Automatically Detecting Narrative Patterns within Gameplay Logs

Stephen Roller (scroller@ncsu.edu)

12/2/09

Abstract

In recent years, many video games have added the ability to play and complete with other players over the Internet. The complex interactions between players in multiplayer games often result in interesting narratives in gameplay. Many of the most interesting narratives go unnoticed by the players, who are too constricted by their own perspective and objectives to see the complete picture. This paper presents the Narrative Pattern Matcher, a system for automatically locating narratives that occur within a multiplayer game. The NPM uses user-defined patterns to generate and apply finite state machines to game logs. Features, implementation and limitations of the NPM are described in detail. Finally, the runtime performance of the NPM is evaluated in an experimental setting.

Introduction

The increased popularity of video games has resulted in more and more activities occurring in a virtual environment. During the course of a multiplayer game, many interesting narratives are likely to emerge through the complex interactions of players. Often times, players are unaware of these narratives. They may be too focused on their own game objective, or simply unable to see the bigger picture through their eyes. Subtle, unplanned teamwork may go completely unnoticed by participants, especially if events occur on opposite sides of the virtual world. If such interactions are ever to be experienced fully, a way to automatically detect and replay narratives is needed.

This thesis describes the Narrative Pattern Matcher (NPM), a system for detecting narratives in specialized game logs of online multiplayer games. The Narrative Pattern Matcher is one component of a larger system, called Afterthought, which produces automatically generated cinematic highlights of multiplayer games.

Related Work

While the work done in automatic game summarization and highlights is limited, there have been a number of related works in recent years. Many sports games, such as Tony Hawk and FIFA, contain limited replay modes. Some games focus on instant replays, but such replays tend to be of a single event with obvious importance, such as a goal being

scored. Many games, such as FIFA, also allow the user to save and replay entire games. However, these games typically require the viewer to choose camera angles and select important moments.

There is a great amount of work being done in automatic summarization of real sports events and videos. However, many of these techniques are more focused on extracting information from individual frames of video footage. Unlike video games though, summarization of real games cannot employ game logs to help find important moments or description of the events occurring.

Other researchers have begun analyzing video game logs in order to make conclusions about events. Tveit and Tveit (2002) apply data mining techniques to game logs from MMORPGs, but focuses on estimating duration of game sessions rather than producing summaries.

Video game summarization has also appeared in the form of automatically generated comic panels. Chan, Thawonmas and Chen (2009) create summarizations of *World of Warcraft* games by logging and screen capturing events. Images are weighted by event frequency and importance and fitted into a random panel layout.

Shamir, Rubinstein and Levinboim (2006) also create comic book summaries of *Doom* games, but uses more advanced techniques for frame selection. A particular agent or character is selected as a narrator, limiting summarization to events experienced by this narrator. The narrator's interaction and proximity with other game entities is calculated as a function of time and smoothed. High points in the interaction function are selected as interesting events, while low points and location changes determine scene changes. Scenes lacking interesting events are ignored. Single frames from each event are compiled and laid out in comic book form.

Halper and Masuch (2003) use similar methods to produce two alternative outputs: a game summarization in the form of a sequence of images and a real-time spectator mode. Changes in world state are logged with periodically. Several *i-functions*, or "interestingness functions," are calculated by taking the derivative of world state. Larger stage changes are reflected as higher values in the i-function. I-functions are combined, smoothed weighted by a predetermined factor. Like Shamir (2006), high points and low points are used to choose frames and detect scene changes. Unlike other systems, Halper's system can also be

used to help spectators find interesting moments in real-time. Spectators can choose to either watch a single scene until its finish, or to abruptly switch to more interesting scenes.

Friedman, Feldman and Shamir's (2004) work is perhaps most similar to Afterthought, in that it attempts to create high-quality video summaries of games. Friedman's system logs game events and passes them into a *narrative extractor* (NE), whose role it is to determine the most interesting parts. Friedman defines interesting as the deviation from regular patterns or *routines*. Routines have several components: a specific start action, a specific end action, and a partially ordered set of intermediate actions. The routine matcher searches logs for specific actions, while keeping track of which dependencies have been met. A routine is located when the end-action is found and all dependencies are filled. Routines with a small number of unsatisfied dependencies are marked as deviations. Similar to Afterthought, Friedman's system also looks for all possible pattern matches within a system, sometimes looking for many routines in parallel. Routines are not located within a time limit are eliminated from the search. Friedman generates movies by concatenating all actions within a particular routine.

