Making Behavior Modeling Accessible to Non-Programmers: Challenges and Solutions

Bart Presnell, Ryan Houlette, Dan Fu
Stottler Henke Associates, Inc.
San Mateo, CA 94404
{bpresnell, houlette, fu}@stottlerhenke.com

ABSTRACT

To create the most effective possible simulations, domain experts must be able to author, monitor, and modify the behavior of simulated agents. Current computational models of autonomous agent behavior are not adequate in this regard. Simple hard-coded models still predominate in many areas, while the most capable and realistic behavior modeling architectures – such as SOAR and ACT-R – are also generally the most difficult to work with, requiring trained programmers to develop and update behavior models. We contend that to enable domain experts without programming expertise to author sophisticated agent behaviors, there are two main challenges that must be addressed: condition authoring and behavior analysis.

Complex conditions – such as the preconditions for a step in a plan – are a necessary part of almost any behavior model, but specifying these conditions is not easy. Text-based authoring is an efficient way to enter the information, but the required syntax can be overwhelming to the non-programmer. Visual authoring methods, by contrast, are better able to guide non-programmers through the authoring process but tend to be much more time-consuming and laborious. The second major challenge is enabling non-programmers to analyze the runtime behavior of the models they create. Behavior models of any significant complexity require multiple "test and fix" iterations to uncover authoring mistakes. Modeling tools must therefore provide data visualizations that permit the non-programmer to see both global structure and specific details in the large volume of data generated by test runs of the behavior model. In addition, authoring tools must easily allow the creation of unit-test-like scenarios.

We have spent the last three years developing an adversary behavior modeling tool for the Air Force, during which time we have attempted to address both of these challenges. We will present lessons learned and suggested best practices as well as areas for future work.

ABOUT THE AUTHORS

Bart Presnell is a software engineer at Stottler Henke Associates. He holds an M.S in Computer Science from the Georgia Institute of Technology. Prior to joining Stottler Henke, Mr. Presnell worked for seven years as a software engineer for leading game development studios. He is currently working to develop an autonomous planning system to be used in videogames and interactive simulations.

Ryan Houlette is a project manager and lead software engineer at Stottler Henke Associates. He holds an M.S. in Computer Science (Artificial Intelligence) from Stanford University. He has participated in the development of a wide range of AI systems, with a particular focus on autonomous agents and intelligent interfaces. Mr. Houlette is lead architect of the SimBionic behavior modeling tool and product manager for the SimVentive simulation construction toolkit. He is also an editor for the AI Game Programming Wisdom book series.

Dan Fu is a group manager at Stottler Henke Associates. He joined nine years ago and has worked on a number of artificial intelligence (AI) systems including AI authoring tools, wargaming toolsets, immersive training systems, and AI for simulations. Dr. Fu was principal investigator on the project that created the SimBionic AI middleware, which enables users to graphically author entity behavior for a simulation or videogame. Dr. Fu holds a B.S. from Cornell University and a Ph.D. from the University of Chicago, both in computer science.

Making Behavior Modeling Accessible to Non-Programmers: Challenges and Solutions

Bart Presnell, Ryan Houlette, Dan Fu
Stottler Henke Associates, Inc.
San Mateo, CA 94404
{bpresnell, houlette, fu}@stottlerhenke.com

INTRODUCTION

To efficiently create the most effective possible simulations, subject matter experts must be able to author, monitor, and modify the behavior of simulated agents. Current computational models of autonomous agent behavior are not adequate in this regard. Simple hard-coded models still predominate in many areas, while the most capable and cognitively realistic behavior modeling architectures – such as SOAR and ACT-R – are also generally the most difficult to work with, requiring trained programmers to develop and update behavior models.

A new generation of tools is attempting to make the process of constructing behavior models more accessible to non-programmers. Examples include COTS tools such as AI.implant and SimBionic as well as government-funded efforts such as the OneSAF Behavior Composer and the Office of Naval Research's Affordable Human Behavior Models. Though these tools do offer some improvements over the status quo, we believe that significant work remains before behavior modeling becomes truly user-friendly. For the past three years, we have been exploring various approaches to simplifying the model authoring process during the course of developing an agent behavior modeling framework. In this paper, we discuss some specific authoring challenges that we have identified and examine various techniques for mitigating them.

