Formal Verification of Computer Narratives

GR@M Technical Report No. 2005-1

Christopher J. F. Pickett
cpicke@cs.mcgill.ca

This was the final report for COMP-525, a formal verification course taken in the Fall 2005 semester.

December 20th, 2005

Formal Verification of Computer Narratives

Christopher J. F. Pickett

December 20th, 2005

Abstract

We consider modern computer narratives as deterministic finite automata with identified initial, winning, and losing states. We discuss various interesting temporal properties in the context of narratives, which range from simple reachability to more complex concepts such as the degree of "pointlessness". These temporal properties can be useful in model checking narratives, but first an appropriate representation is needed. We introduce the *Narrative Flow Graph* (NFG), a class of 1-safe Petri Net, suitable for modelling computer narratives compactly, and the higher-level *Programmable NFG* (PNFG) language that compiles down to NFGs. We show how NFGs can be automatically encoded as transition relations for the NuSMV model checker, and use high-level knowledge from the PNFG language to reduce the number of boolean variables required. We finally provide an experimental analysis of small, medium, and large narratives.

1   Introduction

Modern computer games typically have some kind of narrative backbone providing structure to the gameplay[1]. Unfortunately, plot holes, non-sequiturs, and dead ends abound in the narratives of even massively commercial games. Typically a company will "playtest" a product for several months before release, with the hope of finding at least all of the obvious bugs. However, after release players will invariably find inconsistencies that the games company missed, sometimes so many that dowloadable patches must be issued, especially if some unforseen flaw makes progress impossible for certain players and narrative states. It would be nice to apply formal verification techniques to prevent against these problems.

There are four classes of people that stand to benefit from this work. Game designers, particularly those involved in story construction, would be able to ensure certain narrative properties. Large numbers of playtesters would be liberated from taxing and mundane jobs, able to move on to more meaningful and rewarding work in the burgeoning games industry. Game players would be end up with higher quality products with fewer bugs, and also be able to make queries about the current game state upon getting stuck. Finally, computer scientists would benefit, for not only do they like solving and improving upon solutions to hard problems, but ties between industry and academia would be strengthened, which is something of a weak point in gaming.

Our initial strategy to address the sound narrative construction problem will be to formalise some desired temporal properties of computer narratives, represent these narratives as finite state machines, and then generate models and attempt to check our temporal properties automatically.

1.1   Computer Narratives

A sample computer narrative transcript is shown in Figure 1.1. The player inputs commands, and receives output messages in reply. Computer narratives that consists only of text input and output are known as

---

[1]This excludes combinatorial two-player games such as chess that have traditionally been the domain of "games research".

Interactive Fiction (IF) [17], and will be the focus of our initial efforts for two important reasons. First, there exists a small but consistent amateur IF community exists today that is able to evaluate and provide useful feedback on our techniques immediately and with low overhead. Second, using only the simplest of programming environments—ones without concurrency, graphics, audio, real-time interaction, or non-determinism—we are readily able to provide *complete* and indeed interactively playable representations of a considerable number of IF narratives.

```
> look
It's pitch black, and you can't see a thing!

> inventory
You are carrying:
  a lamp

> light lamp
The lamp flickers to life.

> look
You're in a musty old cellar.  Prakash is here.
```

Figure 1: *Computer narrative transcript.*

The rest of this paper is organized as follows. In Section 2 we formalise some interesting temporal properties in computer narratives. In Section 3 we demonstrate how narratives can be represented as *Narrative Flow Graphs*, a class of 1-safe Petri Net, and briefly introduce a high-level programming language for generating these graphs. In Section 4 we discuss encoding these graphs as input models to the NuSMV [6] model checker, the benefits of a high-level structure, and experimental data for several narratives. We finally discuss related efforts in Section 5, and conclude and present future work in Section 6.

2   Interesting Temporal Properties

We limit our study to computer narratives that can be expressed as deterministic finite automata, with provision for two special states. The *lose* state is reached when the player has reached a non-winning end of narrative, and the *win* state is reached when the player has completed the narrative successfully. These are both sink nodes, and the only transitions out of them are self-loops. We discuss the details of our particular representation in the next section, along with I/O mechanisms and an execution model. In this section we present interesting temporal properties in computer narratives, as previously identified [28]. Building on this previous work, we also describe how formal methods can be used to determine them, where possible.