Finally, Simòes explores the state of game summarization methods, particularly with respect to narrative generation systems. Simòes gives brief summaries of several automated summarization techniques, including some of those explored above. While Simòes does not create any new methods for summarization, she does categorize the different techniques used and explores their relation to narrative generation.

Although Afterthought is unique from these systems in its assumption that sequences of game actions can be treated as a formal language, it is not the first to make this assumption. Jantke (2006) begins by showing that games can be seen as a series of player inputs and that the concatenation of these inputs is one way to play the game. Jantke explains that the game itself can be seen as a generating language for these sequences of inputs that hierarchical layers of meaning can be attached to these inputs. Finally, Jantke gives several examples of games that can be modeled in this fashion. While Jantke does not attempt summarize games, his work does model player actions as a formal language in a manner similar to the Narrative Pattern Matcher.

Relationship with Afterthought

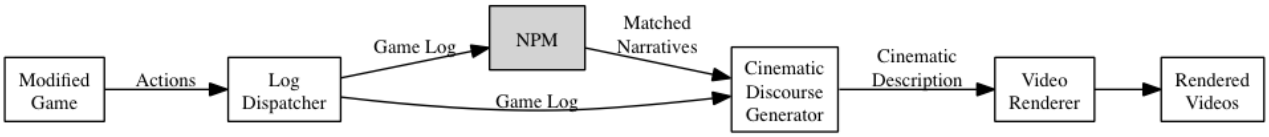


Figure 1: Overview of Afterthought

The Narrative Pattern Matcher is only one part of a larger system called Afterthought. Afterthought is a system for producing cinematic highlights of emergent narratives that occur during Unreal Tournament 3 (UT3) gameplay sessions. Afterthought is composed of five distinct components: a modified UT3 game, the Log Dispatcher, the Narrative Pattern Matcher, the Cinematic Discourse Generator (CDG), and the Video Renderer.

The modified UT3 game watches for important game events, or *actions*, and notifies the Log Dispatcher. An action is any notable occurrence, such as "Player1 kills Player2" or "Player1 picks up an item." Each action has a number of parameters or attributes associated with it, such as amount of damage taken or name of the victim. All actions must have three parameters: type (e.g. damage, kill, pickup), timestamp and unique ID. Actions must be given in the same order they occurred, but their timestamps may overlap.

The Log Dispatcher passes the log of actions as an XML document over a socket connection to the NPM and the CDG. The NPM locates and identifies all narratives that occurred within the gameplay. After finding the complete narratives, the NPM reports matched narratives to the CDG via another socket. The CDG selects which narratives should be rendered and uses the game logs to develop a *cinematic discourse*. The cinematic discourse contains information about which camera shots are appropriate, timing of shots, placement of camera and more. This cinematic discourse is used by the Video Renderer to produce complete, rendered videos portraying the chosen narratives. The final videos can optionally be automatically uploaded to YouTube or another video sharing website.

Overview of NPM

The NPM has several distinct phases. First, the user writes pattern files to describe the interesting narratives the NPM should look for. These pattern files are used to generate executable Finite State Machines (FSMs). The Pattern Matcher reads in the game logs and

feeds them into FSMs, keeping track of possible narratives along the way. The matched narratives are aggregated and reported as output of the entire system.

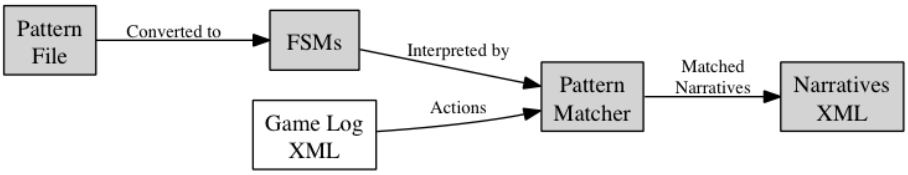


Figure 2: Overview of the Narrative Pattern Matcher

The user specifies patterns in the form of specialized regular expressions. While these regular expressions are similar to regular expressions in automata theory, there are several important differences. Traditional regular expressions describe strings of characters, rather than sequences of high-level actions. Additionally, regular expressions explicitly fail when an unexpected character is reached. The NPM, on the other hand, treats unexpected actions as noise and ignores them. Additionally, the NPM supports storing and referring to previous data, making it more similar to Perl's regular expressions than pure regular expressions (Dominus 1998).