Background

To ground our discussion, we will first give a brief overview of our behavior modeling framework (dubbed Madcap), which served as a testbed for developing the ideas presented here. Madcap was designed to generate dynamic adversary behavior in an operational-level air operations simulation. The framework comprises a behavior model authoring tool and a model execution engine that controls entities within an attached simulation. A two-tiered agent architecture allows multiple levels of autonomy, with a hierarchical task network (HTN) deliberative planner providing goal-driven behavior and a finite state machine (FSM)-based execution layer driving purely reactive or even scripted agents.

Using the Madcap model authoring tool, the user specifies the various components of an agent behavior model. The agent's knowledge about the world is defined using a tree-based editor that displays the hierarchy of types and instances (and their respective attributes). The HTNs that constitute the plan library and the FSMs used by the execution layer are both specified by sketching flowchart-like sequences of actions on a graphical "canvas." The overarching design goal of the authoring tool was to allow the user to conceptualize the behavior model primarily in terms of his or her subject matter expertise rather than in terms of a programming language. In the following sections, we discuss two major obstacles to achieving this goal that we encountered.

CONDITIONAL EXPRESSIONS

At the most basic level, building a behavior model can be thought of as defining a sequence of actions to be carried out by an agent. Any nontrivial model will define multiple possible courses of action, which entails some decision-making on the part of the agent to determine which course to follow. It is the job of the model author to specify the conditions that guide this decision-making process. The exact form of these conditions will vary depending on the agent framework, but they can be broadly characterized as "*a logical expression that evaluates some aspect of the simulation or behavior model state to determine whether a particular course of action pertains*." For example, in the Madcap system the *defend-base-with-aircraft* plan has the precondition "There must be a friendly fighter jet that is within 50 miles of the base and is not already tasked." Written as a formal logical expression that the system knows how to process, this becomes

```
(exists ?aircraft
        (AND (is-type ?aircraft fighter)
             (allegiance ?aircraft friendly)
             (< (distance ?aircraft.location
```

base.location) 50)
(tasking ?aircraft none)))

While the plain English version of this condition is easily understandable, the logical translation of it is daunting to anyone without a background in computer science. Even if the would-be model builder is able to puzzle out its meaning, this is a far cry from being able to generate such an expression on one's own. (Note that while this example is particular to our framework, analogous conditional expressions can be found lurking in nearly any behavior modeling system.) The cryptic nature of conditional expressions thus represents a significant roadblock to the user-friendly authoring of behavior models.

Considering the problem of conditions more closely, we see that there are at least two separate difficulties. First, the syntax is unfamiliar to users who are not logicians or programmers. As a result, a user who knows what condition he or she wishes to specify may be unable to express it in the format demanded by the behavior model – in other words, translating from the English version to the formal logical version is unintuitive. Second, even if we ignore the logical syntax, deriving the fundamental structure of the condition still requires a certain level of procedural thinking that is not natural to many users. The author must first identify each of the clauses that comprise the condition. While clauses that map directly to the underlying simulation (for example, "the distance from the aircraft to the base is less than 50 miles) are relatively easy to identify, untrained authors tend to omit "infrastructural" clauses (such as "aircraft is of type fighter") that would be common sense to a human but that the computational behavior model needs explicitly stated. Once the set of constituent clauses has been gathered, the author must then determine how to structure them into a single statement using words such as "and," "or," and "not." Simple conjunctions or disjunctions of clauses are generally easy for users to grasp, but more complex conditions that involve nested sub-clauses (for example, "*(OR (AND x y) (AND y z))*") quickly become confusing. Conditions involving universal and existential quantifiers can be tricky even for programmers.

Approaches to Condition Authoring

We next consider several approaches to condition authoring that attempt to address the abovementioned difficulties. The initial version of our authoring tool provided a text field with auto-completion (see Figure 1), and the user simply typed the desired conditional expression. This approach required the user to formulate the entire formalized logical expression "on the fly," with proper nesting of parentheses for sub-clauses. This proved daunting even for knowledgeable

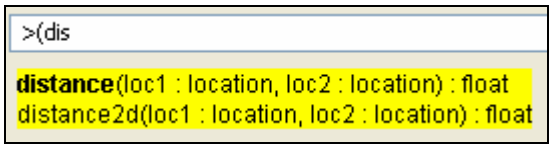

Figure 1. Text Condition with Auto-Complete

engineers. The flat text representation of the condition gave no insight into its logical structure. While auto-completion did help avoid mistakes when filling in the names of simulation entities and attributes, it did not provide adequate assistance to a user who was unfamiliar with the basic expression syntax. We concluded that pure text-based condition authoring was too much of a "blank slate" for non-programmers, who need more guidance.