2.1   Reachability

The most basic temporal property of computer narratives is *reachability*. Given either the initial game state or the current game state s, one would like to know if there exists a path to some future state t. This can be a binary yes/no question, and is answered by checking whether $\mathcal{M}, s \models \mathrm{EF}t$. Alternatively, if at least one exists, we can find an actual path between s and t with the counterquery $\mathcal{M}, s \models \neg\mathrm{EF}t$. Some interesting reachability queries in the context of computer narratives might be:

- Can I lose? $\mathcal{M}, s_{current} \models \mathrm{EF}lose$

- How do I win? $\mathcal{M}, s_{current} \models \neg\mathrm{EF}win$

- How do I unlock this door? $\mathcal{M}, s_{current} \models \neg \text{EF} door.unlocked$

- Are there any zombie states: $\mathcal{M}, s_{initial} \models \neg \text{AG}(\text{EF} win \vee \text{EF} lose)$

A zombie state is one where the player is literally left to roam around without being able to reach either win or $lose$. A game designer might well want to assert the absence of such states, and players could be spared frustration if after a fixed number of turns they were notified that they were indeed amongst the living dead, as it were.

2.2 Distance

The next most obvious temporal property is that of *distance* $\delta(s, t)$, or the shortest path from s to t.

- $\delta(s, t) = \min\{n | \exists s_0, \ldots, s_n . s = s_0, t = s_n, s_i \rightarrow s_{i+1} \text{ for } 0 \leq i < n\}$

However, $\delta(s, t)$ is just length of the reachability counterexample produced by $\mathcal{M}, s \models \neg \text{EF} t$, since BFS is commonly if not always used by model checking software.

The concept of distance is distorted when internal transitions that are not guarded by user input commands are introduced into the model. It is now possible for a shortest path to require more input transitions than an arbitrarily longer path, and thus appear longer to the player or external world. This leads naturally to the concept of parameterized distance, which is determined by the minimal change in some scalar variable along a path. For example, one might want to ask:

- What is the cheapest way to reach state s (in zorkmids)?

- What is the fastest way to reach state s (in game turns)?

- What is the fastest way to reach state s (in commands entered)?

Obtaining this parameterized distance would involve modifying model checking algorithms to use a hybrid DFS–BFS that uses DFS when the current node does not modify the variable under consideration and BFS when it does modify it. Such a strategy could even conceivably be useful outside of narrative verification.

2.3 Separation

Alternatively we might be interested in *separation* $\gamma(s, t)$, or the longest acyclic path from s to t.

- $\gamma(s, t) = \max\{n | \exists s_0, \ldots, s_n . s = s_0, t = s_n, s_i \rightarrow s_{i+1} \wedge (\forall i \forall j. i \neq j \rightarrow s_i \neq s_j) \text{ for } 0 \leq i < n\}$

However, it is not obvious how $\gamma(s, t)$ can be efficiently checked in general. In fact, finding a longest simple path as such is a known NP-complete problem [12].

2.4 Reoccurrence Radius

We now consider the reoccurrence radius $\rho(s)$, or the longest acyclic path from s to any other reachable state t.

- $\rho(s) = \max\{\gamma(s, t) | t \in S \wedge \mathcal{M}, s \models \text{EF} t\}$

We can also compute $\rho(s)$ by finding a minimal r that makes the following formula valid:

- $\forall s_0, \ldots, s_{r+1}.((s_0 = s_{initial}) \wedge (s_i \rightarrow s_{i+1} \text{ for } 0 \leq i \leq r)) \Rightarrow \exists i \exists j.(0 \leq i < j \leq r + 1) \wedge (s_i = s_j)$

The intuition behind this minimization is that we want to find the shortest path length that always implies a cycle exists. Then, one transition short of this length is the longest acyclic path length. This is a propositional formula, and can be checked by SAT [15]. We shall now see an application of $\rho(s)$ to narrative verification.

