(P)NFG: A LANGUAGE AND RUNTIME SYSTEM FOR STRUCTURED COMPUTER NARRATIVES

Christopher J.F. Pickett   Clark Verbrugge   Félix Martineau
School of Computer Science, McGill University
Montréal, Canada, H3A 2A7
email: {cpicke,clump,fmarti10}@cs.mcgill.ca

KEYWORDS
Computer Games, Narratives, Petri Nets, Interactive Fiction, Formal Verification, Languages, Compilers, Interpreters

ABSTRACT

Complex computer game narratives can suffer from logical consistency and playability problems if not carefully constructed, and current, state of the art design tools do little to help analysis or ensure good narrative properties. A formally-grounded system that allows for relatively easy design and analysis is therefore desireable. We present a language and an environment for expressing game narratives based on a structured form of Petri Net, the *Narrative Flow Graph*. Our "(P)NFG" system provides a simple, high level view of narrative programming that maps onto a low level representation suitable for expressing and analysing game properties. The (P)NFG framework is demonstrated experimentally by modelling narratives based on non-trivial interactive fiction games, and integrates with the NuSMV model checker. Our system provides a necessary component for systematic analysis of computer game narratives, and lays the foundation for all-around improvements to game quality.

INTRODUCTION

A large number of computer games have strong narrative components. Most notably this includes adventure and role-playing games, but many first person shooters and 3D games also depend on a narrative backbone structure. Unfortunately, as many gamers are aware, complex narratives often contain either outright flaws or more subtly undesireable game properties [Adams, 2005]. Plot holes, non-sequiturs and narrative dead-ends are not uncommon, and difficult to avoid completely when developing a large game. A formal narrative analysis system that can help to determine these problems and otherwise analyse narratives is clearly desireable.

We initially draw on *interactive fiction* (IF) as a source of well-defined, complex narratives; IF is one of the oldest computer game genres, and provides for interactive storytelling at the most basic, fundamental level: in its most common form, the player enters text commands and receives text messages as output [Montfort, 2003]. This setting allows us to focus on "pure" narrative issues, and to separate out user interface and real time, non-deterministic gameplay concerns; it also means we are able to specify *complete* representations of many games.

We use the *Narrative Flow Graph* (NFG) as a formal structure for representing IF games [Verbrugge, 2002], and provide a new interactive NFG interpreter that allows for actual IF gameplay. NFGs are themselves a class of 1-safe Petri Nets (PNs), and thus we can exploit a wealth of available analysis research. At runtime, we feed this low level game format to the NuSMV formal model checking software [Cimatti et al., 2002] to determine game properties.

NFGs are appropriate for formal analysis, but a higher level expression is required for complex game design. We thus introduce the *Programmable NFG* (PNFG) language that accepts a high level game specification. Our language allows for easy expression of game narratives, and we have been able to produce faithful implementations of real, complex IF games relatively quickly, including the complete Scott Adams game, *The Count* [Adams, 1981]. We have also derived IF representations for the two initial chapters of *Return to Zork* [Barnett, 1993], a graphical point-and-click adventure. The PNFG compiler produces NFGs from these narratives, and we are thus able to analyse non-trivial benchmarks. Although we find that narrative complexity soon limits the analysis available, this is early work and a good baseline system for future experimentation.

Together the NFG interpreter and PNFG compiler form the (P)NFG system, and our software is freely available under the terms of the LGPL; the authors welcome feedback, bug reports, and source code contributions.

Contributions

Specific contributions of this work include:

- A new and formally-backed language for narrative specification. We give precise rules and structure for compilation of high level narrative source code to a corresponding low level representation that allows for narrative analysis.
- A Petri Net-based, interactive narrative interpreter and runtime system that integrates with the NuSMV model checker. This permits finding paths to winning and losing states, and verifying other game properties.
- Experimental data on the representation and analysis of small, medium-sized, and large actual interactive fiction narratives. Real data on non-trivial game narrative structure is of great benefit to further analysis.

In the next section, we discuss related work on narrative analysis. Subsequently, we provide a definition of our NFG formalism, slightly extended to allow for external input and output. We then give an overview of our software framework, and in the following two sections provide full details on the NFG interpreter and the intricacies of formal verification, and describe the PNFG compiler and its code generation strategies. Afterwards we present implementations of several narratives, and provide experimental results obtained using various size and complexity metrics and from our attempts at verification. Finally, we conclude and discuss future work.

RELATED WORK

Flaws in narrative construction and the corresponding need for better processes have been identified in all manner of commercial and non-commercial games [Adams, 2005]. Directed acyclic graph (DAG) representations of plotlines have been proposed as a solution to these problems several times, on `r.a.i-f` [Arnold et al., 1995], by an online IF magazine [Forman, 1997], and by the Oz group [Mateas, 1997]. IFM, the Interactive Fiction Mapper [Hutchings, 2004], is a tool that facilitates map generation and plot DAG creation by end users, and includes a solver that derives a walkthrough from task dependencies. However, DAGs most often cannot provide a complete representation as they cannot model arbitrary cycles or resource consumption.

Higher level narrative development frameworks have been explored [Brooks, 1996, Charles et al., 2002, Young, 2005], and there has also been considerable work on using logic for modelling and analysis. The language $\mathscr{E}$ provides a thorough logic-based approach to describing narratives using actions [Kakas and Miller, 1997], and narratives have also been studied as pure logic programs [Reiter, 2000]. Constraint logic programming can be used to analyse and detect flaws in story chronologies [Burg et al., 2000], and causal normalisation has been examined as a mechanism for ensuring consistency in games [Eladhari, 2002].

As a general rule of thumb, all of the interesting questions about the behaviour of 1-safe Petri Nets are PSPACE-hard [Esparza, 1998]. In order to limit the practical complexity of PN analysis we use the symbolic model verifier NuSMV [Cimatti et al., 2002], which supports a Binary Decision Diagram (BDD)-based backend [Bryant, 1992]. BDDs help to collapse the state space, making feasibile the determination of properties such as reachability for larger problem instances. NuSMV has been used to model PNs in the past [Bobbio and Horváth, 2001], techniques for encoding PNs efficiently using BDDs have been reported [Pastor et al., 2001], and recently it was suggested that clever application of brute force algorithms can be just as if not more efficient [Ciardo, 2004].

The PNFG language we will describe allows for high-level narrative descriptions to be compiled for use by our PN-based NFG interpreter. Previous work in other domains has also yielded methods for translation of languages to a PN model: a formal PN semantics has been defined for the Programmable Logic Controller (PLC) instruction list [Heiner and Menzel, 1998], and SynchNet compiles distributed object coordination specifications down to PNs [Ziaei and Agha, 2003].

