Proceedings of the Sixth AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment

Behavior Compilation for AI in Games

Jeff Orkin, Tynan Smith, Deb Roy

MIT Media Laboratory
75 Amherst St.
Cambridge, MA 02139
{jorkin, tssmith, dkroy}@media.mit.edu

Abstract

In order to cooperate effectively with human players, characters need to infer the tasks players are pursuing and select contextually appropriate responses. This process of parsing a serial input stream of observations to infer a hierarchical task structure is much like the process of compiling source code. We draw an analogy between compiling source code and compiling behavior, and propose modeling the cognitive system of a character as a compiler, which tokenizes observations and infers a hierarchical task structure. An evaluation comparing automatically compiled behavior to human annotation demonstrates the potential for this approach to enable AI characters to understand the behavior and infer the tasks of human partners.

Introduction

Mining data from human players offers a promising new solution for authoring character behavior in games. This data-driven approach has been proven effective for creating computer-controlled players real-time strategy games (Ortanon et al 2007; Weber & Mateas 2009), and preliminary results show potential for generating behavior from data in character-driven games (Orkin & Roy 2009). While it is straightforward to record gameplay to a log file of time-coded actions, state changes, and chat text, it remains an open question how best to process and represent this data such that it will be useful to an AI character.

Many games today implement character behavior with hierarchical representations, such as variants of hierarchical planners or hierarchical finite state machines (Gorniak & Davis 2007; Hoang et al. 2005; Fu & Houlette 2004); for example, Hierarchical Task Networks in *Killzone 2* (Straatman 2009), or Behavior Trees in the *Halo* series (Isla 2005). Ideally, we would like to transform the sequence of observations recorded in a gameplay log into one of these familiar hierarchical structures. This process of parsing a serial input stream of symbols to generate a hierarchical structure is much like the process of compiling source code (Aho et al. 1986).

We compile code for several reasons -- to validate the syntax, to compact the size of the code base, and to represent a program such that it can be readily executed by a machine. The same reasons motivate compiling behavior for characters in games. While observing the interactions of other human and/or AI agents, characters need to parse these observations to recognize valid patterns of behavior, and separate the signal from noise. As we collect thousands (or even millions) of gameplay traces, the storage memory footprint cannot be ignored. While a centralized system can execute a hierarchical plan to control a group of AI characters, if the team includes human players centralized control is no longer an option. In order to cooperate effectively with humans (on the battlefield, or in a social scenario like a restaurant) characters need understand the behavior of others by inferring the tasks they are pursuing, and execute appropriate responses in the right contexts.

In this paper, we draw an analogy between compiling source code and compiling behavior. We propose modeling the cognitive system of a character as a compiler, which tokenizes observations and infers a hierarchical task structure. This structure gives context for understanding the behavior of others, and for selecting contextually appropriate actions in response. However, the analogy is not perfect. There are significant differences between compiling code and compiling behavior. Behavior exists in a noisy environment, where multiple characters may be pursuing multiple, possibly conflicting goals, or engaging in exploratory behaviors that do not contribute to any goal at all. A source code compiler processes code for a single program, and terminates with an error when it encounters invalid syntax. Compiling behavior, on the other hand, requires the ability to ignore fragments of invalid syntax without halting, and simultaneously process tangled streams of observations that may contribute to different goals.

We demonstrate these ideas by describing work with data from *The Restaurant Game*, a corpus of almost 10,000 logs of interactions between waitresses and customers cooperating in a virtual restaurant. First we describe how we tokenize actions, state changes, and chat text into a lexicon. Next we detail how these tokens can be compiled

Copyright © 2010, Association for the Advancement of Artificial Intelligence (www.aaai.org). All rights reserved.

into a model of behavior at runtime, or as a preprocessing step. Finally, we evaluate how well our system compiles behavior as compared to a human, and relate our system to previous work.

