vlissides, Design Patterns: Elements of Reusable Object-Oriented Software. 1994, Boston, MA: Addison-Wesley.
[7] Buschmann, F., R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal, Pattern-Oriented Software Architecture. Vol. 1. 1996, West Sussex, England: John Wiley and Sons.
[8] Laird, J.E., A. Newell, and P.S. Rosenbloom, Soar: An architecture for general intelligence. Artificial Intelligence, 1987. 33(3): p. 1-64.
[9] Wray, R.E. and J.E. Laird, An architectural approach to consistency in hierarchical execution. Journal of Artificial Intelligence Research, 2003. 19: p. 355-398.
[10] Abadi, M. and L. Cardelli, A Theory of Objects. Monographs in Computer Science. 1996, Berlin: Springer Verlag.
[11] Bratman, M., Intentions, Plans, and Practical Reason. 1987, Cambridge, MA: Harvard University Press.
[12] Jones, R.M. and R.E. Wray. Design principles for heavy intelligent agents. in Autonomous Agents and Multiagent Systems. 2003. Melbourne, Australia.
[13] Schut, M. and M. Wooldridge. Principles of Intention Reconsideration. in Fifth International Conference on Autonomous Agents. 2001. Montreal, Canada.
[14] Wooldridge, M.J., Reasoning about Rational Agents. 2000, Cambridge, MA: MIT Press.
[15] Brachman, R., What ISA Is and Isn't: An Analysis of Taxonomic Links in Semantic Networks. IEEE Computer, 1983. 16(10): p. 30-36.
[16] Wray, R.E., J.C. Beisaw, R.M. Jones, F.V. Koss, P.E. Nielsen, and G.E. Taylor. General, maintainable, extensible communications for computer generated forces. in Eleventh Conference on Computer Generated Forces and Behavioral Representation. 2002. Orlando, Florida: Institute for Simulation and Training.
[17] Kiczales, G., J. Lamping, A. Mendhekar, C. Maeda, C.V. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-oriented programming. in European Conference on Object-Oriented Programming (ECOOP). 1997. Finland: Springer-Verlag.
[18] Booch, G., I. Jacobson, and J. Rumbaugh, The Unified Modeling Language User Guide. 1998, Boston, MA: Addison-Wesley.
[19] Bauer, B., J. Muller, and J. Odell, Agent UML: A Formalism for Specifying Multiagent Interaction, in Agent-Oriented Software Engineering. 2001, Springer-Verlag: Berlin.
[20] Aridor, Y. and D. Lange. Agent Design Patterns: Elements of Agent Application Design. in Autonomous Agents. 1998: ACM Press.
[21] Courtemanche, A.J. Design Patterns for Computer Generated Forces. in 8th Conference on Computer Generated Forces and Human Behavior Representation. 1999. Orlando, FL.
[22] Chandrasekaran, B., Generic tasks in knowledge-based reasoning: High-level building blocks for expert systems design. IEEE Expert, 1986. 1(3): p. 23-30.
[23] Taylor, G., F. Koss, and P.E. Nielsen. Special Operations Forces IFORS. in 10th Conference on Computer Generated Forces and Behavioral Representation. 2001. Norfolk, VA.
[24] Padgham, L. and M. Winikoff. Prometheus: A Methodology for Developing Intelligent Agents. in Autonomous Agents and Multi-Agent Systems. 2002. Bologna, Italy.

Author Biographies

GLENN TAYLOR is a Research Scientist at Soar Technology, Inc. His R&D activities include agent-based systems, human behavior modeling, and natural language processing. He received his BS in computer science in 1994, and his MS in computer science and engineering in 1996, both from the University of Michigan.

ROBERT E. WRAY, III, PHD, is a Senior Scientist at Soar Technology, Inc. His research and development experience includes work in agent-based systems, machine learning, knowledge representation and ontology. Previously, he was assistant professor of computer science at Middle Tennessee State University. He received a Ph.D. in computer science and engineering from the University of Michigan in 1998.

Copyright 2004 Simulation Interoperability Standards Organization