This paper builds on our own previous theoretical work defining the Narrative Flow Graph (NFG) as a formal structure for computer narratives [Verbrugge, 2002]. Others have also sought to represent computer narratives using Petri Nets [Natkin and Vega, 2004], introducing several higher level control flow constructs. That work is extended in [Vega et al., 2004] to model spatiotemporal relationships in narratives using *connections* that replace edges dynamically based on transition firing patterns. Coloured PNs have also been used to model narratives in multi-agent interaction scenarios, as demonstrated through an implementation of the card trading game *Pit* [Purvis, 2004]. Finally, although not considered in the specific context of computer narratives, closely related work has seen PNs used to model relationships between tasks in workflow management [van der Aalst, 2002] and to provide a verifiable mechanism for browsing hypertext [Stotts and Furuta, 1989].

Interactive fiction authoring kits themselves have been a favourite of hobbyist programmers and many systems are available. Inform [Nelson, 2001] is one of the most popular, along with TADS [Roberts, 2005, Eve, 2005], Hugo [Tessman, 2004], ALAN [Nilsson and Forslund, 2005], ADRIFT [Wild, 2003], and Quest [Warren, 2004]. AIFT is a new Prolog-based toolkit [Merritt, 2004] inspired by previous work in using IF to teach Prolog [Merritt, 1996], and bears relation to our work in that its rule-based syntax is also amenable to formal verification.

FORMALISM

Narrative Flow Graphs (NFGs) are a class of 1-safe Peri Nets (PNs) that specify some simple abbreviations and additional markings to enforce the narrative flow, and backwards translation is straightforward. For the original Narrative Flow Graph (NFG) formalism and its derivation from 1-safe Petri Nets (PNs) and directed hypergraphs, refer to [Verbrugge, 2002]. Here we introduce a slightly revised but equivalent definition of an NFG in order to build our execution model.

Definition 1 *A* Narrative Flow Graph *(NFG) is a 6-tuple: (S, T, M, a, w, l), where S is a set of unconnected places and T is a set of transitions such that each $t = (S_s, S_c, S_d) \in T$ is connected to $S_s \subseteq S$ source places, $S_c \subseteq S$ context places, and $S_d \subseteq S$ destination places. M is the set of markings or reachable states where each $m \in M$ is a unique distribution of tokens over S, one per place s. a is an identified axiom place that connects to transitions $T_{initial} \subseteq T$ via source edges $a \rightarrow T_{initial}$ only, and w and l are identified win and lose places that connect to transitions $T_{final} \subseteq T$ via destination edges $T_{final} \rightarrow (w|l)$ only. The graph is initialized to an axiom state or marking m_a by filling the axiom place with a token. Transitions are* enabled *when all connected $s \in (S_s \cup S_c)$ for a given t contain tokens, and can thus* fire, *emptying each $s \in S_s$ and filling each $s \in S_d$; tokens are* not *removed from any $s \in S_c$. Firing is mutually exclusive: although multiple transitions may be enabled, only one fires at a time. The narrative thus flows from m_a to either the winning state m_w or the losing state m_l, via the firing of transitions, and through some intermediate set of markings $M_i \subseteq (M \setminus \{m_a, m_l, m_w\})$.*

Although NFGs as defined allow for the full semantics of goal-oriented storytelling, the details of interactivity are unclear. We now extend the original definition to include input and output connections to transitions, for use in real computer narratives or games.

Definition 2 *An* Interactive NFG *(NFG') is an 8-tuple: (S, T, M, a, w, l, I, O), where S,T,M,a,w,l are defined as before, and I and O are sets of input commands and output messages respectively, such that each $i \in I$ is attached to a single transition $t \in T$ and each $o \in O$ is attached to any number of transitions $t \in T$. An* internal transition *$t_i \in T_{internal}$ has zero input commands and can fire as soon as enabled, and an* action transition *$t_a \in T_{actions}$ has one or more equivalent input commands, and fires iff the system receives a matching input string and there is no enabled t_i. Both internal and action transitions may optionally have an output message o attached.*

Thus in an NFG', the narrative flows from axiom to win or lose states as before, but now alternates between waiting for input commands and firing series of transitions based on those commands. Output messages may be produced for the initial command or for any transition that fires as a result, as well as for the transitions that occur between m_a and the first idle state, i.e. during the narrative's prologue. We now redefine NFG to NFG'.

Previously, we also discussed several properties of narratives that can be analysed given the formal structure of an NFG. Among them are 1) *winnability* and *losability* at a given state, or the reachability of m_w and m_l; 2) the *distance* between two markings, or the shortest path between them; 3) the *separation* between two markings, or the longest acyclic path between them; 4) *pointlessness*, the separation between unwinnability and actually losing; and 5) *progress*, the distance between the current marking and m_w. In light of Definition 2, we now redefine these terms to exclude internal transitions. In this initial attempt at verification we concern ourselves only with winnability and losability and the paths to these goals.

SYSTEM OVERVIEW

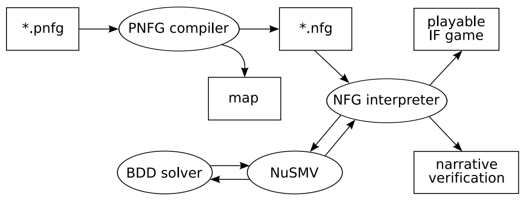

Figure 1: *System overview.*

In Figure 1, an overview of the (P)NFG system can seen. Source narratives in `.pnfg` format are fed to the PNFG compiler and can be used to produce various outputs, including a graphical map of game locations and their connectivity, and low level `.nfg` source files. Generated or handwritten `.nfg` files are then passed to the NFG interpreter which parses the `.nfg` file and creates a dynamic NFG initialized to the axiom state, and then accepts commands from the player until either the winning or losing state is reached, thus forming a playable IF game. The player can also query the interpreter as to the possibility of winning or losing, which causes the interpreter to construct a model of the NFG for the NuSMV model checker. NuSMV in turn depends on a Binary Decision Diagram (BDD) solver backend to find reachability, and ultimately a response is produced for the player.