The Restaurant Game

The Restaurant Game is an online game where humans are anonymously paired to play the roles of customers and waitresses in a virtual restaurant. Players can chat with open-ended typed text, move around the 3D environment, and manipulate 47 types provides the same interaction options: pick up, put down, give, inspect, sit on, eat, and touch. Objects respond to these actions in different ways -- food diminishes bite by bite when eaten, while eating a chair makes a crunch sound, but does not change the shape of the chair. The chef and bartender are hard-coded to produce food items based on keywords in chat text. A game takes about 10-15 minutes to play, and a typical game consists of 84 physical actions and 40 utterances. Everything players say and do is logged in time-coded text files on our servers. Player interactions vary greatly, ranging from dramatizations of what one would expect to witness in a restaurant, to games where players fill the restaurant with cherry pies. Details about the first iteration of our planning system for AI-controlled characters are available in a previous publication (Orkin & Roy 2009).

Lexical Analysis

In order to compile a sequence of observations, we first need to tokenize the input stream. When compiling source code, a lexical analyzer such as the Unix tool *Lex* is employed to match regular expressions in the input, and export tokens in their place (Levin et al. 1992). Tokens may represent a single symbol in the input, or a pattern composed of several adjacent symbols. For example, the *Lex* specification for analyzing Java code might include:

```
"if"      { return IF; }
"else"    { return ELSE; }
"while"   { return WHILE; }
"="       { return EQ; }
"!="      { return NE; }
[0-9]+    { return NUMBER; }
```

Guided by this specification, *Lex* generates a program that transforms human-authored source code into a sequence of tokens that are more easily interpreted by a machine. Our gameplay data consists of a variety of keywords for actions and state changes, names of objects, and arbitrary strings of chat text. We would like to tokenize this sequence of heterogeneous symbols into a common currency of tokens that can be uniformly processed by the behavior compiler.

Tokenizing actions and chat text is more complex than tokenizing source code, due to the fact that actions refer to objects which change state over time, and open-ended natural language chat text is infinitely varied, yet must be categorized into a minimal number of functionally meaningful dialogue acts. Below we detail our approach to tokenizing physical acts and dialogue acts.

Figure 1. Screenshot from *The Restaurant Game*.

Tokenizing Physical Acts

We learn a lexicon of context-sensitive, role-dependent physical acts through a bottom-up procedure. The state of each object in the game (such as steak, coffee, menus, pots, pans, and appliances) is represented by a small vector of variables including ON, ATTACHED_TO, and SHAPE. (Some objects change shape as a result of an action – steak diminishes with each bite). Our lexical analysis procedure tracks the state of each object over the course of each gameplay session. When the lexical analyzer encounters an action (e.g. PICK_UP), we assume that the current state of the object represents the action's preconditions, and the state changes that immediately follow the action represent the effects. The action's representation also includes the character type of the action's actor (e.g. waitress) and the target object (e.g. pie). We store every unique observed action in the lexicon. For example, we have one entry in the lexicon for a waitress picking up pie from the counter.

Recognizing that many objects serve the same function within the game, we automatically cluster objects by their observed affordances. For each type of object, we count how many times the object is the target of each possible action. From these counts, we compute the likelihood of taking each action with each object, and ignore actions with a likelihood below a hand-tuned threshold. Objects that share the same list of likely actions (affordances) are clustered. For example, we learn that customers tend to sit on chairs and stools, and both steak and salmon tend to be picked up by waitresses and eaten by customers. Clustering objects greatly decreases the size of the action lexicon, which refers to these objects. After processing 5,000 gameplay logs, our lexicon contains 11,206 unclustered actions, and 7,086 clustered.

Once we have learned the lexicon, we can use it to

tokenize gameplay logs. Each action in a log can be replaced with an index into the action lexicon, which serves as a unique token. The behavior compiler can then recognize tasks based solely on token sequences, without having to track the details of state changes.

