Behavioral Threads: Coordinated Synthetic Team Behavior for Small Group Training

Webb Stacy, Ph.D., John Colonna-Romano, and Mark Weston
Aptima, Inc.
Woburn, MA
wstacy@aptima.com, jcromano@aptima.com, mweston@aptima.com

Tim Roberts
US Army RDECOM Simulation Training Technology Center
Orlando, FL
tim.e.roberts@us.army.mil

ABSTRACT

The military's requirements for training and mission rehearsal have undergone a recent transformation. Increasingly the focus is on accommodating small groups of trainees in urban environments with adversaries that are less identifiable than ever. This often means that there are fewer training support personnel available, and though they may be knowledgeable about techniques, tactics, and procedures, they may not be experts in operating the equipment and software necessary to provide the training.

There must be an effective way to provide small synthetic teams of opposing forces, supporting forces, or neutral forces that can perform coordinated, realistic actions. Current solutions either provide limited capabilities to a single instructor or provide richer capabilities but require multiple instructors.

We are working on a solution that enables a single instructor to control complex, coordinated actions of a small synthetic team. To accomplish this, we characterize their actions as a set of behavioral scripts, or *plays*, by analogy with sports simulation games. To author the plays, we have extended concepts from the OneSAF Behavior Composer using the computer science idea of a *thread*, which is a portion of a computer program that can run independently from other portions. By analogy, a *behavioral thread* is a portion of a behavioral script that can run independently from other portions.

In this paper, we discuss correspondences between computer science threads and behavioral threads, with special attention to thread coordination mechanisms, thread visualization, debugging, and performance. We describe a behavioral-threads-based play executing in a modern training environment, and conclude by speculating on how the science of teams might contribute to threads in computer science.

ABOUT THE AUTHORS

Webb Stacy, Ph.D., is Vice President of Technology at Aptima. He oversees Aptima's current and future technology portfolios. His focus is the intersection of software and computer science with the science, modeling, and measurement of warfighters as individuals and as teams. He has extensive experience in the development of mission critical software, and holds a Ph.D. in Cognitive Science from the State University of New York at Buffalo and a B.A. in Psychology from the University of Michigan.

Tim Roberts is a Science and Technology Manager of Dismounted Soldier Technologies at the Research Development and Engineering Command - Simulation and Training Technology Center (RDECOM-STTC). His research interests include using commercial gaming systems for virtual training, virtual locomotion, haptic interfaces, hand-held mobile technologies, path planning and control systems, multi-modal interfaces and robotic systems. He received a BS in Electrical Engineering and a MSEE in Control Systems and Robotics from the University of Central Florida.

John ("JCR") Colonna-Romano is the Lead Software Architect at Aptima, Inc. Mr. Colonna-Romano has over 25 years of experience in software system architecture, systems engineering and software engineering. He is interested in system architecture, simulation based training systems, computer game technologies and the application of technology to solve complex problems. Mr. Colonna-Romano has co-authored two books on distributed systems middleware architecture. Mr. Colonna-Romano started his career at the U. S. Department of Defense. Mr. Colonna-Romano received a B.S. and an M.S in Computer Science from Stevens Institute of Technology.

Mark Weston specializes in designing and building interfaces for applications that analyze and optimize complex human organizations in both technical and non-technical domains. His degree included coursework on database applications, artificial intelligence, network protocols, and large scale software design. He is currently interested in text analysis and computational linguistics. Mr. Weston received his B.S. in Computer Science from Union College.

Behavioral Threads: Coordinated Synthetic Team Behavior for Small Group Training

Webb Stacy, Ph.D., John Colonna-Romano, and Mark Weston
Aptima, Inc.
Woburn, MA
wstacy@aptima.com, jcromano@aptima.com, mweston@aptima.com

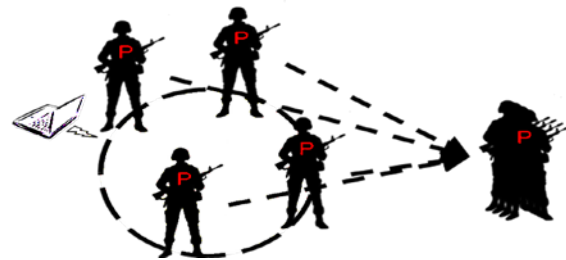
Tim Roberts
US Army RDECOM Simulation Training Technology Center
Orlando, FL
tim.e.roberts@us.army.mil

BACKGROUND

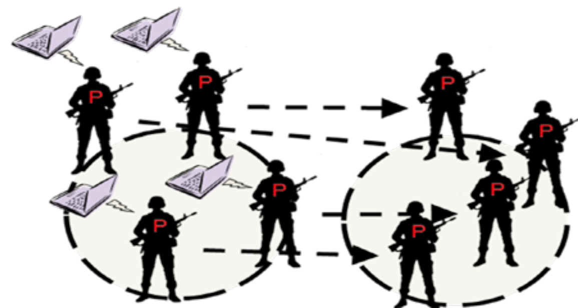
The military's requirements for training and mission rehearsal have undergone a recent transformation. Increasingly the focus is on accommodating small groups of trainees in urban environments with adversaries that are less identifiable than ever. Often there are fewer support personnel available, and though they may be knowledgeable about techniques, tactics, and procedures, they may not be experts in operating the equipment and software necessary to provide the training.