NFG INTERPRETER

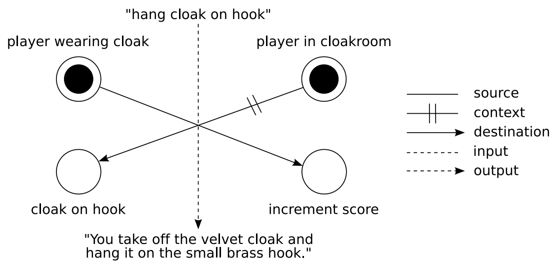

Figure 2: *Example NFG transition.*

An example NFG transition is shown in Figure 2. The player is wearing the cloak (source connection) and is in the cloakroom (context connection). The transition is enabled, since there are tokens in all source and context places. If the player inputs the command, "hang cloak on hook," the transition will fire. This removes the token from "player wearing cloak," keeps the token in "player in cloakroom," and creates new tokens for the "cloak on hook" and "increment score" destination places. Additionally, it causes the message, "You take off the velvet cloak and hang it on the small brass hook," to be printed. This is an *action transition*, as it requires user input to fire. The "increment score" place connects to a separate *internal transition* that fires as soon as enabled and without any user input.

```
transition {
   sources  = player_wearing_cloak;
   contexts = player_in_cloakroom;
   dests    = cloak_on_hook, increment_score;
   inputs   = "hang cloak on hook";
   output   = "You take off the velvet cloak and
               hang it on the small brass hook.";
}
```

Figure 3: *Example NFG source code.*

The `.nfg` source code for this transition is shown in Figure 3. The game specification is simple and consists of one axiom state, one win state, an optional lose state, and a series of transitions. Transitions are nameless, and new places are defined by using a symbol for the first time. A transition may or may not contain sources, contexts, destinations, inputs, or an output, and invalid combinations are weeded at runtime; for example, a transition with no sources or contexts must specify at least one input, for otherwise it would continually fire.

```
main() {

   build AST from .nfg input file;
   build NFG from AST;
   initialize NFG to axiom state;

   while (!won && !lost) {

      while (some $t_i \in T_{internal}$ enabled) {
         fire $t_i$;
      }

      wait for user input;

      switch (input) {
         case "query win":
            ask NuSMV to find winning state;

         case "query lose":
            ask NuSMV to find losing state;

         case "query moves":
            print each enabled $t_a \in T_{actions}$;

         case (some enabled $t_a \in T_{actions}$):
            fire $t_a$;

         default:
            "Sorry, try something else.";
      }
   }
}
```

Figure 4: *NFG interpreter main().*

Pseudocode for the NFG interpreter `main()` is shown in Figure 4. The game input file is parsed and an abstract syntax tree (AST) constructed. A traversal over the AST is used to build the NFG, and it is initialized to its axiom state. Then an event loop is entered, which iterates until either the game is won or lost. Inside the loop, first all enabled $t_i \in T_{internal}$ are fired, and this continues until an idle state is reached; the firing of one internal transition will commonly lead to the enabling of another. Once idle, a prompt is displayed, and the player can input a command. Entering “query win” will build a model for NuSMV with the invariant specification being that a winning state is not reachable; if NuSMV can find a counterexample it prints a trace, and a sequence of firing action transitions is extracted. The corresponding input commands are enumerated, thus presenting the player with a minimal winning solution. If no counterexample exists, the player is informed that winning is impossible. Similarly, “query lose” will either produce a losing solution, or inform the player that losing is impossible. Entering “query moves” does not invoke the verifier, and simply prints out input strings for each enabled $t_a \in T_{actions}$. If the player does not enter a query but an actual command, it is matched against an enabled t_a if possible and the transition is fired, otherwise a default “unrecognized command” error message is printed.

The key cost in creating the model for NuSMV, and indeed for any BDD-based verifier, is the number of boolean variables. Naïvely, places require one boolean each, but we use a token-based encoding in which we identify multiple disjoint $S_{mutex} \subseteq S$ where each S_{mutex} has a maximum of one token. Thus the cost for a S_{mutex} becomes $\lceil \log_2(|S_{mutex}| + 1) \rceil$ or $\lceil \log_2 |S_{mutex}| \rceil$ BDD booleans, depending on whether or not the set can have zero tokens. This token-based encoding becomes tedious and error-prone to do manually for large models, and we exploit the high level information available in the PNFG compiler to derive it automatically.

PNFG LANGUAGE AND COMPILER

For complex narratives, directly programming NFGs is impractical. The low level nature of NFGs can result in a large, intricate graph structure, and there is often significant code redundancy that becomes tiresome to manage, e.g. allowing multiple objects to be picked up and dropped in multiple locations. The size and unstructured complexity of the NFG graph can also be a challenge for efficient narrative analysis, and in general benefits from a higher level organisation.

The *Programmable NFG* (PNFG) format is a high level language designed to allow for easy narrative expression while maintaining a direct, efficient, and structured translation to an underlying NFG. Its syntax is close to those of standard IF toolkits, albeit with less features. A basic PNFG program structures the narrative into *object* and *room* declarations forming the core game data, and various *action* declarations implementing the game logic. Objects and rooms can contain state, counter, and timer variables, and action executions themselves are composed of sequences of commands that test, set, and act on game data. Below we discuss how data components are formed and mapped onto NFG structures, followed by the execution semantics and syntax.

Game Data

The PNFG language provides a simple, static structure for narrative game data. Concepts we now present, such as objects, rooms, state and counters are core to interactive fiction and, as we will show later, accommodate even quite complex game narratives.

Objects & Rooms

In a typical narrative game the player interacts with numerous game *objects*. An example PNFG declaration of a game object is given in Figure 5. This declaration results in a single in-game object referred to at runtime and compile-time by the name, “`cloak`.”

```
object cloak { }
```

Figure 5: *A simple object declaration.*

A further basic IF design idiom is provided by *room* declarations, and an example is shown in Figure 6. In a typical narrative, rooms are the different, discrete locations where the player and other objects can be located. In practice, rooms are merely objects that also act as containers for other objects, and in fact a player with an inventory is also modeled by a room declaration.

A strict containment hierarchy is implied by the use of rooms. An object may only be in one room at a time, and must also must be contained in some room. A special, predefined `offscreen` room with no parent container operates as a base case and initial location for all game objects.

In order to model object containment, two NFG nodes are generated for each object in each possible location. For an object A and room B a node meaning “A is in B” and a node “A is not in B” are created. Use of these nodes is subject to the strict containment property, and all transformations guarantee that when the node for “A is in B” is active all other nodes “A in C”, “A in D” and so on are inactive. This allows us to specify an S_{mutex} for each (object, room) pair containing the “object in room” and “object not in room” nodes.

Alternatively, at the expense of extra transitions, we can generate NFGs without these “not nodes”, and then use the strict containment hierarchy to identify much larger S_{mutex}’s such that the cost of each game object is $\lceil \log_2 |R| \rceil$ instead of $|R|$ boolean variables, where R is the set of rooms.