As a result, we implemented a tree-based editor for conditional expressions (see Figure 2), where each sub-clause in the expression was represented as a sub-tree with the operator at the root. Clauses could be added to

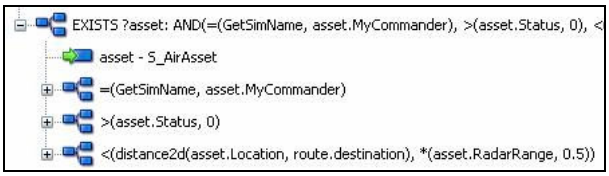

Figure 2. Condition Tree

or deleted from the expression tree using the context menu or toolbar. The tree editor mitigated some of the syntax difficulties for novice users by breaking down the logical expression into more manageable sub-clauses while clearly depicting the expression's overall nested structure. Using the tree, it is difficult to build a syntactically incorrect expression (though semantically nonsensical expressions are clearly still possible). It was, however, extremely tedious to use, forcing the user to build even the tiniest expression by recursively constructing sub-trees of variables and operators. In retrospect, it could also have been improved by using more English-like language in place of the traditional logical operators in the tree – for instance, "*asset.Status is greater than 0*" instead of "*(>(asset.Status 0))*".

Both of the above approaches are closely tied to the actual textual representation of the condition. We also examined a more graphical approach in which the logical expression is depicted as a directed graph (see

Figure 3), where logical operators and their arguments appear as nodes with arrows showing the logical flow

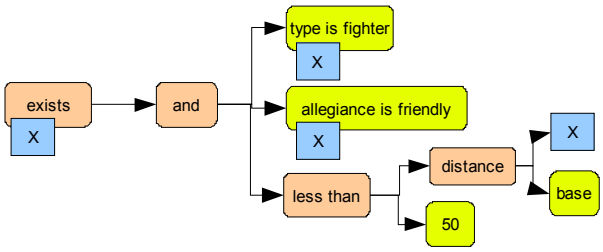

Figure 3. Graphical Condition Authoring

between them. The author builds a condition by dragging operators from a palette and dropping them onto the partially-constructed graph. The goal was to retain the tree's clear presentation of logical structure while making it less onerous to work with. The graph view does afford greater ease of use, enabling a more freeform authoring style where the user effectively "sketches" the desired condition. Ultimately, though, we felt that the usability improvement was outweighed by a less intuitive visual representation, which makes it hard to take in the entire expression at a glance. We thus decided to keep our existing tree editor. Further research in this area might produce other expression visualizations that are more immediately intuitive.

All of the approaches to condition authoring that we have described so far are fundamentally *descriptive*: that is, the user must explicitly define each attribute of the simulation state that the behavior model should test to determine if the condition holds. As a consequence, they all face the difficult challenge of empowering naïve users to construct logical expressions. An alternative approach that has the potential to largely avoid this problem is *authoring by demonstration*, where the user illustrates the desired condition in the context of the simulation itself. The behavior modeling system then automatically extracts the conditional expression from the simulation state at the time of the demonstration. The resulting condition is displayed to the user, who has the opportunity to edit it (using one of the previously-discussed methods). For example, if the user wished to specify the *defend-base-with-aircraft* condition via demonstration, he or she would run the simulation under manual control. When a base came under attack, the user would select an idle friendly fighter that was within 50 miles of the base and order it to attack the incoming enemy. The model would take a snapshot of the state of the world at the time that order was given and extract the relevant condition.

This approach has the considerable advantage of placing the condition-authoring process on familiar ground for the user, allowing him to act directly from subject-matter expertise instead of requiring an intermediary stage of introspection and logical codification (which often results in incompletely-specified conditions). It also largely (but not entirely) insulates the author from the underlying logical syntax and structure. While promising in concept, the authoring-by-demonstration approach does however pose its own significant challenges. To demonstrate a desired condition, the user must first set up an appropriate exemplar situation that provides the needed learning opportunity (i.e., a base under attack with a nearby untasked friendly fighter). This can be time-consuming, particularly when multiple examples are needed to clarify a complex condition. Practice is also required to define concise, clear examples. In addition, the problem of automatically extracting conditions from simulation snapshots can be quite difficult, especially if there are many simulation variables. The resultant conditions may be overly verbose, containing many extra clauses that test irrelevant aspects of simulation state. The user must prune these clauses in the post-processing step.