Unfortunately, current simulation-based training environments are not optimal for this application. For example, the "sweet spot" of OneSAF, the Army's pre-eminent simulation-based training environment, is mid- and high-level conventional warfare-based company-and-above level exercises. Traditionally, synthetic exercises are focused on large force-on-force problems, and may involve heavyweight Semi-Automated Force (SAF) scenarios requiring large numbers of *pucksters* (human controllers of non-trainee synthetic characters). In today's environments, though, the number of personnel available to serve as pucksters is limited, often to one, and those that are available may not have the specialized training necessary to operate SAF applications.

The choices frequently boil down to either one puckster attempting to control the individuals in the team, often with unacceptable results (Figure 1a), or multiple pucksters coordinating with each other in a resource-intensive force-on-force configuration (Figure 1b). Control of realistic, coordinated actions of small synthetic teams by a single operator is not yet possible for platoon- and squad-size exercises, yet this control is exactly what is required.



(a) One Puckster Attempts to Move Multiple Individual Avatars, with Unintended Effects.



(b) Multiple Pucksters Move Multiple Individual Avatars, with Increased Training Resource Demand.

Figure 1. Current Options for Small Team Control. *Note:* Laptop icon represents a human puckster.

An improved solution will allow intuitive, real-time control of small groups of synthetic forces, will offload the human puckster from the details of the coordination of group behaviors, and will allow the puckster to make rapid adjustments to team behavior as circumstances dictate. One promising approach is similar to the control that is possible in sports simulation games where the user selects and executes a pre-planned play. Military exercises are considerably more complex than football games, but the analogy remains useful. Military trainers will select and execute a military "play," which represents the behavior of an opposing force (OPFOR)

team, the behavior of supporting forces, or the behavior of civilian groups. This is similar to the Playbook concept in the domain of unmanned vehicle control (Miller *et al.*, 2005.)

For authoring team plays, the requirements go beyond simply scripting the behaviors of the individual team members. The team scripting mechanism must be able to accommodate recent developments in team science regarding coordination and resource sharing. This paper is about an extension to current practice, *behavioral threads*, that meets these requirements.

A variety of approaches have been used to formally specify individual behavior, ranging from checklists to task decompositions to flowcharts to scripting and programming languages. We know from team science, however, that team behavior involves interdependencies among team members (Orasanu & Salas, 1993) and that there is generally a need to specify coordinated sharing of resources among team members. (Serfaty, Entin, & Deckert, 1994; MacMillan, Entin, & Serfaty, 2004).
There is an area of computer science that directly addresses similar issues, that of multithreaded programming. A *thread of execution* is analogous to the activities performed by one team member, and a *computer program* that contains the threads is analogous to a team play.

In this paper, we explore the consequences of borrowing concepts, practices, and lessons learned from the study of threads in computer science. We will show that mechanisms for coordination and resource sharing among threads are quite useful for describing coordination and resource sharing among team members. We call this collection of ideas *behavioral threads*.

A Team Example

To give us a simple example with which to work, consider a *bounding* play. Bounding is a procedure that gets a team across an area while under fire. Suppose we have a team that has two individuals, Romeo and Sierra. The objective of the play is to get both of them—that is, the entire team—across a street while they are being fired upon.

Figure 2 shows the play. Romeo and Sierra split up, and then Romeo begins crossing the street at the same time that Sierra begins providing cover fire. When Romeo is across the street, the roles are reversed, and Romeo begins providing cover fire for Sierra, who then crosses the street. When both team members have crossed the street, they rejoin each other, and the team goes on its way.

In Figure 2, individual activities are shown with green boxes near the threadlike horizontal lines. Of particular interest for our discussion are the methods for coordinating the individuals' behaviors: decision points are shown with rounded brown boxes, and points of inter-individual coordination are shown with horizontal gray bars.

We use this example extensively in the rest of the paper, whose organization is as follows. We first discuss the computer science notion of threads, including context and coordination mechanisms, and describe how those ideas apply to team behavior.

We then look at the practice of using threads, including common errors and useful development tools, after which we examine the application of threads to a

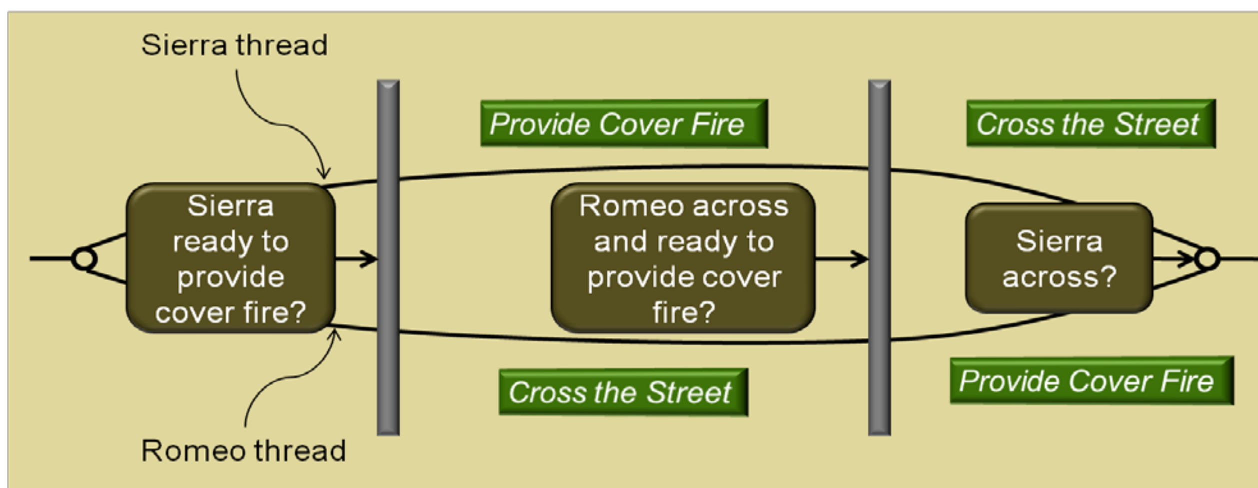


Figure 2. Simple Bounding Play

simulator-based training project that involves small team scripting that builds on the OneSAF Behavior Composer and its infrastructure. We conclude by speculating about the future of behavioral threads, including potential contributions that behavioral threads can make to computer science.