States

Rooms and objects already provide for a simple form of interaction in moving objects from one place to another, and testing for containment. *State* declarations within rooms and objects enable the user to define other, observable binary properties. Figure 6 shows a declaration for a closet which may or may not be lit, and which may or may not be locked.

```
room closet {
   state {lit, locked}
}
```

Figure 6: *A room with 2 declared binary states.*

In the NFG output graph, each state variable (for each defining object) is translated to two nodes, one for each possible value (true or false) that each state variable can have. For Figure 6, four nodes would be generated, `-closet.lit`, `+closet.lit`, `-closet.locked`, and `+closet.locked`. Pairs of nodes for a particular object and state, like the containment relation, maintain a mutual exclusion property and guarantee that exactly one will be active at any one time.

Special state nodes are used to represent winning and losing a game. The built-in object `game` has states `win` and `lose`, with the true (`+`) value of each of those states corresponding to the required NFG win and lose nodes.

Counters and Timers
Counters are used to represent countable properties of rooms or objects, and an example is shown in Figure 7. Counters behave as state variables that can be set, incremented, and decremented to any value in a defined range, and which can also be tested against an arbitrary constant.

```
room you {
   counter {lives 0 3}
}
```

Figure 7: *A counter definition for the inclusive range 0..3.*

In principle, a finite bounded counter can be implemented using just object state declarations and operations. In our current NFG output, we eliminate the "not nodes" needed in such a solution by generating simple unary counters with a single state node for each potential counter value. More efficient binary counter representations and operations are intended for future work; however, as far as verification is concerned, here the cost of a counter is $\lceil \log_2 |C| \rceil$ rather than $|C|$ boolean variables, where C is the set of mutually exclusive counter places.

Counters require programmer code to modify their value. *Timers* are merely special counters which get automatically incremented after every user action is executed.

Execution and Actions

Execution of basic interactive fiction or turn-based adventure narratives consists of first an initialisation or prologue, and then a cycle of listening and responding to user commands, and executing any automatic, internal actions, followed by a finalisation or epilogue [Montfort, 2003]. An equivalent NFG structure is produced by the PNFG compiler, and is shown in Figure 8.

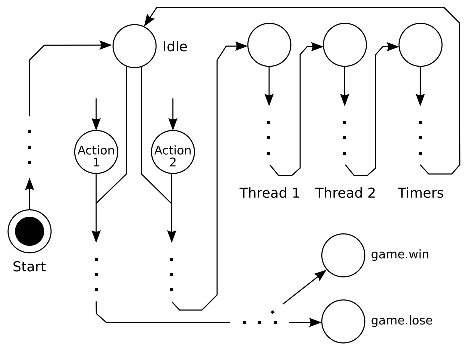

Figure 8: *The general NFG structure for a PNFG program.* The entry points for the main phases of execution are initialisation, user commands, user threads, timers, and finalisation.

A *start* node is the only initially active node; it leads to a series of initialisation activities, terminating at the main *idle* node. At this point user input is processed, activating one of the corresponding stream of actions. This will either terminate in a game win or loss, or eventually pass control to both internal and user-defined *threads*, or automatically executed sequences of instructions. Threads pass control from one to the other, and a special system thread is used to perform automatic timer updates. Finally, control returns to the idle node for another round of user input.

Actual actions consist of sequences of PNFG statements following a conventional procedural language structure. Figure 9 shows a code fragment for a "take all" command in one of the example narratives, *The Count*. Sets of rooms and game items are first defined; the room containing the player (you) is then determined, and all game items are considered. If an item is in the same room as you, then if you are not overloaded it is moved into you (your inventory), and the number of items you are carrying is increased by one. If you have reached full carrying capacity, a message to that effect is emitted instead of taking the object.

```
01  (you,take,all) {
02     places = {hall, kitchen, bedroom, ...}
03     stuff = {sheet, pillow, stake, ...}
04     places $p;
05     if ($p contains you) {
06        for (stuff $s) {
07           if ($p contains $s) {
08              if (you.overloaded) {
09                 "You are carrying too much.";
10              } else {
11                 move $s from $p to you;
12                 you.carried++;
13                 -?you.empty;
14                 if (you.carried==7) {
15                    +you.overloaded;
16  }  }  }  }  }  }
```

Figure 9: *A sequence of PNFG statements corresponding to a "take all" command.* Statements are referred to by number in the text.

Individual actions are sequenced in the NFG using a series of *context* nodes, schematically shown in Figure 10. Each action requires a unique context node as input, and must produce a unique context node on output. Context node activation defines and follows the runtime control flow, and is used to provide the PNFG execution semantics that is not otherwise guaranteed by the underlying NFGs or Petri Nets. Note that context *edges* between nodes and transitions are different: they indicate that the token is not to be removed when the transition fires.

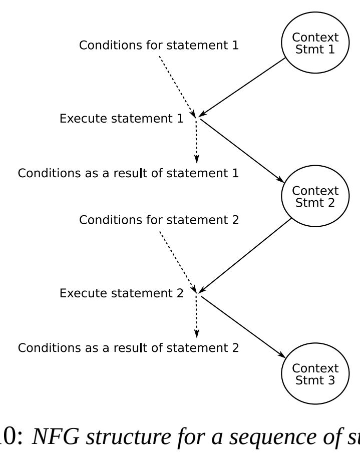
Conditions for statement 1
Context Stmt 1
Execute statement 1
Conditions as a result of statement 1
Context Stmt 2
Conditions for statement 2
Execute statement 2
Conditions as a result of statement 2
Context Stmt 3

Figure 10: *NFG structure for a sequence of statements.*

Statements
Basic PNFG statements are designed to allow easy narrative game expression while ensuring a well-defined, feasible translation to NFGs. Figure 9 illustrates the most fundamental operations, which are surprisingly few. Below we briefly describe each along with its translation to NFGs.

- *Output.* Standard text output is performed by declaring a constant string, as shown in statement 09. The NFG formulation then consists of a single edge from input to output contexts, annotated to inform the NFG interpreter to emit the specified string. These strings are sent verbatim to the console, although they could also provide canonical input to a more sophisticated output layer.

- *Move.* Basic object movement is shown in statement 11. Here one of the game items in the `stuff` set declaration, identified by the variable `$s` and found to be in the same room (`$p`) as `you`, is moved from `$p` to `you`. NFG code for a move operation is shown schematically in Figure 11, and consists of toggling the active object-location nodes appropriately.

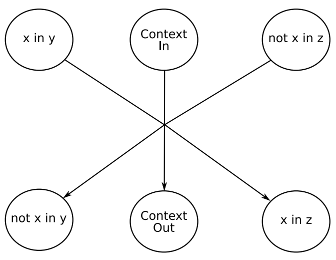

Figure 11: *NFG structure for statement,* "`move x from y to z`".