MODEL DEBUGGING

While improving the authoring of conditions will greatly reduce the difficulty of creating behavior models, anyone who has ever built a complex system knows that things never work correctly the first time. This intuition was borne out in the use of the Madcap model editor. Although the model authoring was fairly time-consuming, the bulk of model development time was actually spent in the debugging process. While the Madcap execution engine did supply logging facilities to help with model debugging, much of the debugging had to be done by software engineers instead of the model authors. From this experience, we discovered two key tools that are needed to help authors debug their models: unit tests and extensive logging with sophisticated data visualization.

Unit Testing

Unit tests are a very popular software engineering paradigm. They allow software engineers to set up tests for each component of a software product. By having these tests, the software engineer is able to easily test each component of the project separately rather than trying to force the entire project into states that test each underling component. In addition, unit test are usually run in an automated fashion which greatly

speeds up the testing process. This automated testing enables development paradigms that emphasize continuous refactoring, such as extreme programming. The unit tests allow programmers to change components and then quickly verify that the changes have not broken the required inputs and outputs.

Both the ability to directly test individual components and to quickly and thoroughly retest components to support refactoring are very valuable in the development of behavior models. Behavior models tend to be complex hierarchical systems, and discovering flaws in the model by merely looking at the results of simulation runs is very difficult. In debugging Madcap behaviors we often had to break the model into small chunks, separately checking condition functions, selected actions, and effect functions. Unfortunately, we had to manually edit the behavior model to test each component. Having a unit test suite to automatically allow testing of each component would have made this testing process much easier. In addition, the development of Madcap revealed that behavior model development was similar to extreme programming. We started with very simple behaviors and situations and slowly built upon these. During this process, we frequently discovered that we needed an entirely new type of behavior. Adding this behavior often required significant refactoring, which took a significant amount of time because of the need to go back and verify existing behaviors. The use of automated unit tests would have greatly reduced this time.

Unit tests would be of great benefit to behavior model development, but for any authoring tool to support effective unit testing the behavior model must provide two features. First, there must be a clear separation between the model execution engine and the simulated world. In the Madcap agent architecture, the agent receives updates from the simulated world, which are stored in an internal knowledge base. The agent then makes behavior choices based entirely on the information stored in the knowledge base. Because of this architecture, we were able to create test conditions by using an initialization file to load information into each agent's knowledge base without having to actually run the simulation. Being able to create a partial world state with no connection to the simulation has two key advantages. First, it allows the author to easily set up any situation that needs to be tested. This is a huge benefit for the author because it allows more thorough testing and validation of the model. The author can concentrate on determining which world states the model must handle, as opposed to expending a great deal of effort trying to force the simulation into the needed test states. Second, it allows the authoring tool to save snapshots of world states from runs of the simulation regardless of whether the simulation natively supports this. To save a state, the tool can either save out an agent's internal representation of the world or query the simulation interface for the current state of the world. For future testing this information can either be loaded into the agent's internal representation or be used to supply a stub simulation interface with the needed information to respond to agent queries. Regardless of the technique used to store world state, it is very useful for the debugging process. It allows the author to study the state and test model changes much more efficiently because they no longer need to wait for the simulation to run to the desired state – they can jump instantly to it.

The second feature the model architecture must support for effective unit testing is having each authorable component be a public functional object. This means the component must be accessible and the component cannot store state within itself. The first requirement specifies that the component can be reached and tested by an external test framework. This is not to imply that behaviors must make all of their internal representation public. If a behavior object has, for example, a precondition that can be authored, the behavior object must simply provide some way to externally test this precondition. Whether the behavior object has a method to retrieve the precondition function object itself or a method to retrieve the result of the precondition test is unimportant, as long as there is some way for an object outside of the behavior object to get the result of the component test. The second requirement – that the component be stateless – is a bit trickier for many behavior models, particularly finite state machines. This requirement is needed because we perform our unit tests on static states. To have the result of these unit tests be consistent with the results during actual simulation runs, the result of any behavior component must be based on purely the current state and not the entire history of states. Our solution for finite state machines is to move the state information from the finite state machine component to the global state. This has two benefits. First, it forces the author to explicitly set this information in test cases. Second, it separates the fairly large static description of the behavior – which in the case of a finite state machine is the transition tables – from the small amount of information needed to update a particular instance – for finite state machines, just the current state.. By making this separation, many agents can share the same behavior description, rather than each agent having their own copy. In very large simulations, this could represent a significant memory savings.