THREADS IN COMPUTER SCIENCE AND TEAM SCIENCE

In computer science, a *program* is a set of instructions that will let a computer perform a task. On simple computer systems, the tasks are performed in a single-tasking environment—the computer works on one task until it is done, then it works on the next, and so on. This is a *single-threaded* execution environment.
On most modern computer systems, though, the computer can work on more than one task at a time. They are able to do this because they have sophisticated multitasking operating systems and, often, multiple cores or CPUs. These are *multi-threaded* execution environments.

Though there are other kinds of multitasking in these environments, in this paper we focus on the mechanisms that allow for multiple threads of execution within a single program. Multithreading in these environments doesn't come automatically—the authors of multithreaded programs must explicitly write them to be multithreaded (cf. Butenhof, 1997).

A common Web browser, for example, may have a thread that handles user input, so that it remains responsive to user requests, and may concurrently have other threads that render the Web page, that send and receive requests over the Web, that manage browsing tabs, and so on. Managing all of these tasks in parallel provides a better overall user experience.

Two key characteristics of such multithreaded programs will help illuminate the analogy with teams. These are the notions of *context* and of *coordination*.

Context

A thread is rarely written as a single block of instructions. Instead, it is often subdivided into discrete subtasks variously called subroutines, functions, or methods. Those subtasks may be subdivided into other subtasks, and so on. As a result, the subtasks are arranged in a hierarchy.

It is also sensible to think about multi-individual team plays as being arranged in a hierarchy. Figure 3 shows a sample subtask hierarchy for a behavioral thread that will be executing the ***Provide Cover Fire*** subtask in the bounding play of Figure 2. It says that ***Provide Cover Fire*** invokes the ***Ready***, ***Aim***, and ***Fire*** subtasks, that the ***Ready*** subtask invokes the ***Load Ammo*** and ***Raise Weapon*** subtasks, and so on. The ellipses indicate that the hierarchy continues until it reaches a *primitive* (which could be an existing subtask in an existing library or could be a subtask that has been directly translated into instructions for the computer.) Subtasks that are not primitive can be called *composite*. Note that the task hierarchy is simply a diagram showing which behaviors invoke which others; it does not display any decision logic. For example, in Figure 3 the ***Fire a Burst/Pause*** and ***Assess*** subtasks will likely be repeated under the ***Fire*** subtask, but the subtask hierarchy does not show this.

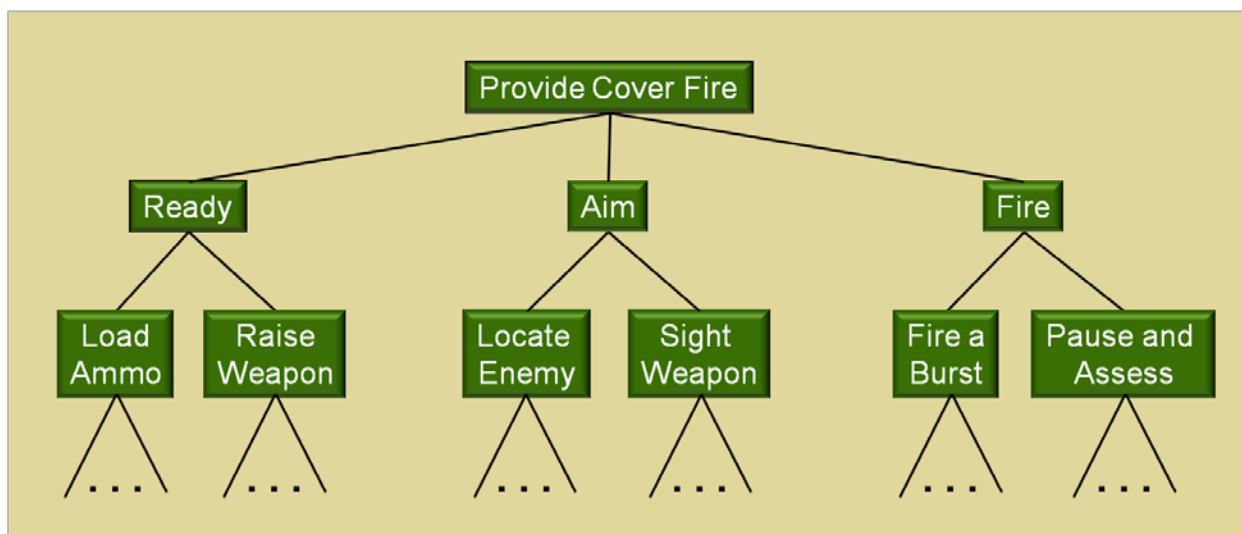


Figure 3. Sample Task Hierarchy of *Provide Cover Fire* Subtask of Bounding Play.