- *Set.* There are several ways to change state variables. In statement 15 the `you.overloaded` state is set to true; this is a blind operation that assumes the state is now false, and will cause control flow to stall if not. If the current state is uncertain, a *safe* set operation changes the state variable if it is not in the desired state already; an example safely setting `you.empty` to false is shown in statement 13. A *toggle* operation flips the state; these three variations are shown as NFG schemata in Figure 12.
- *Counters.* Operations on counters include incrementing and decrementing by a constant value; statement 12 shows a simple increment-by-1 of a counter, and statement 14 shows the use of counters in conditional tests. A schematic unary NFG representation for a counter update is shown in Figure 13. In general, addition or subtraction of a constant c from a counter that can assume r values generates r transitions, each trying to shift the active value node by c. End nodes must contend with potential over/underflow; here the decrement operator becomes the identity operator at the minimum counter value. More efficient math operations are left to future work.

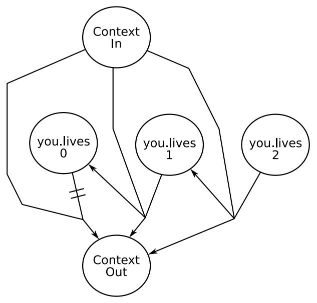

Figure 13: *NFG structure for a counter decrement statement,* "`you.lives--`".

- *If.* Conditional tests are allowed on containment, state and counter/timer values. Statement 05 for instance branches on whether the player is contained in any of the rooms in the `places` set. The NFG schema for simple if-statements is shown in Figure 14. Note that if-statements introduce distinct subsequences of statements on both the true and false branches; control flow (context activation) enters only one side, and merges with the other side to form a common output context.

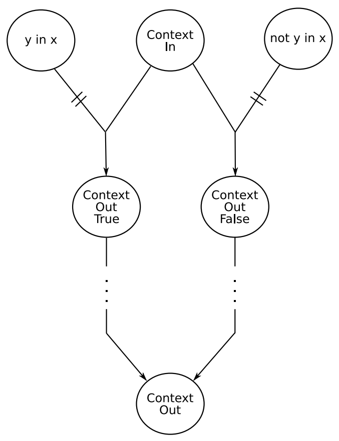

Figure 14: *NFG structure for a statement,* "`if (x contains y) {...} else {...}`". Negative containment tests ("`x !contains y`") and positive/negative state tests are structurally identical.

- *Variables & Sets.* Most operations accept object/room specifiers to be *sets* as well as specific objects; this reduces PNFG source code redundancy. For example, statement 02 declares a set called `places` consisting of the `hall`, `kitchen`, `bedroom` and so on. Statement 04 then declares an element of that set, abstractly represented as `$p`. This variable will induce replication of statements using `$p`; the if-statement of line 05, for instance, represents a collection of conditional tests and bodies, one for each object in the `places` set. The NFG schema for variable usage is shown in Figure 15. The code of lines 06–16 is replicated with the tests, each replica having `$p` bound to a distinct specific room. This can be optimised to eliminate redundancy by redirecting control flow to common subnet structures if replicas contain sequences of identical actions.

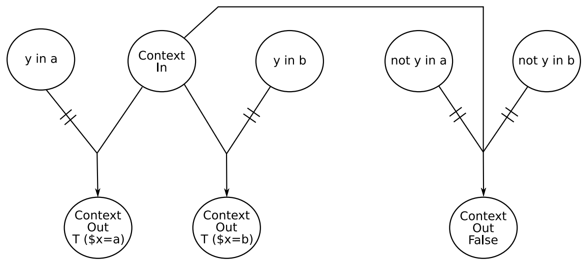

Figure 15: *Using variables.* NFG structure for a statement, "`if ($x contains y) ...`" where "`$x`" is an element of the set "`{a,b}`". Branch bodies and the following merge are not shown.

- *For.* A further use of sets is demonstrated on line 06. In contrast to an if-statement, which executes just one instance of its body even in the presence of set variables, a for-statement executes its body for all possible instantiations of the set variable. A for-statement is trivially expressed as a sequence of body executions, each with the set variable substituted by a different set element.

Actions and Threads

Commands and statements as described above can be executed during initialisation, as a response to user input, or due

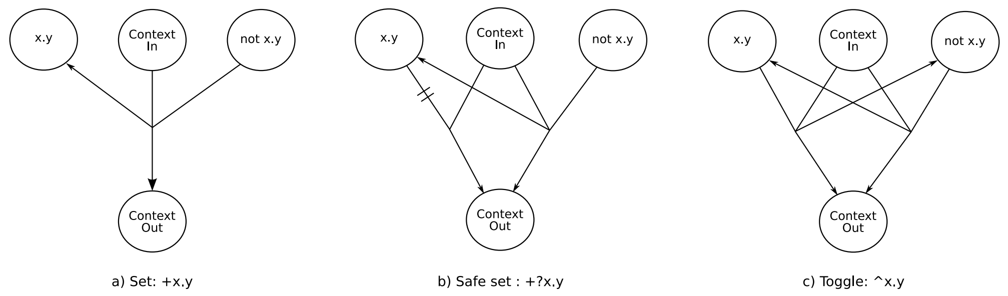

Figure 12: *NFG structure for the 3 main variations of the set statement.* Similar operations are defined for `-x.y` and `-?x,y`.

to automatic processes called *threads*. Figure 8 shows the general relationship between these three structures; here we describe how user actions and threads are defined.

User commands are specified assuming a simplified, canonical language as input. Actions are prefaced by either a (subject,verb) or (subject,verb,object) declaration; when input matching the user command declaration is received, the corresponding sequence of PNFG commands is executed. Currently the subject is always assumed to be "`you`" and ignored during NFG generation. A further syntactic convenience is provided by allowing action declarations to be defined within a room declaration as opposed to globally: this causes the action to be enabled only if the subject is in the enclosing room. Using this feature it is easy to encode the game map through room specific movement commands, as shown in Figure 16.

```
room lighthousefront {
   (you,go,north) {
      "You are now on the mountain pass.";
      move you from lighthousefront to
         mountainpass;
   }
   (you,go,east) {
      "You are now behind the lighthouse.";
      move you from lighthousefront to
         lighthouseback;
   }
   ...
}
```

Figure 16: *Room-specific actions.* These actions shadow global actions with the same user command specification, while the subject (`you`) is in the declared room.

```
thread (bomb.active) {
   if (bomb.ticksLeft==0) {
      "bang!";
      +game.lose;
   }
   bomb.ticksLeft--;
}
```

Figure 17: *A conditional thread declaration.* This thread only executes when the state `bomb.active` is true.

Threads are meant to automate actions that must be done each turn; these would otherwise have to be executed explicitly and redundantly at the end of each action. In the PNFG language threads can either execute unconditionally, or can be predicated on a conditional test, and thus act as "triggered" events. A thread modeling a timer countdown is shown in Figure 17.