As described above, unit testing does place some significant requirements on the underlying behavior architecture. Based on our experience with Madcap, however, we believe that the benefits are worthwhile. Unit testing can help make the model authoring process more accessible to non-programmers by significantly streamlining the iterative development process and granting the author more visibility into and control over the process.

Debug Logging

While unit tests make the debugging process much simpler by allowing the author to easily test smaller parts of the behavior model, there will still be unexpected behavior in the course of running the full simulation. To have subject matter experts reasonably debug unexpected behaviors an authoring tool must support a very sophisticated level of debug logging.

The initial Madcap authoring tool had a fairly extensive set of logging facilities. It produced logs of each agent's selected behaviors and goals throughout the simulation run. In addition, the search process that agents went through to construct the plan was extensively logged. The state of the world at the beginning of the plan search was recorded, and the path through the hierarchical task network was also recorded. However, all of this information was not sufficient to allow non-software engineers to debug the behavior models. There were two main flaws in the Madcap logging system. First, the log of agent actions in the simulation, which was displayed in an intuitive timeline graph, was detailed enough to alert the author that the desired behavior was not performed, but it did not provide enough information to allow the author to determine what had caused the incorrect behavior to be selected. Second, our subsequent attempts to log more of the behavior model decision process resulted in an surfeit of information – enough to produce a memory overflow in many text editors – that when converted into a graphical display was simply too dense to be understandable. While the volume of information generated can perhaps be attributed to the exponential search process of a planning algorithm, as we look to simulate larger and longer scenarios, the number of decisions generated by even a simple reactive system will become similarly voluminous. We therefore propose some tools to help filter unnecessary data and some visualization methods that can be used to display a large amount of model execution data while still providing a useful level of detail.

One essential feature is the ability for the user to enable or disable logging on a per-behavior component basis. From our experience with Madcap, simply logging all information will be overwhelming for any user. A better way to attack the problem is to use a high-level overview of the behaviors executed by each agent during a simulation run to determine when an error occurs, and then add more detailed logging to very specific parts of the behavior model to pinpoint the problem. Authors should be able to select whether components log their input parameters, outputs, or both. The ability to log the output of arbitrary user-specified functions allows the author to annotate the log. In addition, the author should be able to specify time windows or conditions (based on behavior components' input parameters) to limit when data is logged. With these facilities, the author should be able to log precisely the needed information for specific trouble areas without drowning in extraneous information.

A second extremely useful logging tool is an explanation facility that provides the rationale for the selection of executed behaviors as well as the reasons why other behaviors were *not* selected. This facility poses several challenges. We are once again faced with a potentially huge amount of data that must be stored and displayed. In addition, there must be a selection process that determines which agent decisions need explanations. Most computational behavior models are mechanical processes that consider a large number of possible actions that would not be considered for common-sense reasons by a human. The decision-making algorithms do – if they are functioning correctly – generally reject these nonsensical actions, but this still represents a significant fraction of processing time. Ideally we would like the explanation facility to avoid explaining these uninteresting actions. The best way to develop the needed filtering process is through machine learning, but this is work we did not attempt in Madcap.

The final issue that we discuss here is the question of how to display voluminous log data in an intuitive way. A number of display methods were considered and attempted in the development of Madcap. We will review some these below.

The first method to display the log data was a formatted text display. While simple, this approach was fairly effective as it allowed for a large amount of detail to be provided. In addition, it was fairly easy to navigate through the information using text search. As a result, we used text-based log displays frequently in Madcap. The biggest disadvantage with this approach is that it makes it very difficult to see changes in behavior over time.

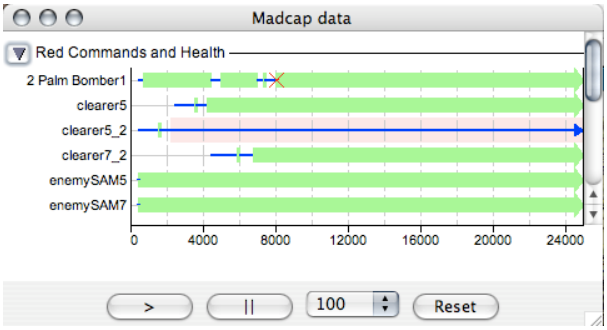

Figure 4. Timeline View

The second method we used was a timeline view (see Figure 4), which was very effective in showing the overall sequence of behaviors. However, this view was ineffective at displaying the details of the agent's decision-making process.