Tokenizing Dialogue Acts

The effects of physical acts are directly observable in the gameplay logs -- when a waitress picks up a steak, the log records that the steak is now ATTACHED_TO the waitress. The same cannot be said of chat text. When a customer says “I’ll have the steak,” there is no explicit representation in the log of the effect this utterance has had on the interaction. Taking inspiration from the philosophical observation that "by saying something, we *do* something" (Austin 1962) we recognize that utterances can serve as *speech acts* (Searle 1969) that serve a function similar to physical acts, effecting the interaction going forward. However, there are many ways to say the same thing, using entirely different words to serve the same function.

Our approach to tokenizing the chat text relies on a classification scheme that clusters utterances semantically, so that we can represent the infinite variety of ways to say the same thing with an identical token. We classify each line of chat text into a {speech act, content, referent} triple, where *speech act* refers to the illocutionary force (e.g. question, directive, greeting), *content* refers to the propositional content (what is the utterance a question about?), and *referent* identifies an object referenced by the utterance (e.g. beer, menu, bill, waitress). For example, “I’ll have a beer” and “Bring me a pint” would both be represented by DIRECTIVE_BRING_BEER. Our scheme has 8 possible speech act labels, 23 content labels, and 16 referent labels. Members of the triple may be labeled OTHER when no existing labels fit. Elsewhere we have described a dialogue act classifier that can be trained to automatically label utterances (Orkin & Roy 2010). For this study, we hand-annotated the utterances in the 100 games, in order to evaluate behavior compilation in isolation in a best-case scenario. We observed 312 unique dialogue act triples in the 100 annotated games. Augmenting the action lexicon with these dialogue act triples, we can now represent the physical and linguistic interactions in a gameplay log as a sequence of tokens that can be processed uniformly by the behavior compiler.

Syntactic Analysis

Having tokenized the gameplay logs, we can now proceed with compiling behavior in a more conventional manner. Guided by a grammar specification, a compiler infers structure by parsing token sequences. Tokens are the terminals of the grammar. Syntactic rules define non-terminals (on the left-hand side) as any combination of terminals and non-terminals (on the right-hand side), forming a hierarchical structure. If a token sequence cannot be fully explained by the grammar, the compiler halts with a syntax error. *YACC* (*Yet Another Compiler-Compiler*) is a Unix tool that generates a parser given a grammar (Levin et al 1992). The generated parser may include user-provided code to compile recognized patterns into a machine-interpretable intermediate representation. The *YACC* grammar specification for Java syntax might include:

```
If_Statement
: IF '(' Expression ')' Statement
| IF '(' Expression ')' Statement ELSE
    Statement
While_Statement
: WHILE '(' Expression ')' Statement
```

Note that there may be multiple valid sequences representing the same non-terminal. In the grammar fragment above, there are two sequences provided for an `If_Statement`. This is the approach we take to recognizing structure in behavior – providing the behavior compiler with a grammar of task decompositions, where each task may be decomposed into any number of token sequences. For example, our grammar for restaurant behavior might include:

```
W_Serves_Food
: W_PICKSUP_FOOD_FROM_COUNTER
    W_PUTSDOWN_FOOD_ON_TABLE
| W_PICKSUP_FOOD_FROM_COUNTER
    W_GIVES_FOOD_TO_C
    C_EXPRESSIVE_THANKS_NONE
| W_PICKSUP_FOOD_FROM_COUNTER
    W_ASSERTION_GIVE_FOOD
    W_PUTSDOWN_FOOD_ON_TABLE
```

In this example, tokens are presented in plain English for readability, while in actuality the grammar terminals are stored as numerical indices into the action lexicon. What we have described so far employs the exact same approach to compiling source code and behavior. However, there are some complications related to the syntax of behavior that prevent us from directly applying a tool like *YACC*, and require an alternative approach.

The first complication arises from the spontaneous nature of gameplay data, which may include syntactically invalid action sequences related to player exploration, indecision, or intentional misbehavior. Invalid sequences may be composed of valid tokens – ordinary actions and/or utterances executed in abnormal contexts, such as a waitress ordering 50 beers and stacking them on the bar instead of serving them to customers. Rather than halting with a syntax error, we would like our behavior compiler to be able to separate the signal from the noise, and to disregard invalid sequences while continuing to process valid observations.