Description of Pattern Language

The most basic operator of the pattern language is an action operator. The action operator is analogous with a single character operator in traditional regular expressions. An action operator has the syntax like `(action-type :parameter1 value1 :parameter2 value2 ...)`. An action matches this operator if the action has a type of *action-type* and each specified parameter of the action is equal to the corresponding value specified. For example, `(kill :victim "Player1")` would match any kill with Player1 as the victim.

The user combines action operators together using three standard regular expression operators: concat, star and union. Concat specifies one pattern should occur immediately after another. Star specifies that one pattern should be matched zero or more times. Finally, union specifies that either of two patterns may be matched. These three operators can be combined and composed in order to build complicated and powerful FSMs. Additional convenience operators, such as one-or-more and repeat, are defined using these three basic operators.

The pattern language can remember the value of specific action parameters in named *variables*, similar to Perl's grouping and substitution. Variables can only store

numbers and strings; variables may *not* refer to higher-level objects, such as actions or lists. Syntactically, variables are always a question mark followed by at least one letter. They can be used as values in action operators in order to ensure parameters are the same across actions and only appear within action operators. For example, `(concat (kill :killer ?Player) (kill :killer ?Player))` will only match two kills with the same killer. Variables are bound the first time they are used. After a variable is bound, its value is immutable. However, since the NPM tests all possible combinations of actions which matching, a different combination may bind the variable to a different value.

The NPM also allows the user to add different constraints to the FSMs. Constraints are additional checks performed during specific transitions in an FSM. Syntax of constraints depends on the type of constraint, as does the point in the FSM when the constraint is checked. Timing constraints, such as these actions must occur within 10 seconds, are the simplest kind and checked at every point within the FSM.

Another type of constraint users may impose is a sequence of actions that may *not* occur between two points. For example, `(dontmatch :bad_dfa (kill :victim ?P) :good_dfa (concat (kill :killer ?P) (score :player ?P)))` means that a player must kill and score, but cannot die in between these two actions. This type of constraint is checked after the entire good pattern matches.

The final type of constraint is a relationship between variables. While only two types of relationships are currently supported, equal to and not equal to, more relationships can easily be added to the system as needed. These "user constraints" may also be combined using standard Boolean logic.

Conversion of Pattern Files to Finite State Machines

Pattern files are transformed from regular expressions into finite state machines in a predictable process similar to automata theory. First, the entire pattern file is read and parsed into an abstract syntax tree (AST). This tree is then traversed in postfix fashion, replacing branches with the FSM equivalent of the branch. After a pattern's entire AST has been processed, the result is the FSM representing the pattern.

Every action operator is transformed into a 2-state FSM with a start state, an end state, and a single transition matching the given action. The three regular expression operators are implemented using the same methods described in automata theory: null

transitions are added in choice places in order to represent choices the system must make. These null transitions are later eliminated through standard NFA to DFA conversion described in (Sipser).

Variables are handled within the transition of an action operator. When a state transition is attempted, the FSM is given the current bindings of all the variables and each parameter of the action is checked against the user specified value. If the user specifies a parameter should be a specific value, then string comparison is used. If the user specifies a parameter should be a variable, then the parameter is compared to the binding of that variable. If the variable is unbound, then it is set to the value it is compared against and the FSM reports the new bindings. The transition is taken only if all parameters check out.

Constraints are implemented by modifying the FSM to perform additional checks at specific transitions. Like normal transitions, constraints may accept the transition (possibly updating variables) or reject the transition. Constraints can be applied to three classes of transitions: transitions out of the start state (start-state transitions), all transitions *not* coming from the start state (all-state constraints) and transitions going into a final state (final-state constraints). Start-state constraints are typically used to bind an extra variable for later use. All-state constraints are used to perform efficient, simple checks for optimization. Final-state constraints, the most common, are used to ensure a matched pattern meets some final check.