In the following section we describe our experience in implementing and analysing several interactive fiction narratives expressed in our system.

EXPERIMENTAL RESULTS

Experimentation with our system at this point is largely focused on ensuring reasonable expressiveness, although we also present some preliminary work on automatic verification. We have selected one simple narrative, two medium-sized story chapters, and one complete and relatively large narrative for our investigations. Below we briefly present basic narrative properties and discuss relevant expression and verification concerns. Maps were generated by using the PNFG compiler to find actions that move `you` between rooms, and laid out using the tool `dot` [Gansner and North, 1999].

1) *Cloak of Darkness* (CoD), is a tiny game that was originally designed to demonstrate the syntax of various IF toolkits to new authors [Firth, 1999]. At the same time it provides a simple sanity check for the most ubiquitous features of IF. While there are only three rooms and three game objects, it includes basic object and room interaction, multiple commands, non-local effects, object state, and (small, finite) counting; an overview map of the game is shown in Figure 18. Expression in our system is straightforward, and data on three variations of it are given in columns 1–3 of Table 1.

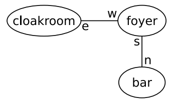

Figure 18: *Map for Cloak of Darkness.*

In *Cloak of Darkness*, the player moves between the foyer, cloakroom, and bar by issuing standard movement commands such as "go east" and "go south". The player starts off wearing a cloak, and as long as it is worn, the bar is obscured by darkness; the player can hang up his cloak on the brass hook in the cloakroom to bring light to the bar and score one point. If the player attempts any non-movement action or an invalid movement action whilst in the darkened bar, a warning message is printed and a counter is incremented. Once the player has lit the bar, it is possible to read a message in the dust on the floor. If the counter has been incremented past some predefined limit, it reads, "You have lost!", otherwise it reads, "You have won!" and the player's score goes up by another point. In either case the game ends immediately.

We first implemented *Cloak of Darkness* as an NFG, and in the absence of a parser mapped multiple variations of a command to a single action transition. We chained sequences of output messages and internal transitions together by having a destination of the first transition be a source of the second, much as context nodes order statements in NFGs generated by the PNFG compiler. We found it was necessary to duplicate a fair amount of code, and that accounting for all possibilities was error-prone. Even after extensive playtesting, NuSMV was still able to detect a subtle flaw in our implementation: the player could win simply by entering "read message" twice in the foyer at the beginning of the game. The advantage of writing IF at such a low level is that the author has direct control over the evolution of the game state; however, it is somewhat like writing in an assembly language and this soons becomes tedious. These usability factors motivated us to develop the higher-level PNFG representation. In Table 1, "CoD (full)" is more robust and "CoD (tiny)" is very minimal, retaining only the essential semantics.

Table 1: *Basic data on example narratives.* The number of BDD booleans is $\sum \lceil \log_2 |S_{mutex}(i)| \rceil$, or the sum of the logarithms of disjoint place sets that maintain a mutual exclusion property, such that there is a maximum of one token in the set at any time. The cost of verification derives from the number of BDD booleans and transitions. All `.pnfg` narratives were compiled without "not nodes".

property	CoD (tiny)	CoD (full)	CoD (full)	RTZ-1 (tiny)	RTZ-1 (full)	RTZ-2 (full)	Count (tiny)	Count (full)
source	.nfg	.nfg	.pnfg	.pnfg	.pnfg	.pnfg	.pnfg	.pnfg
rooms	3	3	4	8	10	21	4	22
objects	3	3	1	7	19	36	3	29
PNFG lines	–	–	544	347	596	1133	244	2162
places	21	69	303	366	1275	1876	272	15378
transitions	45	167	462	850	3341	8030	459	82371
BDD booleans	21	69	27	42	98	117	30	212
verifiable	yes	yes	yes	yes	no	no	yes	no
steps to win	5	6	6	5	6	22	5	180

The PNFG version is structured somewhat differently from the hand coded NFG versions. Here only the cloak is defined as a PNFG object, and the other two immobile objects are encoded through state variables and messages. An extra room is also used to encode the player's inventory ("`you`"). The resulting NFG is about three to four times as large as the hand coded version, illustrating the relative cost of a high level structure and our current, quite naïve code generation strategies. However, due to high level knowledge about mutually exclusive places, we were actually able to generate a *more* efficient model for verification that used fewer boolean variables.

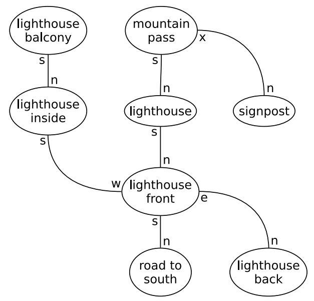

Figure 19: *Map for Return to Zork, chapter 1.*

2) *Return to Zork* (RTZ) [Barnett, 1993] is a large and complex graphical adventure, set in the same world as its pioneering IF ancestor *Zork* [Lebling et al., 1979]. We chose to translate this game to PNFG source code for two reasons. First, we wanted to demonstrate the ability of our system to model narratives that are not strictly text-based, and second, the representation of RTZ is greatly aided by the fact that it can be divided into specific chapters [Spear, 1994].

In columns 4–6 of Table 1 we show data on the first two chapters of RTZ, and in Figures 19 and 20 we show the corresponding maps. We also include a tiny variant of Chapter 1 that lacks many interactions, but that will verify due to the fewer BDD booleans and transitions. The full versions of the chapters still exceed the current capacity of our analyser; however, chapter sizes are in general within an order of magnitude of the size of a small, analysable narrative like CoD. A chapter-based division may thus be sufficient, as well as perhaps necessary for practical narrative analysis.

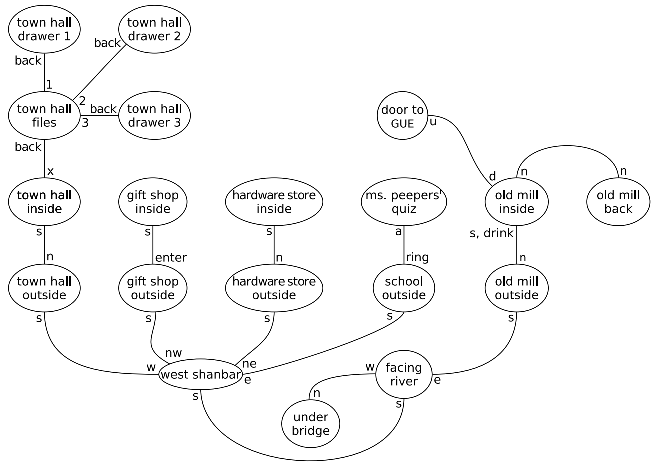

Figure 20: *Map for Return to Zork, chapter 2.*

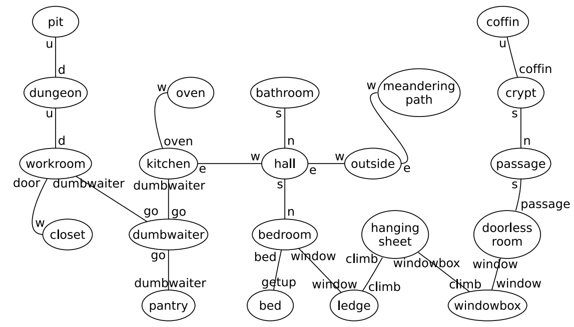

Figure 21: *Map for the full version of The Count.* Not shown is that the `sleep` command, as well as an automatic timeout at nightfall will return `you` to the bed from any room.