The second complication relates to the fact that there are multiple human players interacting, who may be pursuing multiple tasks at once. A waitress might take a customer’s order while cleaning the table, as the customer drinks wine. The actions belonging to these tasks may be arbitrarily

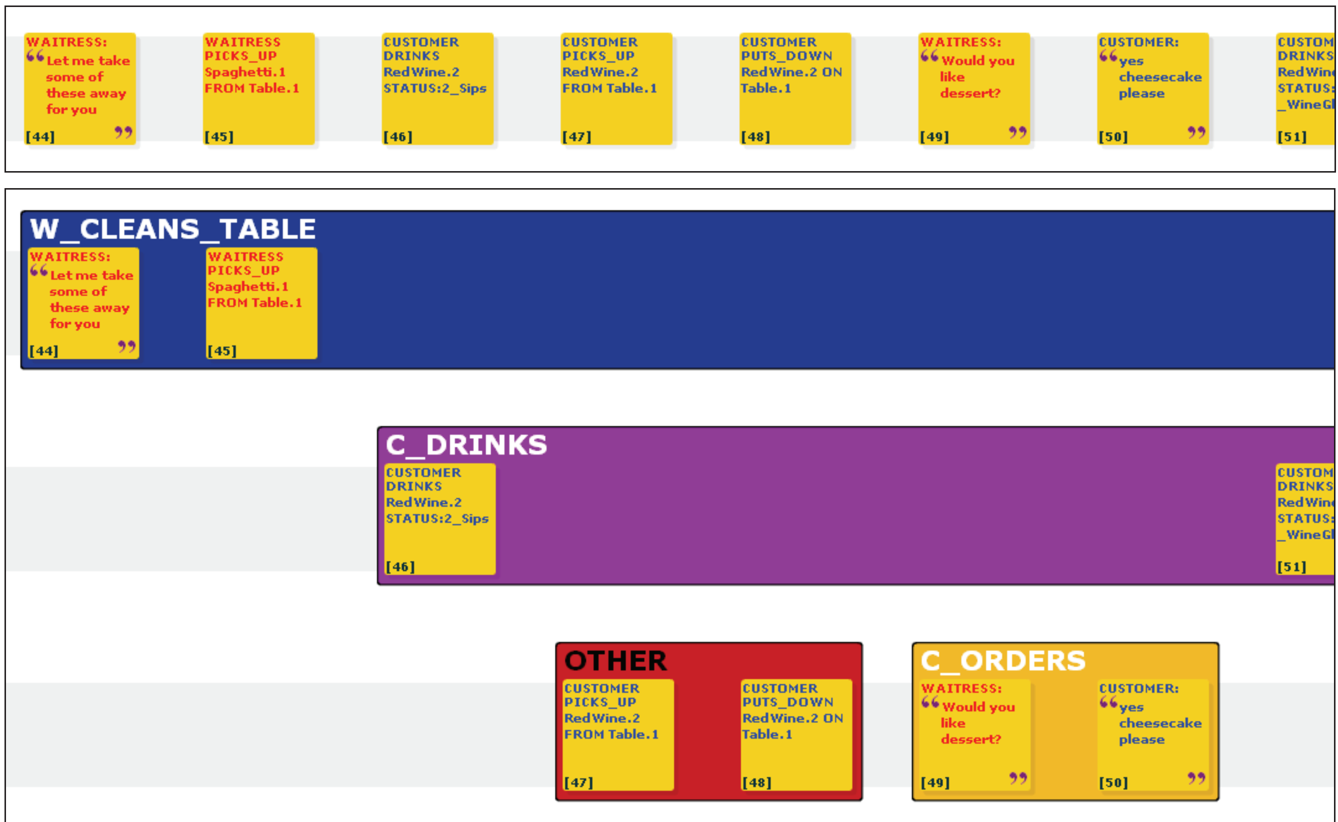

Figure 2. Gameplay trace in EAT before (top) and after (bottom) annotation.

intertwined, and may contribute to unrelated top-level goals (e.g. satisfying hunger, earning money, getting promoted). To bring the analogy back to source code, it's as if the compiler is faced with code from multiple programs that have been arbitrarily combined into one input file, and needs to compile them all simultaneously. Below we describe our approach to behavior compilation that addresses these complications.