This framework is used explicitly by the OneSAF Objective System (OOS) Behavior Composer (Garcia & Griffith, 2005), and is the basis for the behavioral threads application we discuss in this paper. In the OOS Behavior Composer, primitive behaviors exist in a library, and other behaviors are composite. *Execution* of the task hierarchy is a systematic visiting (and sometimes revisiting) of the subtasks in the hierarchy. If execution is stopped at any point, it is "in" one of the subtasks. Since that particular subtask was invoked from higher-level subtasks, it is also "in" those. In general, there is a stack of subtasks that exactly locate the point of execution. This stack is called the thread's *context*. Specifically for behavioral threads, we call this the *play context.*

Coordination

The subtask hierarchy in a Web browser might exist in one thread—say, the user-response thread—but it will wind up coordinating its activities with other threads like the one responsible for network communications and the one responsible for rendering pages. Similarly, the threads in the bounding play need to coordinate about when to change roles. But what do we really mean by coordination, and how does it occur?
There are many implementations of program threads, and many variations on the coordination mechanisms, but they boil down to two kinds of functions: resource management and sequence dependency enforcement.

***Resource management*.**
When multiple threads or individuals share resources, there needs to be a way to avoid the confusion that can arise from resource conflicts. For computer programs, the resources involved are often locations in shared memory or access to input/output devices. For team behavior descriptions, these could be any shared and limited resource—an ammunition store, a pair of binoculars, an oxygen tank, or even a canteen full of water.

Mechanisms for resource sharing in computer science include a *mutex*—short for 'mutual exclusion'—and its multi-resource cousin the *semaphore*. For both, the strategy is to provide a token that signals when a resource is being used and to enforce a protocol that requires acquiring the token before taking the resource and releasing the token when done with the resource.
The sign on the washroom door on an airplane is an everyday example of a mutex. When a passenger enters the washroom, locking the door automatically changes the sign to "In Use," signaling to other passengers that the resource is unavailable. Leaving the washroom changes the sign to "Vacant," signaling the resource's availability.

Though we know of no system that provides mutexes or semaphores for team resource management, we believe they are useful for describing team behavior. For example, imagine that a team's ammunition is carried in a pack that allows access by only one individual at a time. If two synthetic team members need to get ammunition at the same time, a mutex will provide a realistic mechanism to emulate what live team members would do. Without mutexes, synthetic team members might access the pack at the same time or might prevent each other from accessing the pack at all.

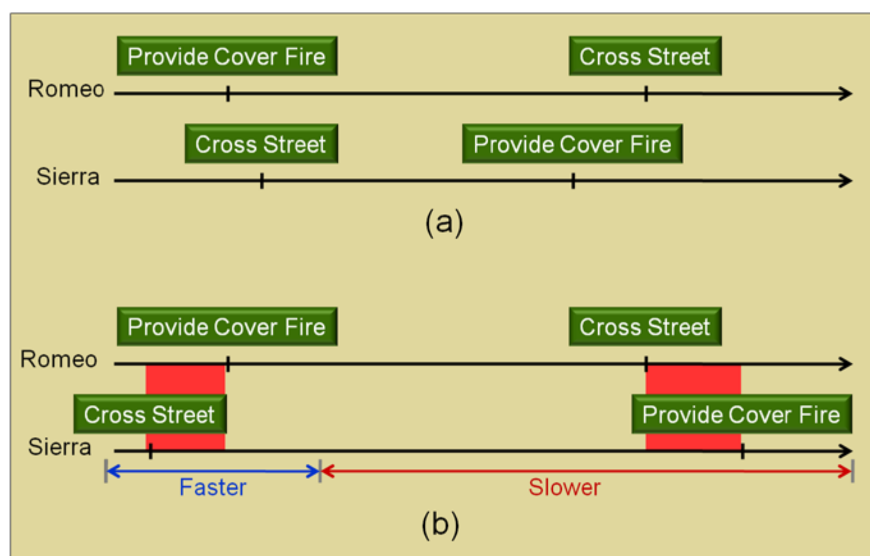


Figure 4. Uncontrolled Subtask Sequencing. (a) Expected Speed of Execution; (b) Actual Speed of Execution.

Sequence Dependency Enforcement.

There are times when tasks in one thread must be completed before tasks in different threads begin. This is easy to arrange when there is only a single thread of execution or a single individual involved—just sequence the later task after the earlier one. When there are multiple threads or multiple individuals, though, tasks are by definition being performed in parallel. If the threads or individuals don't coordinate somehow, then there is no control over when the subtasks in one thread occur relative to subtasks in another thread. In fact, if there are factors that change the speed of execution of a thread, subsequent execution of the tasks might result in the subtasks being performed in a different order.

For example, Figure 4a shows the expected execution order of the subtasks from the bounding play described in Figure 2. In Figure 4b, however, Sierra has been able to execute his tasks in his thread more quickly in the early part of the play, and more slowly later. As a result, he begins crossing the street before there is cover fire, and doesn't provide cover fire until after Romeo has begun crossing the street. The resulting dangerous uncovered times are indicated in red. Clearly, the street crossing subtask has a sequence dependency on the covering fire subtask.

Two mechanisms that allow threads to respect sequence dependencies are *condition variables* and *barriers*. Condition variables allow a thread to wait for some condition to be true or some event to occur. The bounding play needs condition variables that require that one thread's street crossing task waits until the other thread's covering fire task begins. This allows for correct execution of the bounding play independent of the speed of execution of either thread.