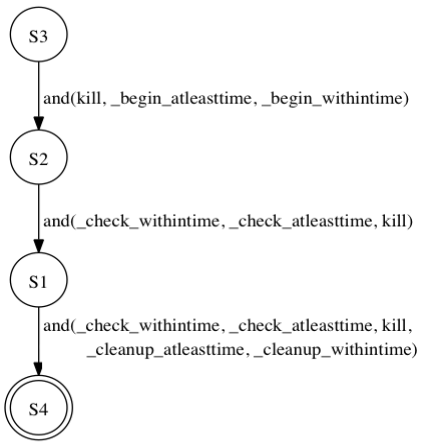


Figure 3: An example FSM with timing constraints.

These three classes of constraints may be composed so that complicated checks are performed efficiently and predictably. For example, consider the pattern `(within-time 10 (atleast-time 5 (concat (kill) (kill) (kill))))`, which matches any 3 kills that all occur within 5 to 10 seconds. The FSM generated for this pattern is shown in Figure 3. The start-state transition from S3 to S2 first looks checks that the incoming action is a kill. If it is, then begin_atleasttime and begin_withintime are both executed, storing the timestamp of first kill for comparison later. The all-state transitions are modified to include check_withintime and check_atleast time before checking for the next kill. The final-state

transitions are performed last to perform cleanup of temporary variables. Note that start-state and final-state checks are performed in the same order added, but all-state checks are performed in reverse order. This allows timing constraints to short circuit and fail early.

The "don't match" constraint, which ensures that a pattern is *not* matched, works in a similar fashion. A special final-state constraint is added; when reached, it instantiates a new pattern matcher for the bad pattern and replays all events since the start-state transition. If the bad pattern does not match, then the final transition is taken. Variable constraints are also checked in final-state transitions.

Algorithm for Pattern Matching

While the process of converting pattern files into FSMs has been explained thoroughly, the actual process of matching patterns has not.

To begin, a match tree is created for each FSM. A match tree is an n-ary tree that keeps track of all possible matches. Every node stores a state number, variable bindings, and the action that caused the transition into this state. The children of a node are all the successful transitions out of the node's state, while the parent chain of nodes represents the actions that led to this state. Match trees are initialized as a single root node containing the FSM's start state, no bound variables and a placeholder action. Whenever a new action is received, the match tree is traversed in postfix order. The FSM checks to see whether, in this state and with these bindings, the new action causes a transition. If the transition is taken, a new child node is created, containing the new state, new bindings (if any), and the new action. This match tree ensures all possible combinations of actions are checked. If an action ever causes a transition to a final state, the parent chain of the new node contains all the actions that compose a particular narrative.

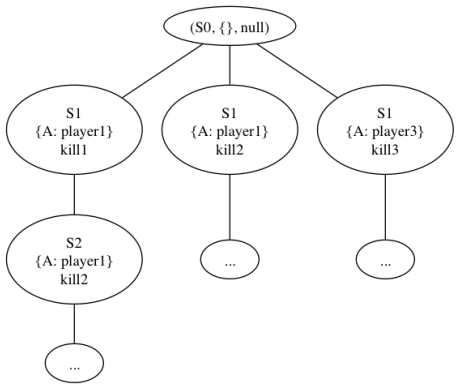


Figure 4: A possible match tree for `(star (kill :killer ?A))`

The matching algorithm described above is very simple, but does convey the entire truth. Several optimizations were added, resulting in a more complex algorithm. The first optimization, called the vocabulary optimization, works by filtering actions before they reach the match tree. When an FSM is created from the pattern file, it keeps track of its

vocabulary, or the set of action types referenced in the FSM. Only actions in the vocabulary set need to be processed by the match tree. This does not preclude the action from being processed by a different FSM. This significantly improves the signal-to-noise ratio of incoming actions, but is ineffective if the FSM has a rich vocabulary.