Behavior Compilation

The grammar provided to the behavior compiler could be authored by hand, but to maximize coverage of possible valid sequences we learn the grammar from human-annotated gameplay transcripts. We have previously published details about the implementation and evaluation of EAT & RUN (Orkin et al. 2010). EAT (the Environment for Annotating Tasks) is an online annotation tool for non-experts and RUN (not an acronym) is an algorithm for recognizing tasks at runtime, trained on annotations from EAT. We summarize this previous work here.

EAT presents each gameplay session as a timeline, where nodes represent physical acts and dialogue acts. The annotator encloses node sequences in labeled boxes representing tasks, and assigns a task label. When nodes from multiple tasks are interleaved, the annotator can move nodes up or down on the screen to disentangle the tasks, allowing tasks that overlap temporally to enclose only the relevant nodes. Bigger boxes can enclose smaller task boxes, to annotate higher levels of the hierarchy. We annotated the first level of the task hierarchy in 100 games, from which we generated a grammar by exporting each unique token sequence as a syntactic rule. Our grammar has 28 task labels (e.g. W_DEPOSITS_BILL, W_SERVES_FOOD, C_GETS_SEATED, C_DRINKS), serving as non-terminals, plus an OTHER label for unclassifiable token sequences. Based on a ten game subset, we found substantial agreement between the annotations of an expert and five annotators who were not involved with the development of EAT (average kappa of 0.81). Annotating higher levels of the hierarchy remains for future work, and we will represent these higher-level tasks by exporting the sub-tasks (which may temporally overlap) as a sequence of tokens representing task start and end points.

RUN is a simple, greedy algorithm that recognizes tasks at runtime as observations arrive, or can be run as a preprocessing step to process a corpus of gameplay logs. The algorithm extends multiple observation sequences in parallel by maintaining incomplete sequences on an Open list. A sequence can be validly extended by appending an observation, as long as it remains a subsequence of the right-hand side of a syntactic rule. When a sequence in the Open list grows to match a complete sequence on the right-hand side of a syntactic rule in the grammar, a task label is assigned (the left-hand side of the matching syntactic rule)

```
// Process all new observations.
Iterate over new observations
   Try to append obs to seq in Open
   Else try to add obs as new seq in Open
   Else try to append obs to seq in Closed

   // Detect completed rules.
   If a seq in Open was added or extended
      If seq matches a complete rule
         Move seq to Closed and apply label

// Salvage fragments left in Open.
Iterate over seqs in Open
   Try to append to seq in Closed
```

Figure 3. Outline of the RUN algorithm.

and the sequence is moved to the Closed list. In cases where an observation cannot extend a sequence or start a new sequence in the Open list, sequences in the Closed list may continue to be extended. This captures the fact that the grammar includes sequences that may be subsequences of longer syntactic rules. At any point in time, the sequences in the Closed list represent the recognized task history so far. Sequences not found in the Closed list are assigned a task label of OTHER. RUN processes tokens at one level of the task hierarchy at a time, initially processing only the terminal tokens. The output of RUN at one level serves as the input tokens for the next level.

Compiling the corpus of gameplay logs with RUN generates a library of thousands of hierarchical plans, executable by a character. Compiled logs are stored in text files where each line contains a token, time code, task instance ID, and plain English description of a physical act (for debugging) or the line of quoted human chat text corresponding to the dialogue act. We are currently building on previous work (Orkin & Roy 2009) to develop a planning system driven by the RUN algorithm. Characters employ RUN to process observations, and infer which tasks have been completed and which are in progress. Retrieving compiled logs with similar task histories from the library provides the means to select an appropriate next action in the context of current observations.