A barrier is a mechanism that requires all threads to arrive before any of them can proceed. This is most often used for specifying stages or phases of execution. For example, imagine a slightly different version of the bounding play where Romeo and Sierra are subteams of three individuals rather than single individuals. The idea is that one subteam of three individuals provides cover fire for the other subteam of three individuals. Each individual still gets their own thread, however, so in this variant there are 6 threads. A good way to ensure that an entire subteam has crossed the street is with a barrier—no individual on the subteam should continue until all individuals on the subteam have arrived.

Existing simulation systems often have mechanisms similar to condition variables and barriers, but because they aren't accompanied by resource management mechanisms, they run the risk of not being *atomic*, that is, of being interleaved with interfering activities from other threads or individuals.

For example, in the bounding play in Figure 2, the condition for switching roles—the condition asking if Romeo is across the street and ready to provide covering fire for Sierra—can be expressed as a condition variable. It is necessary that both parts of this condition be checked at once, however.

If checking whether Romeo has crossed the street is separate from checking if Romeo is ready to fire, one part of the condition could change before the other is checked. For example, suppose Romeo has crossed the street but notices that he dropped his communication device while he was crossing. In order to save time, he loads his weapon and makes himself ready to provide cover fire while he goes back out into the street to retrieve it. If Sierra checks whether Romeo has crossed the street—there is a moment when he has done so—and afterwards checks whether Sierra is ready to provide cover fire—there is also a moment when this is true, though not simultaneously with having crossed the street—Sierra will mistakenly begin crossing the street.

Though this example may seem somewhat artificial, in practice the lack of atomicity is a major source of incorrect and nondeterministic thread behavior, especially as the number of threads increases.

Alternative Approaches to Parallelism

There is another approach to synthetic team programming that is worth comparing: teams of autonomous agents. As it turns out, there is an analogous technique in computer science, too: message passing.

In parallel computing, message passing is a major alternative to threads, and in fact the leading technology involved in today's high-performance supercomputers. (Grupp et al., 1996; Yang & Guo, 2005; Meuer et. al., 2009). Typically, the parts of a program that execute in parallel execute on different machines that not only have their own CPUs, but their own memory, disk drives, and other resources. For this reason, in the message passing paradigm the coordination of sequence dependencies plays a much larger role than does resource management, though this can come at the expense of an increased demand on the communications subsystem.

One way to think of this is that parallel execution is performed by a set of distributed, mostly independent

entities. On the behavioral side, this is also a good way to describe distributed teams of synthetic agents, about which there has been substantial research, especially in the areas of collaborative planning (Lesveque, Cohen, & Nunes, 1990), joint intention (Grosz & Kraus, 1996), and domain-independent rules for how teams should work together (Tambe, 1997; Schurr et. al., 2005.)

However, much of that research has a different focus than behavioral threads. First, the aim is to discover general rules and mechanisms that will allow teams of autonomous agents to accomplish complex goals. To the extent the research is successful, the agents will not need to be scripted to accomplish goals they have never addressed before. This is a fascinating and difficult quest, but our goals are more modest: to find convenient, workable formalisms to express scripted team behavior. Though it is our hope that the behavioral threads formalism applies to all team plays, it will still be necessary to author the threads. Second, the distributed nature of the agents is often a requirement, not a choice; cooperating agents in general need to be able to reside at different locations on a network. Although we are not convinced that being distributed would preclude the use of behavioral threads, it is worth noting that on the computer science side, threads by themselves are generally not sufficient to cover distributed applications. Finally, message passing among agents often enables agent-based solutions to scale up to large problems, but may not be as convenient as behavioral threads for smaller, more controlled synthetic team situations. Just as both threaded and message-passing approaches to parallel computer programming have their places, so do behavioral threads and distributed agents.

THREADS IN PRACTICE

So far, we have been discussing the theory of threads. We now turn to a discussion of threads in practice, including common authoring errors and helpful authoring tools. We conclude the section by describing how we have applied behavioral threads to the scripting of synthetic team behavior in the system for small team training described in the introduction.

Errors

Programming systems that involve parallelism can often be confusing; concurrency at any but the tiniest of scales is hard to understand. A multithreaded program or play, even if correct in design, can easily display hard-to-reproduce and obscure bugs while it is being developed.

If programmers of multithreaded computer programs make characteristic programming errors, it stands to reason that authoring of multi-individual team plays are vulnerable in similar ways. Lu, Park, Seo, & Zhou (2008), in a first-of-its kind study, categorized the concurrency-related programming errors found in several open-source multithreaded computer programs, including MySQL, Apache, Mozilla, and OpenOffice. Table 1 gives an overview of the proportion of bugs they found.

<table>
<tr><th colspan="2">Type</th><th>Pct.</th></tr>
<tr><td colspan="2">Deadlock bugs</td><td>29%</td></tr>
<tr><td rowspan="3">Non-deadlock bugs</td><td>Atomicity</td><td>47%</td></tr>
<tr><td>Order</td><td>22%</td></tr>
<tr><td>Other</td><td>2%</td></tr>
</table>

Table 1. Results of Concurrency Bug Survey. From Lu, Park, Seo, & Zhou (2008).

Deadlock bugs occur when each of two threads have locks (like mutexes or semaphores) that the other needs to continue, so neither will ever be able to release the lock they already have. These are often, but not always, resource management bugs. The two main categories of non-deadlock bugs, atomicity and order, are often sequence dependency bugs, but could also be resource management bugs. Note that these three categories account for a whopping 98% of all issues identified.

Behavioral threads are very likely subject to the same kinds of issues, and this study implies that both resource management and sequence dependency bugs are important to watch for and prevent.

What sorts of tools might we employ to go about doing so? We now turn to a discussion of threads debugging and visualization tools and how they would work with behavioral threads.

