PNFG: A FRAMEWORK FOR COMPUTER GAME NARRATIVE ANALYSIS

by

Félix Martineau

School of Computer Science
McGill University, Montreal

June 2006

A thesis submitted to McGill University
in partial fulfillment of the requirements of the degree of
Master of Science

Your file Votre référence
ISBN: 978-0-494-28511-4
Our file Notre référence
ISBN: 978-0-494-28511-4

Abstract

Narratives play a significant role in many computer games, and this is especially true in genres such as role-playing and adventure games. Even so, many games have narratives which possess a certain number of flaws that can deteriorate the playing experience. This less than satisfying gameplay experience can obviously affect the commercial success of a given game. Our research originates from the need to identify these narrative flaws. In response to this need, we present a framework for computer game narratives analysis. Our work focuses on Interactive Fiction games, which are textual, command-line and turn-based games. We first describe a high level computer narrative language, the *Programmable Narrative Flow Graph* (PNFG), that provides a high level, user-friendly interface to a low level formalism, the *Narrative Flow Graph* (NFG) [38]. The PNFG language is delivered with a set of enhancements and low level optimizations that reduce the size of the generated NFG output. As part of our work on the analysis of narrative structures, we developed a proof of concept heuristic solver that attempts to automatically find solutions to games from a lightweight high level representation. We also define narrative game metrics and present a metrics framework that simplifies the measurement and development of such metrics. These metrics contribute to broadening our general knowledge about game narratives.

Résumé

Les structures narratives jouent un rôle important dans les jeux pour ordinateurs et cette affirmation se confirme particulièrement dans des genres ludiques comme le jeu de rôle ou le jeu d'aventure. Malgré tout, plusieurs jeux possèdent des structures narratives comptent un certain nombre de problèmes qui peuvent contribuer à détériorer l'expérience de jeu. Cette même diminution peut aller jusqu'à affecter le succès commercial du jeu. Notre recherche émanne de ce besoin d'identifier ces fautes dans la structure narrative des jeux. En réponse cette même demande, nous présentons une structure applicative ayant pour but l'analyse des structures narratives. Notre travail s'effectue sur les jeux de fiction interactive, qui sont de nature textuelle et qui réagissent à des commandes entrées par le joueur. Tout d'abord, nous décrivons un langage de programmation haut niveau, le *Programmable Narrative Flow Graph* (PNFG) (Plan de Flux Narratif Programmable), qui offre une interface conviviale vers une base formelle, le *Narrative Flow Graph* (NFG) (Plan de Flux Narratif) [38]. Le langage PNFG est livré avec une série d'améliorations et d'optimisations qui réduisent la taille du NFG généré. Dans le cadre de nos recherches sur l'analyse des structures narratives, nous avons développé la preuve de concept d'un solutionneur heuristique dont le but est de trouver la solution du jeu et ce à partir d'une représentation haut niveau légère. Nous définissons également des mesures pour les structures narratives et présentons une structure applicative qui simplifie l'évaluation de ces mesures. Ces dernières contribuent à élargir nos connaissances générales au sujet des structures narratives.

Acknowledgments

I would like to take this opportunity to first thank my thesis supervisor Clark Verbrugge. My work in the field of games research originates from his vision, and I am very grateful to have had the opportunity to work on this very exciting research. He has been a great source of help and guidance every step of the way.

I also would like to thank Chris Pickett for his contribution to the PNFG framework. I am also very grateful for the work of Professors Joerg Kienzle, Hans Vangeluwe, and Bettina Kemme, as well as the members of the Games Research at McGill group for contributing to a very dynamic and enjoyable working environment, especially Alexandre Denault who helped me out for my first conference presentation. As well, I would like to extend my thanks to the McGill School of Computer Science, members of the faculty like Professor Martin Robillard, whose Software Evolution course led me to develop a critical eye when reading research papers. I also want to express my thanks to Grzegorz Prokopski of the Sable Research Lab for all the technical support he provided, and for taking the time to answer my numerous Linux questions.

Finally, I am very thankful to my parents and friends for their precious support. They have been an endless source of encouragement. Their indirect contribution to my work has been invaluable.

Contents

List of Figures

List of Tables

Chapter 1
Introduction and Contributions

Narratives play a significant role in many computer games, and this is especially true in genres such as role-playing and adventure games. Even so, many games have narratives which possess a certain number of flaws that can deteriorate the playing experience. Some of these problems are inconsequential, affecting only minor elements of game aesthetics, although even these interfere with a game player's sense of immersion. Other problems, however, can lead to narrative dead-ends where the player is completely stuck and is not able to finish the game at all. This leads to a less than satisfying gameplay experience, and obviously can affect the commercial success of a given game. Our research originates from the need to identify these narrative flaws.