Evaluation

We evaluate our approach to behavior compilation based on the criteria of correctness and compactness of the compiler's output. Ideally, the compiler will filter out syntactically invalid behavior; produce a task hierarchy that closely resembles human annotation for the same gameplay log; and the exported files will be significantly smaller than the original logs.

We evaluated our compiler with a 10-fold cross-validation test of recognizing tasks in the 100 human annotated gameplay traces, where each fold was trained on 90 traces and tested on 10. Our results show that RUN works well on our dataset, achieving a 0.744 precision and 0.918 recall. Precision measures how often RUN and the human assigned the same task label to a token. Recall measures how many of the human-annotated task instances were recovered by RUN. The baselines for precision and recall are 0.362 and 0.160 respectively. While these results are encouraging, we need to evaluate our system on a dataset of gameplay traces from another domain for comparison. The fact that RUN scores higher recall than precision indicates that the system is doing a good job of getting the gist of the interaction, but is sometimes omitting tokens within a sequence.

Unlike a source code compiler, which terminates with a syntax error when encountering invalid syntax, our compiler needs to filter out invalid behavior while continuing to process subsequent observations. We can measure how well the compiler filters invalid behavior by scrutinizing the number of tokens assigned to task OTHER by a human and by the compiler. We found the compiler has 89% precision and 82% recall for task label OTHER, meaning that a human agreed with 89% of the tokens that the compiler filtered out, and the compiler caught 82% of the tokens filtered out by a human.

The file sizes of the compiled log files are considerably smaller than the original gameplay logs. We compiled a directory of 9,890 log files, and the size of the directory decreased dramatically from 2.87 GB to 165 MB. Currently log files are compiled into a text-based format, which includes a plain English description of a physical act for debugging or the line of quoted human chat text corresponding to the dialogue act. Further compression is possible by compiling into a binary format, omitting the debug information, and storing references into an external dictionary of phrases rather than storing the raw dialogue text (which is often repeated).

Related Work

Behavior compilation is closely related to plan recognition (Kautz & Allen 1986), but requires the ability to recognize multiple plans simultaneously, in cases where temporally overlapping tasks contribute to different top-level goals. Furthermore, gameplay data is typically not logged in a tokenized format amendable to plan recognition, requiring lexical analysis as a preprocessing step. Gorniak and Roy employed plan recognition to disambiguate directives issued to an AI-controlled partner while solving a puzzle in *Neverwinter Nights* (2005). Their system employed a probabilistic parser guided by a hand-crafted behavior grammar. Our system learns a grammar from annotated data, and tries to recognize tasks contributing to multiple top-level goals in parallel, which incorporate a wider variety of speech acts than directives.

Our approach to learning hierarchical plans from annotated gameplay data is similar to work on Case-Based Planning for strategy games (Ortanon et al 2007). Their system extracts cases from a small number of annotated games. In our system, a small number of human-annotated games train the behavior compiler to automatically annotate a corpus of thousands, generating a large case base that captures subtle variations between gameplay logs.

In addition, we do not assume anything about the hierarchical structure from the temporal overlaps of tasks, and instead rely on human annotation of higher-level tasks to determine which tasks encompass lower-level sub-tasks. Unlike a strategy game, where all player interactions ultimately contribute to the high-level goal of winning the game, there is no notion of winning *The Restaurant Game*. Plus, we are modeling the interactions between two players. Thus we cannot make assumptions about hierarchical structure based on temporal relationships – ordering food while the waitress cleans the table does not mean that ordering is a sub-task of cleaning. These tasks contribute to unrelated top-level goals of having a meal, and keeping the restaurant clean.