2.5 Pointlessness

We say that a game is *pointless* if $\mathcal{M}, s_{current} \models \neg\text{EF}win$. Furthermore, we can quantify the degree of pointlessness: a game is p-pointless iff it takes a maximum of p steps from pointlessness to actually losing. In order to calculate p, we might try:

- $p = \gamma(s_{pointless}, lose)$

Now, if $\mathcal{M}, s_{pointless} \models \neg\text{AG}(\text{EF}lose)$, then $p = \infty$, because the system can enter a zombie state where neither win nor $lose$ is reachable. Otherwise, $p = \rho(s_{pointless})$. Proof: $lose$ is a sink node by definition, and reachable on all paths since $p \neq \infty$, so any putative acyclic longest path found by $\rho(s_{pointless})$ not including $lose$ can be extended to include $lose$. Thus, although the problem of separation remains NP-complete, we can use an efficient SAT solver to answer the separation question for the specific problem instance of pointlessness.

3 Representing Narratives

In order to check for temporal properties in computer narratives, a specific finite state machine representation is needed. The sheer size of narratives under consideration precludes using automata with one node per state. *Petri Nets* in fact provide a compact representation, and we adopt their structure in our definition of *Narrative Flow Graphs* [28, 20], with some slight modifications to ensure the right semantics for interactive narration. A high-level representation is also desired from a programmability perspective, and at the end of this section we introduce the *Programmable Narrative Flow Graph* language and compiler [20].

3.1 Petri Nets

A Petri Net is a kind of finite state machine that is widely used in the modelling and simulation of concurrent processes. In our current work, we ignore that this is the natural application domain, and instead exploit their compact structure to model large, single-process systems at a low level. We will consider *1-safe* Petri Nets, for which the following two rules of thumb hold [9][2]:

1. All interesting questions about the behaviour of 1-safe Petri Nets are PSPACE-hard.

2. Nearly all interesting questions about the behaviour of 1-safe Petri Nets can be decided in polynomial space.

A Petri Net is a 4-tuple (S, T, F, M_0):

- S is a set of *places*

- T is a set of *transitions*

- $F \subseteq (S \times T) \cup (T \times S)$ is a *flow relation* such that no arc connects two places or two transitions

- $M_0 : S \to \mathbf{N}$ is an initial *marking*, or distribution of *tokens* over $s \in S$.

Petri Net markings are the states of the system, and a 1-safe Petri Net only ever has 0 or 1 tokens in any place, for all markings. A transition $t \in T$ is *enabled* if all source places and no destination places contain tokens (these particular enabling semantics ensure the net is *contact-free*), and can then *fire*, removing tokens from all source places and filling all destination places, as shown in Figure 3.1. The firing of transitions determines of reachability of new markings starting from M_0.

---

[2]This paper is recommended reading: not only is it entertaining but it covers complexity issues in model checking Petri Nets in considerable detail.
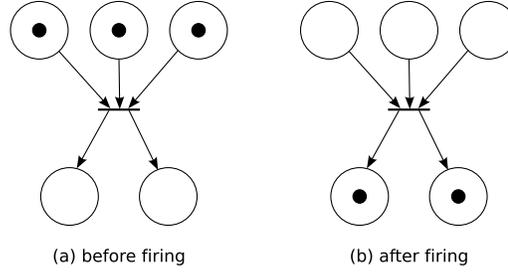
(a) before firing   **(b) after firing**

Figure 2: *A 1-safe Petri Net transition with three source places and two destination places.* In (a) the transition is enabled, as each source place contains a token. In (b) the transition has fired, each source place having been emptied of its token and each destination place filled.

3.2 Narrative Flow Graphs

In order to provide a framework for representing interactive computer narratives, we extend (or restrict, depending on perspective) 1-safe Petri Nets to *Narrative Flow Graphs* (NFGs) by specifying the following additional elements:

- M_{win}, a sink marking that signifies winning

- M_{lose}, a sink marking that signifies losing

- L, a set of labels

- $I : T \leftrightarrow L$, a one-to-one mapping between transitions and input labels

- $O : T \rightarrow L$, a many-to-one mapping between transitions and output labels

- $T_{actions} : t \in T \; . \; I(t) \in L$, action transitions

- $T_{internal} : t \in T \; . \; I(t) \notin L$, internal transitions

We say that the narrative flows from M_0 to M_{win} or M_{lose} via some series of $t_a \in T_{actions}$ and $t_i \in T_{internal}$. During execution, internal transitions take priority, and firing is sequential. An example action sequence is shown in Figure 3.2.

In Figure 3.2, there is a `carrying_lamp` node connected by a double-headed arrow to the action transition, and is both a source and destination place. Although such nodes break the property of contact-freeness required by our Petri Net definition, we do in fact allow them, for there exists a straightforward translation to a contact-free net via introduction of another internal transition that simply moves the token from a now distinct destination back to the source place.