In response to this need, we present a framework for computer game narratives analysis that includes a high level computer narrative language, a framework for measuring narrative metrics, and a heuristic solver that attempts to automatically find solutions to games from a high level representation. Our work focuses on Interactive Fiction games, which are textual, command-line and turn-based games. The player typically controls an avatar through a natural language interface, and playing usually consists of reading text on screen, entering a command, and then reading the text that appears as a result of the command being processed by the game. This sequence of events is repeated until the game ends in a win or a loss. This game genre is suitable for our research because it possesses the narrative properties we wish to analyze

while being relatively simple from a technical point of view. We describe this game genre in detail in Section 3.1.

We have previously explored the design for a high level language that allows for formal analysis of game narratives [29]. *Programmable Narrative Flow Graphs* (PNFGs) provide a high level, user-friendly interface to a low level formalism, the *Narrative Flow Graph* (NFG) [38]. This direct translation of PNFG programs to NFGs allows us to access a wide variety of research on understanding, optimizing and analyzing Petri Nets while maintaining our high level narrative programming environment.

We now extend and further develop the PNFG language. Our initial design included only quite basic language features; here we present the design in greater depth, and also extend the language with several useful syntactic forms. Although these extensions are largely "syntactic sugar" for patterns of lower-level operations they reduce programmer effort, and more importantly they allow for a reduction in the amount of redundancy in the low level NFG translation. We show how these more complex constructs are compiled to NFG structures, and also how this translation ensures a more efficient output structure.

Even with appropriate syntax, non-trivial narratives cannot be feasibly analyzed from a naive PNFG→NFG translation. Since our overall goal is to be able to analyze game narratives, we have also considered more general, low level optimizations that reduce the size of the NFG output, eliminating various kinds of redundancy. Reducing the size and complexity of the NFG is a crucial step in practical, formal analysis of non-trivial game narratives, and we discuss the necessity for optimizations and their relative impact. Since our design includes a practical implementation, we are also able to get real results on NFG output size reductions. These results show the significant effects of simple game optimizations, and also give guidance on the kinds and magnitudes of impacts due to specific game constructs, behaviours, and programming styles.

Our original work in [29] also featured a solver module, using the NuSMV formal model checking software [9]. This software uses a brute force approach on a Binary Decision Diagram (BDD) [8] boolean representation of a generated NFG to find the

minimal solution to the game narrative, among other properties. In a narrative game, a solution is a sequence of commands entered by the player that lead to winning the game. Because of this brute force nature, the size of the state-space of the narratives we wish to solve is always a key issue. Even with many optimizations, considering the full state space for larger narratives is not realistic, and we investigated the potential of developing a heuristic solver that uses high-level information found in the PNFG intermediate representation. This heuristic solver looks at the conditions that need to be satisfied when winning the narrative in order to build a solution. Such high-level strategies can contribute to make possible the analysis of very complex narratives. We have developed a proof of concept version of the heuristic solver and we will discuss the possibilities that arise from our results using this initial version in Chapter 5.

We also sought to use high-level information to measure and evaluate game properties. We will present a set of narrative game metrics, and discuss the results from measuring them on some example narratives whose complexity range from very simple to fairly complex. Some of theses examples are even based on commercial works of interactive fiction. We devoted significant efforts on creating an intuitive representation that made use of the structure of the PNFG file itself, and we will give a detailed description of the metrics framework we are using to evaluate the different properties.

1.1 Contributions

Specific contributions of this work include:

- We give a detailed overview of the basic PNFG language, design and compilation.
- We present language extensions to the PNFG language, including specific NFG compilation strategies. These constructs reduce both redundancy in the initial PNFG code and in the underlying NFG.
- We define and provide experimental data on the effect of a variety of low-level

NFG optimizations. These optimizations have varied effects, but can overall greatly reduce the NFG size.

- We present a proof of concept narrative game solver that uses chains of conditions in the PNFG source file to heuristically build a solution, and discuss the potential of this method.
- We provide a metrics framework that works with the PNFG intermediate representation to allow the evaluation of metrics in narrative games.
- Through the use of our metrics framework, we can determine whether a game can be won or not. This can be viewed as the first step towards detecting flaws in narrative games.