Since the time it takes to process an action is proportional to the size of the match tree, it is important to keep the match tree to a minimum. One optimization, called same-state, works by pruning unnecessary branches from the match tree. If an action causes a transition into the same state, a new child is *not* created; rather, the new action is "tacked" onto the existing node. This prevents a new branch from being created, thus reducing the number of checks needed for future actions. While this prevents some narratives from being found, such narratives would be different in uninteresting ways. For example, `(star (kill))` may produce the set of narratives {[k1], [k2], [k3], [k1, k2], [k1, k3], [k2, k3], [k1, k2, k3]} without the same-state optimization, but would produce {[k1], [k2], [k3], [k1, k2], [k2, k3], [k1, k2, k3]} with the optimization. While only one narrative is eliminated (in this example), the difference in the match trees is significant (Figure 5). Same-state optimization helps *only* when the star operator is applied to a single action operator.

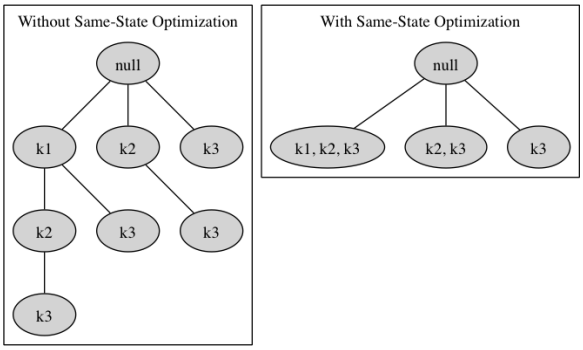


Figure 5: Example match trees with and without Same-State optimization

Another optimization, called kill branches, prunes branches that will never match. When a constraint realizes that subsequent actions will never progress this branch, a special exception is thrown. The earliest parent that cannot cause a match is found and removed from the tree (along with its children). Such an exception is thrown only under two conditions: when a within-time constraint finds an action failing the timeout condition and when a dontmatch constraint finds a match of the bad pattern. Because narratives are reported when located, no earlier narratives will be missed due to kill-branches.

Because each FSM has its own match tree, patterns can run in separate processes and take advantage of multiple cores available in modern computers. Parallelization also prevents slow FSMs from blocking faster ones from completing, making the pattern matcher only as slow as its slowest pattern. Furthermore, if one pattern is taking too long,

it can be killed without affecting any of the other patterns. Parallelization offers a significant performance boost for many nontrivial cases. Since parallelization can create difficulty with debugging, it can easily be toggled on or off.

Limitations of Narrative Pattern Matcher

While the pattern matcher has many features, it also has several distinct limitations. Because the pattern matcher looks for all possible narratives in a game log, the match trees can grow very large, resulting in large memory usage and slow performance. While the described optimizations help mitigate some of these issues, they do not cover all cases. For example, `(repeat 5 (kill))` will find any 5 kills. If more than 5 kills are in the log, the match tree will grow very large as it locates all 100 choose 5 possible narratives.

Expressivity of the pattern language is limited since patterns are modeled as simple FSMs. Patterns concisely defined with context free grammars can be difficult or impossible to with FSMs. Additionally, the pattern matcher has no ability to keep track of world state; there is no way to derive a player's score or health. While variables and constraints significantly improve expressivity, the inability to perform arbitrary calculations (such as adding two variables) further exacerbates the issue. Extensibility is also limited in this way; without greater expressivity, the pattern matcher must sometimes be modified to add support for specific desired patterns.

Jamie Zawinski once jested, "Some people, when confronted with a problem, think 'I know, I'll use regular expressions.' Now they have two problems" (Friedl 2006). Composing regular expressions is often very difficult; many times the user's mental model of a regular expression is incongruent with the FSM actually generated. Users may have trouble with corner cases and exceptions to rules, or they may accidentally allow for false matches. To alleviate such problems, the pattern matcher can optionally output a graphical representation of the generated FSMs.

Furthermore, Afterthought's pattern language has additional nuances that can cause confusion. Variable scoping can be particularly tricky to new users; if an FSM has some path that does not bind a variable, referring to that variable will cause unexpected results. These unexpected results can slip through testing, resulting in bugs not in the pattern matcher, but in the patterns themselves! Unfortunately, these obstacles preclude most nontechnical users from writing narrative patterns.