3.3 NFG Execution

Narrative Flow Graphs can be interpreted, providing for actual interactive gameplay or storytelling. The execution model used by our current interpreter is given in Figure 3.3.

An in-memory NFG is built from a text file, and initialized to the starting state M_0. Then, as long as neither M_{win} nor M_{lose} is satisfied, the system alternates between firing enabled internal transitions, waiting for user input, and matching user input to some action. If the input is "query win" or "query lose", the interpreter will ask NuSMV [6] to check unreachability of the desired state. If a counterexample is produced then a solution consisting of necessary action transitions is printed, otherwise the player is informed that the
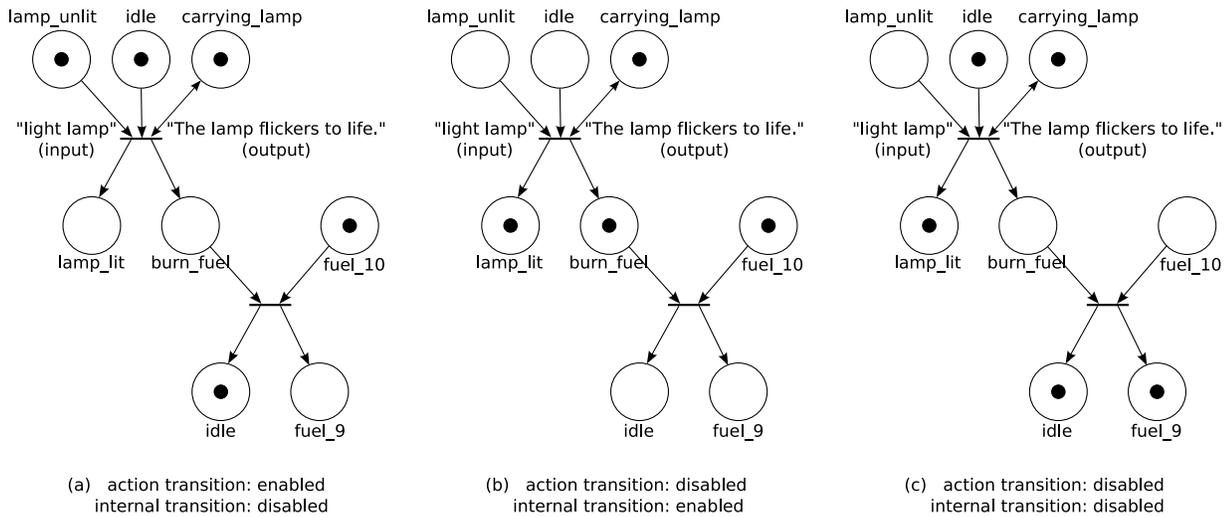
Figure 3: *A sequence of two NFG transitions corresponding to the action "light lamp".* In (a) the action transition is enabled, and will fire when the system receives a matching input string, outputting a message. In (b) the action transition has fired, and an internal transition that silenty consumes some fuel is enabled. In (c) the internal transition has fired and the system is idle again. The two `idle` places in the figure are actually a single place in the model.

```
initialize NFG to M_0
while (M_NFG, M_current ⊭ M_win ∧ M_NFG, M_current ⊭ M_lose)
    while (some t_i ∈ T_internal enabled)
        fire t_i
    wait for user input
    switch (input)
        case "query win":
            M_NFG, M_current ⊨ ¬EF M_win
        case "query lose":
            M_NFG, M_current ⊨ ¬EF M_lose
        case "query moves":
            print I(t_a) for each enabled t_a ∈ T_actions
        case I(t_a) for some enabled t_a ∈ T_actions:
            fire t_a
        default:
            "Sorry, try something else."
```

Figure 4: *NFG interpreter execution model.*

6

state asked for is indeed unreachable. If the input is "query moves", all enabled action transitions will simply be listed without any need for model checking. If the input matches an action transition, that transition will fire, and after subsequent internal transitions have fired the system will again be idle; otherwise an error message is produced. Although output labels associated with transitions are sent immediately to the console upon firing, they could instead be used to drive graphics and audio subsystems; similarly, one could imagine non-keyboard input devices being used to generate input strings.