1.2 Roadmap

In the next Chapter, we discuss related work on narrative analysis and representation. In Chapter 3 we give an overview on the structure of PNFG programs by presenting core functionalities of the language. We then explain how the different constructs and syntactic components of PNFG are translated into an NFG representation. Chapter 4 describes our different NFG optimizations, and Chapter 5 deals with the heuristic solver and how it differs from our previous approach at finding a valid solution automatically. We will refer to this as solving narrative games. The metrics framework and the metrics themselves are presented in Chapter 6. Chapter 7 describes each example narrative we have used and gives results for our optimizations, solutions found by the heuristic solver, and metrics results. We conclude in Chapter 8 and discuss future work in the area of computer narrative analysis.

Chapter 2
Related Work

Interactive Fiction (IF) is one of the first and also one of the oldest computer genres, largely because it has a very limited technical overhead. IF can be defined as being "A computer program that generates textual narrative in response to user input, generally in the form of simple natural-language commands" [1]. The term was first used in a 1987 BYTE magazine article as a label for story centered computer games [35]. Among the most famous titles, we find examples such as Adventure (also known as ADVENT or Colossal Cave) [11], Zork [20], and The Hitchhiker's Guide to the Galaxy [2]. IF's popularity reached its peak in the mid eighties, and was all but dead by 1990. Today, a very active if small IF online community exists and is organized around different USENET newsgroups such as `rec.games.int-fiction` (or r.g.i-f), [37] and `rec.arts.int-fiction` (or r.a.i-f) [16], the first focusing mainly on playing games, while the latter deals with the creation of new IF works. Detailed examples of IF games can be found in Section 7.1.

In recent years, attempts have been made at analyzing IF from a theoretical approach, arguing that it is a legitimate literary art form [23]. When we look at what has been written on IF from this point of view, the term Interactive Fiction in itself is problematic and has faced a very tough opposition from literary theorists. It has been criticized as "...facing enormous problems" and that "Interactive fiction is [...] in reality largely the rhetoric for a Utopia" [17]. Since the term has been accepted as representing the game genre we are exploring, we will continue to use it, but keep in

mind the term can be controversial at times.

Traditional approaches at narrative analysis have included the decomposition of stories from films, such as Christopher Volger's Hero's Journey [39], and have been proposed as narrative patterns for computer game design [32]. Within the realm of computer games, two different types of narrative can be observed: embedded, and emergent narrative [19], where the former is "pre-generated narrative content that exists prior to a player's interaction with the game", and the latter refers to the narrative that "...arises from the set of rules governing interaction with the game system" [33].

It has been said that computer games fall within a continuous space that goes from ludological, the extreme being computer chess, to narratology, with DVD movies at the other end of the spectrum [21]. As we move towards the narratology extrema, the notion of game logic becomes a bigger concern and can also lead to critical failures within the game. For example, a player needs a key which she cannot obtain in order for the game to progress. This problem of unwinnability has led to the definition of *p-pointlessness* where a small value of p ensures a quick termination of an unwinnable game [38]. This is one example of a property we would want to derive from our computer narrative analysis.

Different solutions have been discussed to reduce problems in narrative games, an example being a plot diagramming module for the popular IF authoring tool TADS [4]. This particular module uses Directed Acyclic Graphs (DAG) to represent the narrative, but it has been shown that DAGs are not suitable for the representation of computer game narratives [38]. Other game construction kits such ScriptEase [22] contribute to reducing the number of logical errors in games by providing a simpler approach to game authoring. Other approaches reduce the programming effort to a minimum, and aim to allow the user to express games from a limited genre such as board games with the *Metagame* [28] game generator, to offering a broader scope with the *Extensible Graphical Game Generator* [27].

Interactive Fiction toolkits are very popular among the IF community, and the most popular authoring kits are probably Inform [25] and TADS [13, 31]. Inform is based on the tool used by the company Infocom, who published some of the most

successful works of IF during the eighties. Inform 7, the latest version of the toolkit, features an advanced IDE that allows users to create IF by writing natural English language sentences [26], an approach that radically differs form the traditional computer programming used in previous versions of Inform and other toolkits. TADS features an very detailed object-oriented language with features such as dynamic objects, structured exceptions, and automatic garbage collection, among others. The different authoring tools allow programmers to create complex storylines, although none of these systems directly address the issue of game analysis.