The third method we used was a hyperbolic tree (see Figure 5), which was reasonably effective in showing the plan search process. The tree depicts the overall flow of the search while allowing the user to focus on a particular segment of the search to get a great deal of detail. Though the hyperbolic tree is well-suited to displaying the plan search process, it is not the ideal structure for showing the execution history of other elements of components of our agent architecture, such as finite state machines and reactive behavior tables. This suggests that multiple visualization methods may need to be deployed in concert to obtain full coverage.

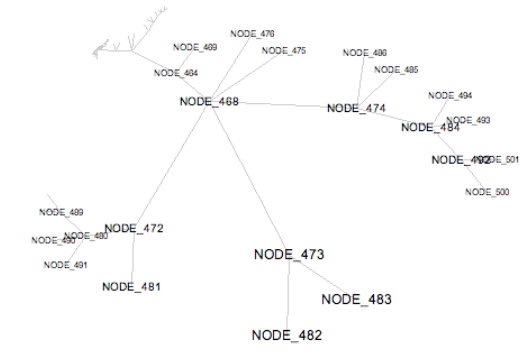

Figure 5. Hyperbolic Tree

The final visualization method that we considered was a TreeMap, which is a 2D display of hierarchical data that uses color and shape to distinguish between leaf nodes. In a TreeMap, the user can select a node to view extremely detailed information about that node. While this approach was somewhat effective at showing planning search data, it was better-suited to the display of static data than to sequences of changing states over time. This proved problematic, since one of the core requirements for visualizations of behavior model execution is the ability to depict the agent's sequence of decisions.

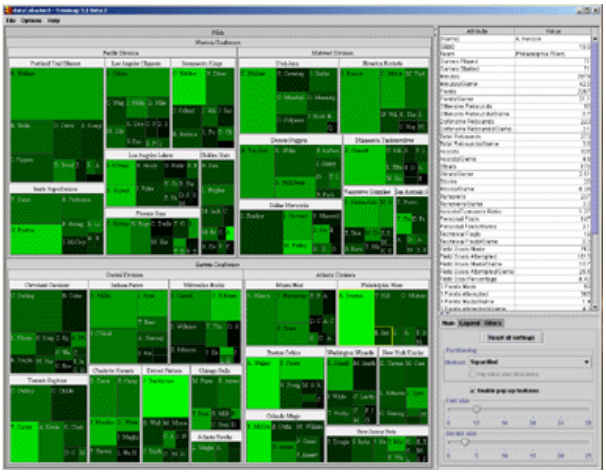

Figure 6. TreeMap View

In our work on Madcap, we did not find a perfect way to display the log data generated by behavior models. However, we discovered that the most effective techniques used sophisticated rendering techniques to display the entire sequence of decisions made by a behavior model with very little detail, but also allowed the user to drill down to specific points in the sequence to see more detail. Future work needs to be done using those broad principles to formulate improved visualizations that are more specific to the needs of behavior models.

CONCLUSION

The problem of allowing subject matter experts to author behavior models is very difficult. This is primarily because the behavior model authors are asked to do tasks that software engineers spend a great deal of time learning to do: changing intuitively understood conditions into a series of Boolean expressions a computer can understand, and then attempting to discover why the computer did not get the expected answer for our series of Boolean expressions. We have attempted to convey some of the many issues involved in trying to ease the burden of creating logical statements. In addition, our experience with developing behavior models for our own agent architecture has

convinced us that making simplified model debugging tools available in a model authoring tool would make the authoring process much more accessible to non-programmers.

ACKNOWLEDGEMENTS

This work was supported in part by Air Force Research Laboratory grant FA8750-05-C-0057.

REFERENCES

Beck, K. (1999). Embracing Change with Extreme Programming. *Computer* 32, 10, 70-77.

Johnson, B. (1992). TreeViz: treemap visualization of hierarchically structured information. *Proceedings of the SIGCHI conference on Human factors in computing systems*, 1992, 369-370.

Lamping, J., Rao, R., & Pirolli, P. (1995). A focus+context technique based on hyperbolic geometry for visualizing large hierarchies. *Proceedings of the Conference on Human Factors in Computing Systems*, 1995, 401-408.