3.4 Programmable Narrative Flow Graphs

Although NFGs are suitable for interpreted execution and verification purposes, it is tedious to construct them manually, for it is akin to programming in an assembly language. To this end, we introduced the imperative *Programmable Narrative Flow Graph* (PNFG) language and compiler [20]. This language is similar to that provided by other popular narrative authoring toolkits, such as Inform [19] and TADS [23], with the added benefit that it compiles to a simple, low-level NFG finite state machine.

High-level constructs such as rooms, objects, sets, counters, booleans, functions, user actions, global and local scopes, conditional branches, statement sequences, and output messages are provided. There is no loss of expressive power in the abstraction, at least assuming the NFG execution model previously described, however the generated NFG code is naturally more verbose than what a human might create. A sample of PNFG source code is given and described in Figure 3.4.

```
object lamp {
   state { lit }
   counter { fuel 0 10 }
}

room cellar {
   (you, look) {
      if (you contains lamp && lamp.lit) {
         "You're in a musty old cellar.";
         lamp.fuel--;
      } else {
         "It's pitch black, and you can't see a thing!";
      }
   }
   ...
}
```

Figure 5: *PNFG source code fragment.* Here a `lamp` object is defined, which has a `lit` state, and a `fuel` counter ranging from 0 to 10. There is a `cellar` room, and shown is one of the actions associated with this room, `(you, look)`. The conditional tests if the player `(you)` is carrying the `lamp` and if the `lamp` is `lit`. If so, an output message is printed, and the `lamp.fuel` counter is decremented. Otherwise, a different output message is printed.

The language accepted and compilation strategies used by the PNFG compiler are described in detail in [20]. For the purposes of formal verification, the primary benefit of a high-level source language is not usability, but in fact that we can generate more efficient NFG models, as we shall see in the next section.

4 Symbolic Modelling and Verification

The ultimate goal of this work is automated analysis of complex computer narratives that are comparable in scope to those used in the games industry. Although all of the temporal properties described in Section 2 are theoretically interesting, there is a need to demonstrate viability on examples large enough for real-world narrative authors to take notice. Thus for now we focus on simple reachability queries in Narrative Flow Graphs, and our primary concerns are generating efficient NuSMV [6] models and optimising the output of the PNFG compiler. Reachability questions arise frequently amongst game designers, game testers, and game players, and are certainly worth answering in their own right.

4.1 Generating NuSMV Models

NuSMV models are generated and verified at runtime when the player enters a reachability query. M_0 is initialized to the current in-memory marking, NuSMV builds and checks the model according to the CTL query specification, and either a binary yes/no answer or a counterexample is produced. In the event of a counterexample, action transitions are extracted from the trace, and the labels $I(t_a)$ printed in order as a solution path.

Translation from an NFG or Petri Net to the transition relation format required by NuSMV is straightforward, as is expressing (un)reachability specifications. At first, it seemed appropriate to use one boolean in the model for each place in the NFG, representing whether or not that place contained a token. Indeed, this was sufficient for verification of a small hand-coded example with 69 places and 167 transitions, as shown in Section 4.3. However, this technique does not scale well, and in the next subsection we present a more efficient alternative encoding.

An input variable is used to select the next transition. Naïvely, this variable ranges over all transitions, even though the vast majority of them are disabled and cannot fire for any given program state. Limiting the size of the input variable range to the maximum number of transitions that might fire for a given node in the PNFG control flow graph would not actually improve results, as this maximum is also quite high. There is a possibility that modifying NuSMV to accept a different input variable range depending on the current model state might help significantly.

NuSMV exhibits a high sensitivity to variable ordering. Even for the same input model, small examples might quickly exhaust available memory, run for many hours without any progress, or produce a solution within a few minutes. Experimentation with NuSMV's variable ordering options did not help, and it seems likely that some heuristics for a fixed ordering based on high-level PNFG knowledge are necessary.

4.2 Exploiting PNFG Structure

Given only the knowledge from low-level NFG structures, it is difficult to create a NuSMV encoding much more efficient than one that uses one boolean per place. However, minimizing the number of booleans needed by the model is critical, as having fewer booleans simplifies the construction and manipulation of OBDD's by NuSMV.