Results & Runtime Performance

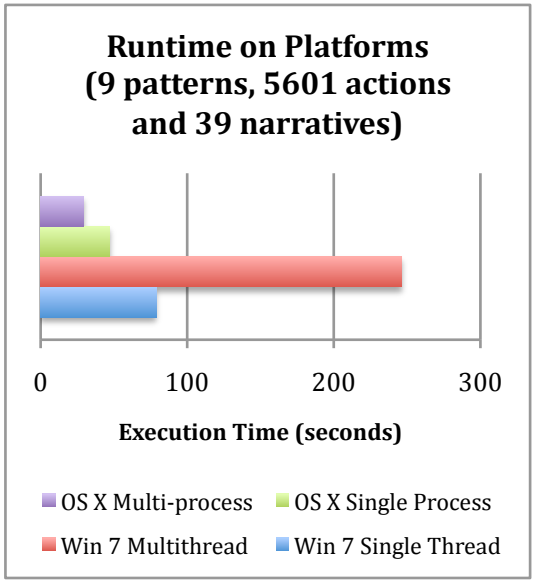


Figure 6: Runtime Performance on Windows and OS X, with and without parallelization.

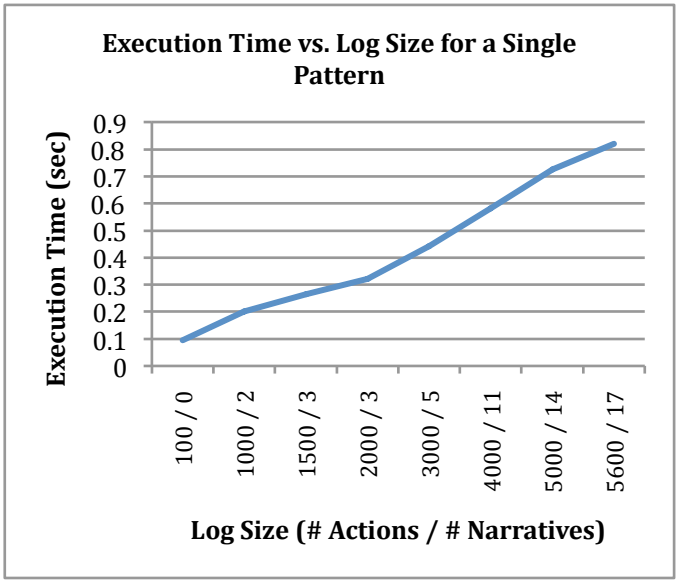


Figure 7: Runtime as a function of Log Size

A controlled experiment was performed to measure the effectiveness of Afterthought as a whole. Three multiplayer games of capture the flag were hosted. 29 players were shown four different videos of their match and asked to rate the videos on five scales, including humorous, dramatic and coherent. While these experimental results only apply to the Cinematic Discourse Generator, the NPM was benchmarked using game logs and patterns from the experiment. All benchmarks run on the same Macbook Pro with a 2.53 GHz Core 2 Duo processor and 2GB of RAM.

Figure 7 shows runtime performance of a single pattern as a function of log size. The pattern chosen was the most commonly matched pattern in the experiment. A log from one of the experiment sessions was chosen and truncated to different lengths for the benchmark. Figure 6 shows a benchmark of several patterns run on four different setups: OS X with and without multi-processing and Windows 7 with and without multithreading. These benchmarks were run with the same 8 patterns used in the evaluation experiment, along with the complete log as used in Figure 7. Testing showed that the NPM always ran faster on OS X than Windows 7.

The large difference in speed between parallelized code in Windows and OS X is most likely explained by a difference in implementation. The OS X version took advantage

of multi-processing, while the Windows version was only multithreading due to a limitation in the implementation language, Python. As a result, OS X was allowed to take advantage of multiple processor cores, but Windows was limited to only one.

Benchmarks also showed that parallelized code was slower for *both* operating systems for short logs. Presumably, the overhead of new processes and synchronization exceed the time saved in cases of short logs with few narratives. Benefits of parallelization on OS X were even more pronounced quad-core Mac Pro, but these benchmarks were not included because of major differences in hardware.

Figure 6 and Figure 7 clearly show that performance is a function of log size, number of patterns and total number of matched narratives. However, these results should be taken with a grain of salt. Performance is widely dependent on the patterns themselves; more complex patterns with larger FSMs usually result in poorer performance. Some patterns exploit weaknesses in the pattern matching algorithm and have horrible performance (see Limitations of Narrative Pattern Matcher). On the other hand, patterns with timing constraints are likely to be much faster, since the match tree will likely be automatically reduced throughout the matching process.