3) *The Count* [Adams, 1981] is one of the original Scott Adams adventure games, a great source of classic IF narratives. This game is smaller than the sum of the RTZ chapters, but at least as complex in terms of narrative structure. Here we have implemented both a partial test of the initial 4 game rooms and 3 objects (column 7), and the full version with 22 rooms and 29 objects, which includes multiple timers, counters, and user threads (column 8). The map for the full version is shown in Figure 21. The minimal solution depth for the full version is 1–2 orders of magnitude longer than those of our other narratives, giving a further indication of relative complexity.

The Count possesses a number of properties of interest to verification. At several points progress can become *pointless*, owing to loss of an essential item (e.g., the stake, cigarette) or expiration of different time limits. There are also a number of subtle story logic flaws, such as a locked closet that can be used to prevents the antagonist's access to some objects (the stake) but not others (the sheet). The latter problem is highly game specific, but in general, custom verification queries could be used within our system to check important aspects of narrative consistency.

As a complete and non-trivial game *The Count* provides a good benchmark goal for our system. At this stage it is much too large to analyse formally; however it gives a good indication as to the scale of a realistic problem space. Segmenting the narrative into separately analyseable portions may reduce the complexity, and an automatic system for doing so is part of our future work.

CONCLUSIONS & FUTURE WORK

The complexity of the structures generated for our larger narratives implies a need to significantly improve the process of verification. Use of chapters and other narrative decompositions can certainly help, as seen for *Return to Zork*, and exploring different verification strategies and Petri Net encodings can greatly affect the analysis cost. For example, the elimination of "not nodes" from our original design reduced the number of BDD booleans required for *The Count* from 890 to 212, although the number of transitions roughly doubled. Another strategy that appears quite promising is identification of individual puzzles, or groups of strongly related tasks and state variables, followed by construction of more modular, hierarchical Petri Net models.

An advantage of our system is that we can exploit the high-level PNFG structure during verification, and in general the complete PNFG→NFG→NuSMV path provides many opportunities for optimisation. It will also be interesting to analyse other properties besides winnability and losability, such as pointlessness, and to be able to provide the player with answers to questions such as, "How do I get this door unlocked?"

As far as usability of the system is concerned, we have not conducted any kind of user study outside of our own narrative development. Our system is a prototype design, and does not support many advanced programming features provided by other IF toolkits, such as object orientation, inline VM assembly, and animated graphics, or their robust standard libraries that enable parser customization, multiple world models, and NPC interaction. However, in terms of the features that (P)NFG does currently support, we find it to be comparably usable.

Furthermore, interoperability with other IF toolkits is an important goal. In this study we have translated narratives by hand to get them into our input format; a direct, automatic translation of game specifications, however, would allow us to more efficiently examine a much larger body of works [Kinder et al., 2005]. Of course, the complex syntax and details of different language specifications make this a non-trivial technical challenge.

Our system is playable as interactive fiction, but quite minimal. Adding natural language processing, a staple of most IF environments, would certainly make game play more true to the genre. This would be conceptually straightforward to add as a component that generates and reacts to our canonical input and output. Arbitrary forms of I/O could be connected similarly, allowing many multimedia effects, and expansion into other turn-based genres.

Narratives are a basic and ubiquitous component of computer games, and writing complex and error-free computer narratives is a difficult task that affects many genres, not only interactive fiction. Modern games are large, intricate software programs, and formal approaches and analyses stand to benefit both developers and players. The (P)NFG language, compiler, and runtime system provides a formal structure for narrative analysis, and helps move the burden of narrative debugging away from the play tester and into software tools.

ACKNOWLEDGEMENTS

We would like to thank Alessandro Cimatti for his help with NuSMV. This research was funded by NSERC and FQRNT.

REFERENCES

[Adams, 2005] Adams, E. (1998–2005). The designer's notebook: Bad game designer, no twinkie! parts I–VI. `http://www.gamasutra.com`.

[Adams, 1981] Adams, S. (1981). The Count. Adventure International. `http://www.msadams.com`.

[Arnold et al., 1995] Arnold, J., Baggett, D., Clements, M., Russoto, M. T., Newland, J., Plotkin, A. C., and Shiovitz, D. (1995). Game design in general. Discussion thread in `rec.arts.int-fiction` archives.

[Barnett, 1993] Barnett, D. (1993). Return to Zork. Activision Publishing, Inc.

[Bobbio and Horváth, 2001] Bobbio, A. and Horváth, A. (2001). Model checking time petri nets using NuSMV. In *Proceedings of the 5th International Workshop on Performability Modeling of Computer and Communication System (PMCC'05)*, Erlangen, Germany. Extended abstract.

[Brooks, 1996] Brooks, K. M. (1996). Do story agents use rocking chairs? The theory and implementation of one model for computational narrative. In *Proceedings of the fourth ACM International Conference on Multimedia*, pages 317–328, Boston, Massachusetts.

[Bryant, 1992] Bryant, R. E. (1992). Symbolic boolean manipulation with ordered binary-decision diagrams. *ACM Comput. Surv.*, 24(3):293–318.

[Burg et al., 2000] Burg, J., Boyle, A., and Lang, S.-D. (2000). Using constraint logic programming to analyze the chronology in "A rose for Emily". *Computers and the Humanities*, 34(4):377–392.

[Charles et al., 2002] Charles, F., Mead, S. J., and Cavazza, M. (2002). Generating dynamic storylines through characters' interactions. *International Journal of Intelligent Games & Simulation*, 1(1):5–11.

[Ciardo, 2004] Ciardo, G. (2004). Reachability set generation for petri nets: Can brute force be smart? In *Proceedings of Applications and Theory of Petri Nets 2004: 25th International Conference (ICATPN'04)*, volume 3099 of *LNCS*, pages 17–34, Bologna, Italy. Springer-Verlag.