Conclusion

Our evaluation compared the output of the behavior compiler to human annotation of the same gameplay logs, and demonstrated that the compiler does well at inferring tasks from observations and filtering out invalid behavior, when tested on our data set. Future work remains to show that these results generalize to other data sets, and that the compiled logs work well to drive character behavior at execution time. We are currently developing a planning system driven by the compiled logs to test execution, and will soon begin developing a new game to test generalization of the approach to a completely different scenario.

When tokenizing observations from our current scenario, we benefit from the fact that actions are discreet. For example, clicking the PICK_UP button on the interface results in the character immediately grasping the object. While interaction in other games may not be as discreet, we are optimistic that similar tokenization should be possible. The intuition for this optimism is that if behavior can be represented with a hierarchical plan or state machine, there must be some means of recognizing the completion of actions in order to proceed. In theory, these completion points can be transformed into tokens for behavior compilation.

Executing the plan compiled from one gameplay log works no differently than currently existing systems in the game industry, which execute behavior represented in a hierarchical plan or state machine. More complex is a system that dynamically pivots between thousands of compiled logs, to adapt a character's behavior to cooperate with another autonomous human or AI-controlled partner. Running the behavior compiler at execution time, as a character processes observations, will provide the means to recognize the task history of the interaction, and use this history to find similar episodes in the corpus.

Acknowledgements

This research is supported by a grant from the Singapore-MIT GAMBIT Game Lab.

References

Aho, A.V., Sethi, R., and Ullman, J.D. 1986. Compilers: Principles, Techniques, and Tools. Addison-Wesley.

Austin, J.L. 1962. *How to do things with Words*. Oxford University Press.

Fu, D. and Houlette, R. 2004. The Ultimate Guide to FSMs in Games. *AI Game Programming Wisdom 2*. Charles River Media.

Gorniak, P., and Roy, D. 2005. Speaking with your sidekick: Understanding situated speech in computer role playing games. *Proc of Artificial Intelligence and Digital Entertainment*.

Gorniak, P. and Davis I. 2007. SquadSmart: Hierarchical planning and coordinated plan execution for squads of characters. *Proc of Artificial Intelligence and Interactive Digital Entertainment*.

Hoang, H. Lee-Urban, S. and Munoz-Avila, H. 2005. Hierarchical plan representations for encoding strategic game AI. *Proc of Artificial Intelligence and Digitial Entertainment*.

Isla, D. 2005. Handling Complexity in the Halo 2 AI. *Proc of the Game Developers Conference*.

Kautz, H. and Allen, J. 1986. Generalized plan recognition. *Proc of the National Conference on Artificial Intelligence*.

Levine, J.R., Mason, T., and Brown, D. 1992. *LEX & YACC* (2nd edition). O'Reilly.

Orkin, J., Smith, T., Reckman, H., and Roy, D. 2010. Semi-Automatic Task Recognition for Interactive Narratives with EAT & RUN. *Proc of the 3rd Intelligent Narrative Technologies Workshop*.

Orkin, J. and Roy, D. 2010. Semi-Automated Dialogue Act Classification for Situated Social Agents in Games. *Proc of the AAMAS Agents for Games & Simulations Workshop*.

Orkin, J. and Roy, D. 2009. Automatic Learning and Generation of Social Behavior from Collective Human Gameplay. *Proc of Autonomous Agents and Multiagent Systems*.

Ortanon, S., Mishra, K., Sugandh, N., and Ram, A. 2007. Case-based planning and execution for real-time strategy games. *Proc. of the 7th Int. Conference on Case-Based Reasoning Research and Development*.

Searle, J.R. 1969. *Speech Acts: An Essay in the Philosophy of Language*. Cambridge University Press.

Straatman, R. 2009. The AI in Killzone 2's Bots: Architecture and HTN Planning. *AIGameDev.com*.

Weber, B. and Mateas, M. 2009. A Data Mining Approach to Strategy Prediction, *Proc of the IEEE Symposium on Computational Intelligence in Games*.