Tools

As the behavioral threads approach becomes widely used, there will be a strong need for tools that will assist in providing a view into executing scripts and in finding bugs of the sort described in the last section. There are really three kinds of tools of interest:

- Error detectors, which find errors automatically;
- Debuggers, which find errors in the executing play with the human author; and

- Visualization tools, which provide authors with situation awareness of the play as it executes.

Automatic error detectors can identify many kinds of potential problems in running programs, most often deadlocks and their relatives, most often by analyzing resource locks like mutexes (Intel, 2009; Sun, 2009.) Agarwal & Stoner (2006) were able to go beyond that and develop algorithms that detect deadlocks in running programs that use mutexes, semaphores, and condition variables. Adaptation of these sorts of tools, applied to the authoring of team plays, will greatly assist in eliminating obscure and difficult-to-find bugs that are otherwise inevitable in concurrent applications.

What about the bugs that the error detectors don't find? For them, there are debuggers. Probably the most well-known is TotalView (TotalView, 2009), which was originally developed to debug parallel applications on the BBN Butterfly, but is now in widespread use in many multithreaded environments. It allows users to run parallel programs, to stop one, several, or all threads at selected places, inspect variables, and otherwise interact with the state of threads at various stages of execution. As with the automatic error detectors, adaptation of this kind of functionality will help ensure that the library of behavioral-threads-based team plays is reliable and stable.

There are also tools that allow the visualization of executing threads. One such set, Visual Threads (HP, 2009), automatically collects and summarizes information about thread status (running, waiting, and so on) and automatically detects potential resource sharing and synchronization problems. It then displays that data in a variety of visual formats, allowing programmers to understand the dynamics of the running program and its potential bugs.

Even more interesting is a suite designed by Zhao & Stasko (1995) called Polka. Polka provides visualizations for thread state, context, and history, and can show the status of mutexes, semaphores, condition variables, and barriers. Figure 5 shows what the visualization of the ***Cover Fire*** subtask might look like in Polka if it were being executed by a three-person subteam. The context itself is exactly as shown in Figure 3, and the "location" of each of the three individuals (that is, of each of the three behavioral threads) is shown with a different token. In this case, one individual is in ***Load Ammo***, one is in ***Fire a Burst,*** and one is in ***Pause and Assess***. As the program runs and the behavioral threads change context, the tokens will move to new boxes; at any point, the behavioral threads—the individuals—comprising the subteam can be paused to inspect the activities in which they are currently engaging. It is easy to imagine that this would be a useful aid to help authors understand the workings of team plays they are authoring.

Now that we have had a brief taste of tools that might in the future be adapted for authors using behavioral threads, we turn to an actual application that is using them.

A Case Study

We are using behavioral threads in a program called Virtual Puckster (VP). The goal of VP is to develop an application that will allow intuitive, real-time control of small groups of synthetic forces that will relieve the human puckster from attending to the details coordinating group behaviors by allowing them to select team behavior plays from a library. VP is being implemented in a way that will allow it to work with most modern gaming and simulation environments.

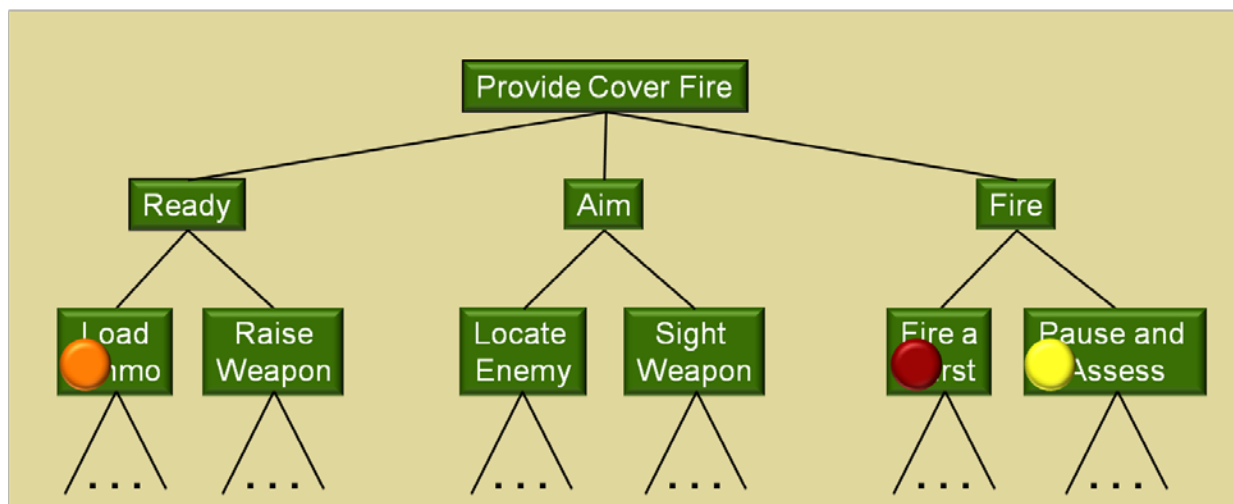


Figure 5. Visualization of Behavioral Threads for Subteam Version of Bounding Play.

VP allows a single human to provide effective control of a synthetic team, alleviating the problems identified in the introduction in Figure 1. The improved situation can be seen in Figure 6.