Conclusions & Future Direction

In this paper, the author defines and describes the Narrative Pattern Matcher, a program for finding user-specified patterns of behavior in multiplayer games. The NPM is shown to be one component of Afterthought, a complete system for generating content-rich videos of gameplay. The important features and implementation of the pattern language are shown. A complete description of the pattern-matching algorithm, along with appropriate optimizations and heuristics is also provided. Finally, we consider the limitations of the system, as well as runtime performance in an evaluation of Afterthought.

Although the NPM is rich and powerful, it has room for many improvements. In future work, we hope to extend the NPM to include more expressive features, such as basing patterns on Modal Finite State Machines and allowing more powerful user-defined constraints. We also hope to provide nontechnical users with the ability to write patterns by creating an intuitive, graphical FSM editor. Finally, we hope to replicate Afterthought in games other than UT3, such as Counter-Strike or The Sims.

References

Chan, C.J., R. Thawonmas and K.T. Chen. "Automatic storytelling in comics: a case study on World of Warcraft." Proceedings of the 27th international conference extended abstracts on Human factors in computing systems. Boston, MA: ACM, 2009. 3589-3594.

Dominus, Mark-Jason. How Regexes Work. 1998. 24 September 2009 <http://web.archive.org/web/20080331225447/http://perl.plover.com/Regex/article.html>.

Friedl, Jeffrey. Source of the famous "Now you have two problems" quote. 15 September 2006. 24 November 2009 <http://regex.info/blog/2006-09-15/247>.

Friedman, D., et al. "Automated creation of movie summaries in interactive virtual environments." Proceedings of IEEE Virtual Reality 2004. IEEE Computer Society, 2004. 191-199.

Halper, N. and M. Masuch. "Action summary for computer games: Extracting action for spectator modes and summaries." Proceedings of 2nd International Conference on Application and Development of Computer Games. 2003. 124-132.

Jantke, K.P. "Pattern Concepts for Digital Games Research." Knowledge Media Technologies (2006).

Shamir, A., M. Rubinstein and T. Levinboim. "Generating comics from 3d interactive computer graphics." IEEE Computer Graphics and Applications 26.3 (2006): 53-61.

Simòes, A. "Story Summarization Applied to Narrative Generation Systems." (n.d.).

Sipser, M. Introduction to the Theory of Computation. 2nd Edition. Boston: International Thomson Publishing, 2006.

Tveit, A. and G.B. Tveit. "Game usage mining: Information gathering for knowledge discovery in massive multiplayer games." Proceedings of the International Conference on Internet Computing (IC'2002). CSREA Press, 2002. 636-642.

Appendix A: Definition of Pattern Language

Actions, Parameters and Variables:

`(action-type :parameter value :parameter value ...)`: Defines a pattern that maches a single action of `action-type` with the specified parameters. Action-type and parameters are game specific. A value may be a string (in quotations), a number or a variable.

Variables always begin with a question mark and a letter, such as `?PlayerA` or `?team1`.

Example: `(kill :killer ?PlayerA :victim ?PlayerB)`

Regular Expression Operators

`(concat pat1 pat2 ...)`: Defines a pattern where pat1 is matched, then pat2, etc.

`(union pat1 pat2 ...)`: Defines a pattern where pat1 matches exclusively, or pat2 matches exclusively, etc.

`(star pat)`: Defines a pattern where pat is matched zero or more times.

`(oneormore pat)`: Defines a pattern that matches pat one or more times.

`(repeat N pat)`: Defines a pattern that matches pat exactly N times.

Pattern Constraints

`(within-time T pat)`: Defines a pattern that matches only if all actions in `pat` occur within `T` seconds.

`(atleast-time T pat)`: Defines a pattern that matches only if all actions in `pat` occur over at least `T` seconds.

`(dontmatch :bad_dfa badpat :good_dfa goodpat)`: Matches `goodpat` only if `badpat` does not match over the occurrence of `goodpat`.

`(constrain C pat)`: Matches pat if the variable constraint `C` is satisfied.

Variable Constraints

`(== ?A ?B)`: Satisfied only if ?A and ?B are equal.