Fortunately, some of the code generation strategies used by the PNFG compiler allow us to generate more efficient NFG representations. In particular, we can identify multiple disjoint $S_{mutex} \subseteq S$ that can only ever contain one token at a time. In our original place-based enocoding, $|S_{mutex}|$ booleans would be needed to represent each such set, but if we switch to a token-based encoding, only $\lceil \log_2 |S_{mutex}| \rceil$ booleans are needed.

There are three main applications of this technique with respect to high-level PNFG constructs. Narratives define a set of rooms R and a set of objects O. Each object $o \in O$ can be moved between rooms, but can only exist in one room at time; thus, there is an S_{mutex} for each object representing its current location in

R, and the cost per object is $\lceil \log_2 |R| \rceil$ booleans. Given 20 rooms and 20 objects, this reduces the modelling cost from $20 \times 20 = 400$ booleans to $5 \times 20 = 100$ booleans.

Second, PNFG provides counters, or integer scalars, that are compiled to NFGs using a unary representation. This is highly inefficient under a place-based encoding, as one boolean is needed for each possible counter value, but since a counter can only assume one value at a time, only $\lceil \log_2 |C| \rceil$ booleans are needed per counter, where C is the set of possible counter values. Thus our naïve unary translation becomes as efficient as a proper binary encoding for purposes of model checking.

Third, control flow in the imperative PNFG language is sequenced using chains of *context* places that correspond to the nodes of a control flow graph. In our current sequential execution model, these are naturally mutually exclusive, and the cost of modelling the `pc` is $\lceil \log_2 |P| \rceil$ where P is the number of CFG nodes. We will need to reconsider this for language and interpreter extensions that allow for concurrent execution; it might for instance then be appropriate to construct a model with one token per process.

The effect of these reductions is shown in the following section. Further improvements are a part of future work, and will involve providing more high-level structures to the programmer so that a better encoding of the same narrative can be achieved.

4.3   Experimental Results

We used three existing computer narratives as a source of benchmark data. *Cloak of Darkness* [10] is a simple proof-of-concept narrative that has been implemented using many different authoring toolkits. Chapters 1 and 2 of *Return to Zork* are the initial two segments of a much larger graphical adventure, according to the partitioning suggested in [24]. *The Count* is a classic text adventure from 1981 [1], and is significantly longer and larger than the other narratives.

narrative title	Cloak of Darkness	Cloak of Darkness	Return to Zork Ch. 1	Return to Zork Ch. 2	The Count
source format	.nfg	.pnfg	.pnfg	.pnfg	.pnfg
PNFG linecount	–	544	596	1133	2162
rooms	3	4	10	21	22
objects	3	1	19	36	29
transitions	167	462	3341	8030	82371
places	69	303	1275	1876	15378
BDD booleans	69	27	98	117	212
verifiable	yes	yes	no	no	no
steps to win	6	6	6	22	180

Table 1: *Experimental data on narrative verification.* "source" indicates whether the high-level PNFG language or low-level NFG format was used to express the narrative. "rooms" and "objects" are basic narrative structures that the player interacts with. "transitions" and "places" are the transitions and places in the final NFG, and "BDD booleans" is the number of booleans needed in the NFG model produced for NuSMV, before variable ordering and BDD reduction. "verifiable" indicates whether basic reachability queries could be answered, and "steps to win" is an upper bound on the minimum number of action transitions that must fire on the path $M_0 \rightarrow \ldots \rightarrow M_{win}$.

Basic experimental data on narrative verification are given in Table 4.3. There are three main conclusions that can be drawn from the data. First, the NFG representation of narratives appears to grow exponentially in size, with 2–3 orders of magnitude difference in place and transition counts between small and large

narratives, while the PNFG representation grows fairly linearly. This suggests the second conclusion, which is that scalability is an issue; here we are only able to answer reachability queries for small narratives with a few rooms and objects and a shallow solution depth.

In turn we have the third point, which is that representing narratives in the PNFG language and exploiting the available optimisations can have a huge impact. As far as the number of input BDD booleans required for verification is concerned, the PNFG version of *Cloak of Darkness* is actually more efficient than the hand-coded NFG version, due to the token-based encoding and S_{mutex} reductions described in Section 4.2. With respect to the larger narratives, without the high-level knowledge available from the PNFG compiler, there would be absolutely no hope in verifying them on current systems: in a naïve place-based encoding, *The Count* would require 15378 booleans. Although theoretically these mutexes could be identified for hand-coded NFGs, the error and tedium in doing so makes it highly impractical.