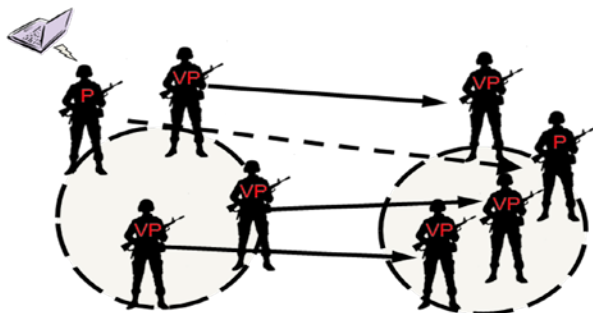


Figure 6. Improved Movement of Small Synthetic Teams: One Human Puckster Moves Entire Team. *Note:* Laptop icon represents the human puckster.

The VP system itself is shown in Figure 7. It consists of a graphical user interface (GUI), Group Behavior Engine (GBE) and a Group Behavior Play Library. The VP GUI is used to control the VP system, gather group control inputs from the human puckster and give the user team level information. The GBE generates coordinated group behaviors based on a specified play, the current conditions of the simulation environment and the control inputs from the human puckster.

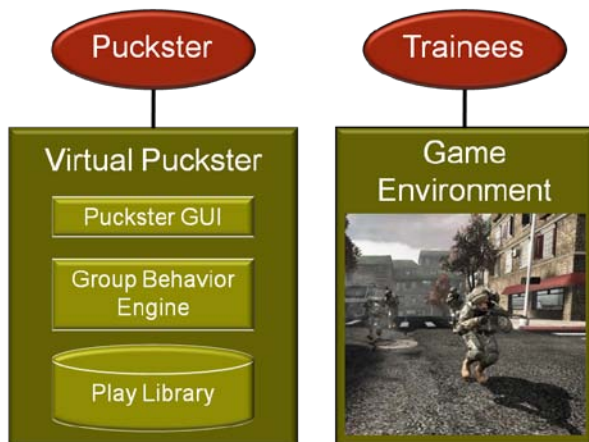


Figure 7. Schematic of Virtual Puckster.

Plays in the Library are created using the behavioral threads concepts discussed in this paper. In doing so, we use the OneSAF behavior composer to construct the basic play and let it generate the resulting XML. We then supplement that XML as necessary with threads coordination mechanisms. Figure 8 shows an actual Virtual Puckster play, slightly more involved than Figure 2's bounding play, expressed in terms very similar to those used by the OneSAF Behavior Composer. The synthetic team is specified to move in formation until it is fired upon. Then it switches to a ***React to Fire*** subplay where the team divides up into two subteams, each team searches for cover and then performs a coordinated bounding move towards the attacker, alternating supplying cover fire and moving. If the attacker is killed, the entire team reforms and resumes the team move play.

The play uses a set of primitive actions (such as move and fire) which are supported by the game simulation environment. VP supports a stack-based context for each unit or sub-unit—that is, for each behavioral thread—in a play.

VP is using the concepts and tools described in this paper and is building a library of plays. Beyond that library (and the tool extensions it will entail), we can also ask if concepts from team science might make contributions to threads in computer science.

WHAT TEAM SCIENCE HAS TO SAY ABOUT MULTITHREADED PROGRAMS

We have seen that multithreaded computer programs have interesting similarities with multi-individual teams, and that the authoring of a play for synthetic teams benefits from borrowing concepts, tools, and best practices from the field. We now turn to the question of how multithreaded programs might benefit from what is known about human teams.

We believe there are two sets of answers. First, human teams show several synchronization mechanisms not currently available with threads programming infrastructure. Humans participating in teams can anticipate the actions of their teammates (MacMillan, Entin, & Serfaty, 2004) and, when problems or "bugs" in team behavior arise, they are often self-correcting (Entin, Weil, See, & Serfaty, 2005).

For example, it is rare that humans are stuck for prolonged periods in deadlocks—it is an everyday occurrence for humans, even strangers, to realize the situation and to fix it. Suppose two people want to walk through a door that can only accommodate one at a time. Each person believes it is polite to let others go first, so, for a brief moment, they are in deadlock—neither wants to walk through the door until the other has gone. This might well stall an automated system, but the situation is almost always resolved quickly by humans. Multithreaded programs thus might benefit from synchronization mechanisms that anticipate the actions of other threads and that are self-correcting.

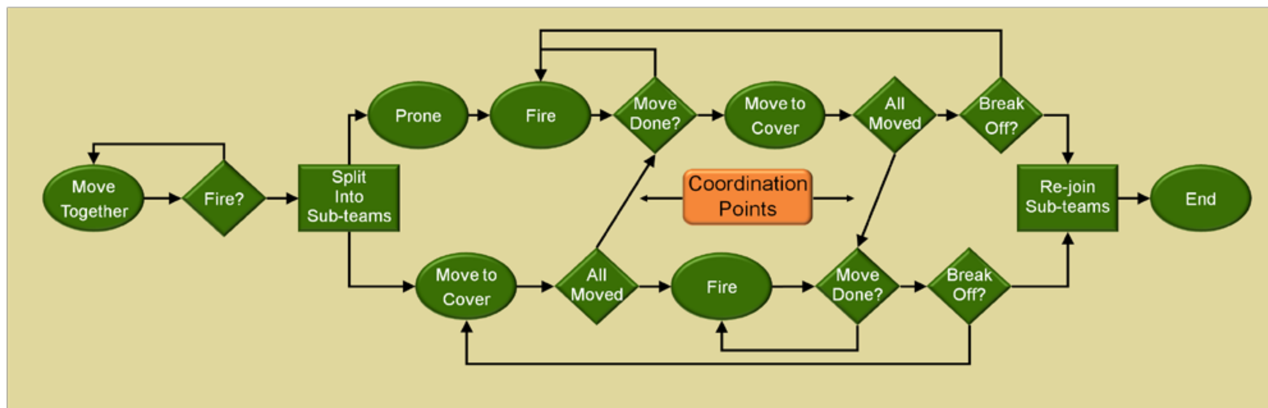


Figure 8. Bounding Play Expressed in OOS Behavior Composer-like Terms.