[Cimatti et al., 2002] Cimatti, A., Clarke, E., Giunchiglia, E., Giunchiglia, F., Pistore, M., Roveri, M., Sebastiani, R., and Tacchella, A. (2002). NuSMV version 2: An opensource tool for symbolic model checking. In *Proc. International Conference on Computer-Aided Verification (CAV 2002)*, volume 2404 of *LNCS*, pages 359–364, Copenhagen, Denmark. Springer-Verlag.

[Eladhari, 2002] Eladhari, M. (2002). Object oriented story construction in story driven computer games. Master's thesis, Stockholm University.

[Esparza, 1998] Esparza, J. (1998). Decidability and complexity of petri net problems—an introduction. In *Lectures on Petri Nets I: Basic Models*, volume 1491 of *LNCS*, pages 374–428. Springer-Verlag.

[Eve, 2005] Eve, E. (2005). *Getting Started in TADS 3: A Beginner's Guide, version 3.0.8*. `http://tads.org`.

[Firth, 1999] Firth, R. (1999). Cloak of darkness. `http://www.firthworks.com/roger/cloak/`.

[Forman, 1997] Forman, C. E. (1997). Game design at the drawing board. *XYZZY News*, 4:5–11.

[Gansner and North, 1999] Gansner, E. R. and North, S. C. (1999). An open graph visualization system and its applications to software engineering. *Software—Practice and Experience*, 30(11):1203–1233.

[Heiner and Menzel, 1998] Heiner, M. and Menzel, T. (1998). A petri net semantics for the PLC language instruction list. In *Proceedings of the Fourth Workshop on Discrete Event Systems (WoDES '98)*, pages 161–172, Cagliari, Italy.

[Hutchings, 2004] Hutchings, G. (2004). *IFM: Interactive Fiction Mapper, version 5.1 manual*. `http://www.sentex.net/~dchapes/ifm/`.

[Kakas and Miller, 1997] Kakas, A. C. and Miller, R. (1997). A simple declarative language for describing narratives with actions. *Journal of Logic Programming*, 31(1-3):157–200.

[Kinder et al., 2005] Kinder, D., Granade, S., Blasius, V., and Baggett, D. M. (2005). The interactive fiction archive. `http://www.ifarchive.org`.

[Lebling et al., 1979] Lebling, P. D., Blank, M. S., and Anderson, T. A. (1979). Zork: A computerized fantasy simulation game. *IEEE Computer*, 12(4):51–59.

[Mateas, 1997] Mateas, M. (1997). An Oz-centric review of interactive drama and believable agents. Technical Report CMU-CS-97-156, School of Computer Science, Carnegie Mellon University.

[Merritt, 1996] Merritt, D. (1996). *Adventure in Prolog*. Amzi! Inc., 5861 Greentree Road, Lebanon, Ohio 45036 USA.

[Merritt, 2004] Merritt, D. (2004). AIFT: Amzi! Interactive Fiction Toolkit. `http://www.ainewsletter.com/downloads/if_docs/`.

[Montfort, 2003] Montfort, N. (2003). *Twisty Little Passages*. The MIT Press.

[Natkin and Vega, 2004] Natkin, S. and Vega, L. (2004). A petri net model for computer games analysis. *International Journal of Intelligent Games & Simulation*, 3(1):37–44.

[Nelson, 2001] Nelson, G. (2001). *The Inform Designer's Manual*. The Interactive Fiction Library, PO Box 3304, St Charles, Illinois 60174, USA, 4th edition.

[Nilsson and Forslund, 2005] Nilsson, T. and Forslund, G. (2005). *The ALAN Adventure Language, version 3.0dev36 manual*. `http://www.alanif.se`.

[Pastor et al., 2001] Pastor, E., Cortadella, J., and Roig, O. (2001). Symbolic analysis of bounded petri nets. *IEEE Transactions on Computers*, 50(5):432–448.

[Purvis, 2004] Purvis, M. K. (2004). Narrative structures for multi-agent interaction. In *2004 IEEE/WIC/ACM International Conference on Intelligent Agent Technology (IAT 2004)*, pages 232–238, Beijing, China. IEEE Computer Society.

[Reiter, 2000] Reiter, R. (2000). Narratives as programs. In Cohn, A. G., Giunchiglia, F., and Selman, B., editors, *KR2000: Principles of Knowledge Representation and Reasoning*, pages 99–108, San Francisco. Morgan Kaufmann.

[Roberts, 2005] Roberts, M. J. (1987–2005). TADS: The Text Adventure Development System. `http://tads.org`.

[Spear, 1994] Spear, P. (1994). *Return to Zork - The Official Guide to the Great Underground Empire*. BradyGames.

[Stotts and Furuta, 1989] Stotts, P. D. and Furuta, R. (1989). Petri-net-based hypertext: document structure with browsing semantics. *ACM Transactions on Information Systems (TOIS)*, 7(1):3–29.

[Tessman, 2004] Tessman, K. (2004). *The Hugo Book—Hugo: An Interactive Fiction Design System*. The General Coffee Company Film Productions, Toronto, Canada, 1st edition. `http://www.generalcoffee.com`.

[van der Aalst, 2002] van der Aalst, W. (2002). *Workflow Management: Models, Methods, and Systems (Cooperative Information Systems)*. The MIT Press.

[Vega et al., 2004] Vega, L., Grünvogel, S. M., and Natkin, S. (2004). A new methodology for spatiotemporal game design. In *Proceedings of the CGAIDE'2004, Fifth Game-On International Conference on Computer Games: Artificial Intelligence, Design and Education*, pages 109–113.

[Verbrugge, 2002] Verbrugge, C. (2002). A structure for modern computer narratives. In *CG'2002: International Conference on Computers and Games*, volume 2883 of *LNCS*, pages 308–325.

[Warren, 2004] Warren, A. (2004). *Quest Documentation, version 3.51*. Axe Software. `http://www.axeuk.com/quest/`.

[Wild, 2003] Wild, C. (2003). *ADRIFT: Adventure Development & Runner—Interactive Fiction Toolkit, version 4.0 manual*. `http://www.adrift.org.uk`.

[Young, 2005] Young, R. M. (2005). Cognitive and computational models in interactive narrative. In Forsythe, C., Bernard, M. L., and Goldsmith, T. E., editors, *Cognitive Systems: Human Cognitive Models in Systems Design*. Lawrence Erlbaum. To appear.

[Ziaei and Agha, 2003] Ziaei, R. and Agha, G. (2003). SynchNet: A petri net based coordination language for distributed objects. In *GPCE '03: Proceedings of the second international conference on Generative programming and component engineering*, volume 2830 of *LNCS*, pages 324–343.