5   Related Work

Our work here is based on previously defining NFGs and the PNFG language for computer narratives [28, 20]. There have been few other attempts to formalise computer narratives rigourously, and as far as we know none that have attempted to use model checking software.

Directed acyclic graph (DAG) representations of plotlines have been proposed as a solution to narrative construction problems several times, on r.a.i-f [2], by an online IF magazine [11], and by the Oz group [16]. IFM, the Interactive Fiction Mapper [13], 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 [3, 5], and there has also been considerable work on using logic for modelling and analysis. The language $\mathcal{E}$ provides a thorough logic-based approach to describing narratives using actions [14], and narratives have also been studied as pure logic programs [22]. Constraint logic programming can be used to analyse and detect flaws in story chronologies [4], and causal normalisation has been examined as a mechanism for ensuring consistency in games [8].

Others have also sought to represent computer narratives using Petri Nets [18], introducing several higher level control flow constructs. That work is extended in [27] 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* [21]. 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 [26] and to provide a verifiable mechanism for browsing hypertext [25].

6   Conclusions and Future Work

We have seen that there are several interesting temporal properties in computer narratives represented using finite state machines, and that some of these can be expressed in a form amenable to formal verification. A class of Petri Net, the Narrative Flow Graph, is suitable for representing the large state space of these systems efficiently, and works to provide interactive gameplay as well as a structure for model checking by NuSMV. That experimental results at this time are limited to smaller narratives due to a state space explosion problem supports the existing consensus that software verification is hard, but we also see that exploiting the structure built into a high-level programming language environment can reduce the cost significantly.

As for future work, our primary goal is still verification of larger problem instances. However, this may require more aggressive techniques than simply improving the model encoding at the level of the NFG

format. Work is underway to provide the PNFG compiler with a full suite of optimisations, and initial support for dataflow analysis and code commoning is already present. In addition, we might be able to identify certain reusable state bits in the generated models. For example, if for any two objects in a narrative, their state bits can be proven readable and writable in two distinct, mutually exclusive time frames, it might be possible to reuse these bits in different contexts. From a usability perspective, it would also be nice to extend the PNFG compiler to accept LTL and CTL specifications, both in their natural form and according to the patterns of specification given in [7].

We would also like to check other temporal properties besides reachability, in particular the degree of pointlessness property explained in Section 2. This seems quite feasible, at least for small problem instances, it simply remains to express the query for our NFG models in a form that existing SAT solvers such as `zChaff` can understand.

This work clarifies some of the issues surrounding verification of computer narratives, and provides a useful starting point for more detailed investigations. In particular, we have a high-level language, compiler, runtime system, model generation scheme and interface with popular model checker, and several different benchmarks available for research and experimentation. It is hoped that eventually, this work will lead to the adoption of formal methods by the games industry.

References

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

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

[3] K. M. Brooks. 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, 1996.

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

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

[6] A. Cimatti, E. Clarke, E. Giunchiglia, F. Giunchiglia, M. Pistore, M. Roveri, R. Sebastiani, and A. Tacchella. 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, July 2002. Springer-Verlag.

[7] M. B. Dwyer, G. S. Avrunin, and J. C. Corbett. Patterns in property specifications for finite-state verification. In *ICSE '99: Proceedings of the 21st international conference on Software engineering*, pages 411–420, Los Alamitos, CA, USA, 1999. IEEE Computer Society Press.

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

[9] J. Esparza. 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, 1998.

[10] R. Firth. Cloak of Darkness. `http://www.firthworks.com/roger/cloak/`, 1999.

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

[12] M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman & Co., New York, NY, USA, 1979.

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

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

[15] D. Kröning. Formal verification 251-0247-00. Course at ETH Zürich, 2005. `http://www.inf.ethz.ch/personal/daniekro/classes/251-0247-00/ws2005-2006/`.

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

[17] N. Montfort. *Twisty Little Passages*. The MIT Press, Dec. 2003.

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

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

[20] C. J. F. Pickett, C. Verbrugge, and F. Martineau. (P)NFG: A language and runtime system for structured computer narratives. In *Proceedings of the 1st Annual North American Game-On Conference (GameOn'NA 2005)*, pages 23–32. Eurosis, Aug. 2005.

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

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

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

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

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

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

[27] L. Vega, S. M. Grünvogel, and S. Natkin. 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, 2004.

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