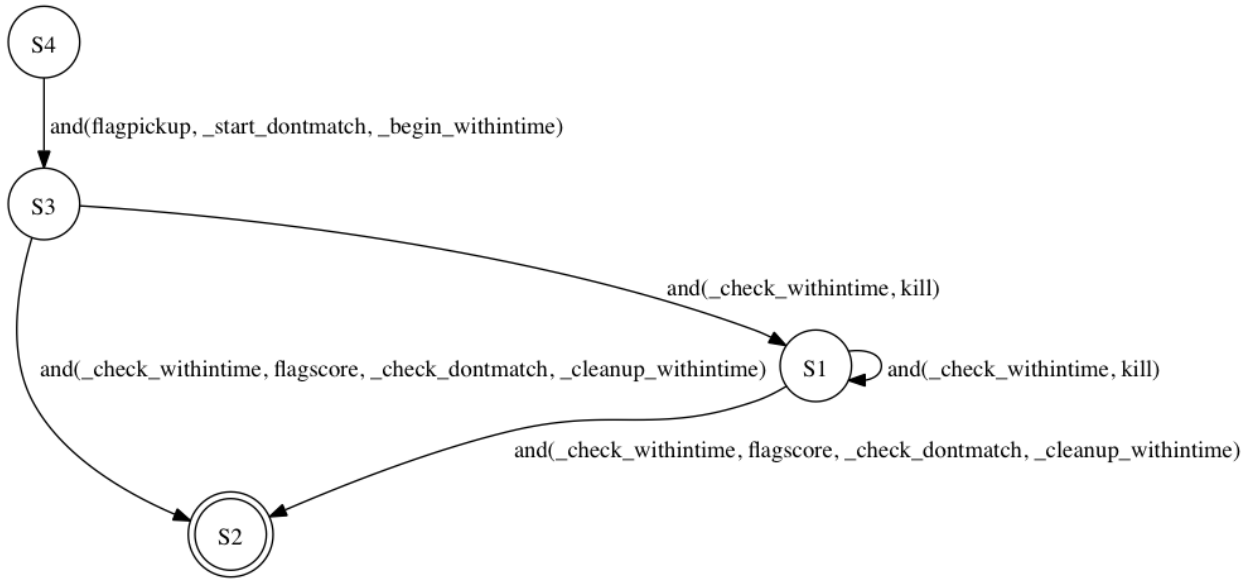
`(!= ?A ?B)`: Satisfied only if ?A and ?B are not equal.

`(and C1 C2 ...)`: Satisfied only if all variable constraints `C1`, `C2`, etc. are satisfied.

`(or C1 C2 ...)`: Satisfied if any of variable constraints `C1`, `C2`, etc. are satisfied.

Appendix B: Example Patterns and Generated FSMs

```
# Player picks up the flag, kills some players, then scores.
(pattern capturetheflag :score 100
   (within_time 90
      (dontmatch
         :bad_dfa
         # Don't want the flag to be returned at all.
         (union
            (flagreturning :flagName ?FLAGNAME)
            (flagautoreturning :flagName ?FLAGNAME)
         )
         :good_dfa
         (concat
            (flagpickup :playerPawn ?PAWNA :flagName ?FLAGNAME)
            (star (kill :killerPawn ?PAWNA))
            (flagscore :playerPawn ?PAWNA)
         )
      )
   )
)
```



```
# Player A picks up the flag and is killed by Player B.
# Player C then kills Player B, picks up the flag, and eventually scores.
(pattern continuetoscore :score 300
   (within_time 90
      (dontmatch
         :bad_dfa
            # Don't want the flag to be returned at all.
            (union
               (flagreturning :flagName ?FLAGNAME)
               (flagautoreturning :flagName ?FLAGNAME)
            )
         :good_dfa
         (concat
            # Player A picks up the flag.
            (flagpickup :playerPawn ?PAWNA :flagName ?FLAGNAME)

            # Player B kills Player A.
            (kill :killerPawn ?PAWNB :victimPawn ?PAWNA)

            # Player C kills Player B.
            (kill :killerPawn ?PAWNC :victimPawn ?PAWNB)

            # Player C picks up the flag.
            (flagpickup :playerPawn ?PAWNC :flagName ?FLAGNAME)

            # Player C scores with the flag.
            (flagscore :playerPawn ?PAWNC)))))
```