The second set of ideas comes from the fact that most human teams have leaders who help organize, coordinate, and steer their teams to accomplish their missions in uncertain environments. This is almost never true for multithreaded programs. We believe it would be worth exploring (and perhaps creating infrastructure to support) the idea of a *metathread* as an analog to the team leader. Like the team leader, it could monitor the performance and accomplishments of the other threads, detect, diagnose, and repair interthread performance problems, and otherwise help the threads deal with unexpected obstacles to performing their mission.

We believe that there is a rich, beneficial opportunity for computer science and team science to inform each other, and we hope this paper constitutes a small step in that direction.

ACKNOWLEDGEMENTS

We gratefully acknowledge the US Army RDECOM Simulation Technology Training Center for their support and advice of this research, much of which was performed under contract W91CRB-07-C-5007. Thanks also to Daniel Serfaty, Jean MacMillan, Jared Freeman, Georgiy Levchuk, and Nathan Schurr, all of Aptima, for their discussions about early versions of these concepts.

REFERENCES

Agarwal, R., & Stoner, S.D. (2006). Run-time detection of potential deadlocks for programs with locks, semaphores, and condition variables. *Proceedings of the 2006 Workshop on Parallel and Distributed Programs: Testing and Debugging*. 51-60.

Butenhof, D. B. (1997). *Programming with POSIX Threads*. Reading, MA: Addison-Wesley Professional Computing.

DelSignore, J. (2003). TotalView on BlueGene/L. *BlueGene/L Workshop*, Reno, NV.

Entin, E. E., Weil, S. A., See, K. A., & Serfaty, D. (2005). *Understanding team adaptation via team communication*. Paper presented at the Human Systems Integration Symposium, Arlington, VA.

Garcia, C.J., & Griffith, T.W. (2005). A composable behavior modeling system for rapidly constructing human behaviors. *Proceedings of the 2005 Interservice/Industry Training, Simulation and Education Conference*, Session S-4.

Grosz, B., & Kraus, S. (1996). Collaborative plans for complex group actions. *Artificial Intelligence*, **86**, 269-358.

Grupp, W., Lusk, E., Doss, M. & Skjellum, A. (1996). A high performance portable implementation of the Message Passing Interface standard. *Parallel Computing*, **22**(6), 789-828.

Hewlett-Packard (2009). *Visual Threads*. Available online at http://h30097.www3.hp.com/dtk/visualthreads_ov.html.

Intel, Inc. (2009). *Thread Checker*. Available online at http://software.intel.com/en-us/intel-thread-checker/.

Lesveque, H.J., Cohen, P.R., & Nunes, J.H.T (1990). On acting together. *Proceedings of the 1990 Conference*

of the American Association of Artificial Intelligence, 94-99.

Lu, S., Park, S. Seo, E., & Zhou, Y. (2008). Learning from mistakes—a comprehensive study on real world bug characteristics. *Proceedings of the 2008 Conference on Architectural Support for Programming Languages and Operating Systems*, 329-339.

MacMillan, J., Entin, E. E., & Serfaty, D. (2004). Communication overhead: The hidden cost of team cognition. In E. Salas & S. M. Fiore (Eds.), *Team cognition: Process and performance at the inter and intra-individual level.* Washington, DC: American Psychological Association.

Miller, C., Funk, H., Wu, P., Goldman, R., Meisner, J., Chapman, M. (2005). The playbook approach to adaptive automation, In *Proceedings of the Human Factors and Ergonomics Society's 49th Annual Meeting*, Orlando, FL.

Meuer, H., Strohmaier, E., Dongarra, J., & Simon, H. (2009). Top500 Supercomputer Sites Project, available online at http://www.top500.org.

Orasanu, J. & Salas, E. (1993). Team decision-making in complex environments. In G. Klein, J. Orasanu, R. Calderwood, & C.E. Zsambok (Eds.), *Decision Making in Action: Models and Methods*. Norwood, NJ: Ablex.

Schurr, N., Okamoto, S., Maheswaran, R.T., Scerri, P., & Tambe, M. (2005). Evolution of a teamwork model. In R. Sun (Ed.), *Cognition and Multi-Agent Interactions: From Cognitive Modeling to Social Simulation.* 307-327. Cambridge: Cambridge University Press.

Serfaty, D. S., Entin, E. E., & Deckert, J.C. (1994). *Implicit coordination in command teams.* In A. H. Levis & I. S. Levis (Eds.), Science of command and control: Part III coping with change (Aip information systems) (87-94).

Sun Microsystems, Inc. (2009). *Thread Analyzer*. Available online at http://developers.sun.com/sunstudio/downloads/ssx/tha/.

Tambe, M. (1997). Agent architecture for flexible, practical teamwork. *National Conference on AI (AAAI-97)*, 22-28.

TotalView Technologies (2009). *TotalView Debugger*. Available online at http://software.intel.com/en-us/intel-thread-checker/.

Yang, L.T., & Guo, M. (2005). *High Performance Computing: Paradigm and Infrastructure.* Hoboken, NJ: Wiley & Sons, Inc.