The representation of a game narrative as a Petri Net is at the core of our work, and builds on our previous, somewhat naive approach to Petri Net generation. Petri Nets for complex systems can be quite large in practice, and different solutions have been proposed to reduce the state space of Petri Nets. Work has been done, for instance, to develop reductions that are compatible with bisimulation principles [34]; examples include the fusion of equivalent *places* and the replacement of some *places* by others. Abstract interpretation has also been considered as a means to derive non-structural invariants of a given Net [10]. Similar Petri Net reductions have also been used in Artificial Intelligence to represent a team plan and its projections on individual agents, by using techniques such as fusion of consecutive activities, fusion of parallel activities, and fusion of choice between activities [7]. Using structural reductions like projection and redundancy removal, it is possible to reduce the size of probabilistic timed Petri Nets [18]. Reusing existing Petri Net reductions and applying them to a domain-specific representation can be rewarding, as demonstrated in [12], where reductions suggested by Berthelot [6] are applied to Task-Interaction Graph-based Petri Nets.

The narrative programming language we present here as part of our framework directly extends our initial definition of *Programmable Narrative Flow Graphs* (PNFGs), a high level language that is easily mapped to a Petri Net model [29]. Other work on game narrative structure has led to the definition of a *Classical Game Structure* [30] where the player starts with a few choices that lead to more, and where these choices "gradually narrow back down again to a few, or single action the player must accomplish". This concept is defined as a "convexity", and a series of such convexities

can be seen as representing the different parts of a game.

In his survey of interactive fiction, Smith modelled what he refers to as the "traditional interactive fiction". He separated it in sections where "the interactor may operate with some freedom. But to get to the next section he must bow to the prescriptions of the story and thus temporarily abandon his freedom in order to progress" [35]. While other such models of "traditional" game narratives exists, we do not know of any model that is backed by any data.

Chapter 3
PNFG Language Design

Flaws in game narratives can be easily identified during the gameplay, but we currently lack the tools to analyze these narratives in order to detect these flaws and have good narrative properties. From that problem arises the need for a high level programming language that is built on formal principles. Such a language will provide the strong base that is needed to formally analyze these narratives.

In this chapter, we first give a description of an interactive fiction narrative and the representation we use in order to do the analysis (Section 3.1). In Section 3.2, we define the basic PNFG language constructs, and Section 3.3 details the PNFG game control flow. Section 3.4 describes the different PNFG statements that can be used to represent a game narrative. Finally, in Section 3.5, we discuss the syntactic sugar components we have added to the PNFG language in order to extend its expressive power and reduce code duplication.

3.1 Narrative Representation as a PNFG

Interactive Fiction games (IF) are textual, command-line and turn-based games typically composed of an avatar moving through a fairly minimal virtual environment consisting of *rooms* or locations, and including some number of *objects*. The avatar is usually controlled by the player through a natural language interface, incorporating simple commands to take, drop, and use objects in different manners. Game progress

and conflict is represented by different puzzles or obstacles that must be overcomed by suitably arranging or employing game objects. The game can be won by solving all or most of the problems, or lost by incorrectly solving one or more puzzles.

IF is a very interesting game genre for our research because it allows us to focus on the narrative qualities of a game, while limiting the technical complications that invariably come with other genres, such as 3D graphics (or even 2D graphics), sound, networking, and other non-narrative aspects of more contemporary game designs. In our case we further exclude the natural language interface, as another aspect that is tangential to the main narrative structure of the game.

To represent IF games, we use the *Narrative Flow Graph* (NFG) [38], a special class of 1-Safe Petri Nets that provides a simple syntax and operational semantics for describing narratives. A Petri Net is composed of nodes (*places*) and *transitions,* and directed edges, where edges run between places and transitions. When all the incoming nodes connected to a particular transition t have a *token*, t can *fire,* removing tokens from the input nodes and inserting tokens into the output nodes of t. This state transition easily represents the typical IF behaviour of triggering an event during game play based on existing game state, which explains why NFGs are based on Petri Nets. 1-Safe Petri Nets have the added property that each place can contain only one token. With NFGs, we also note the presence of Context Edges that behave like a bidirectional connection to a transition in a Petri Net. These edges are used to represent that a certain property in the narrative will remain true even after an action has been executed. For example, the player will still have a key in his inventory even after she has used it to unlock a door. The NFG definition also explicitly defines specific starting and ending nodes. The starting node a is unique, while there are two ending nodes: the losing node l and winning node w. Having these starting and ending nodes allows for paths to be defined in the structure. The starting node a represents axiomatic precedence, all initial conditions are connected to a The NFG definition explicitly states that the nodes l and w must no be simultaneously-reachable. This last specification is crucial when representing narrative games, because we cannot win and lose a game at the same time. A formal definition and further details on NFGs can be found in [38]. Figure 3.1 shows a basic NFG representation for the trivial

narrative used as a motivating example in [21].

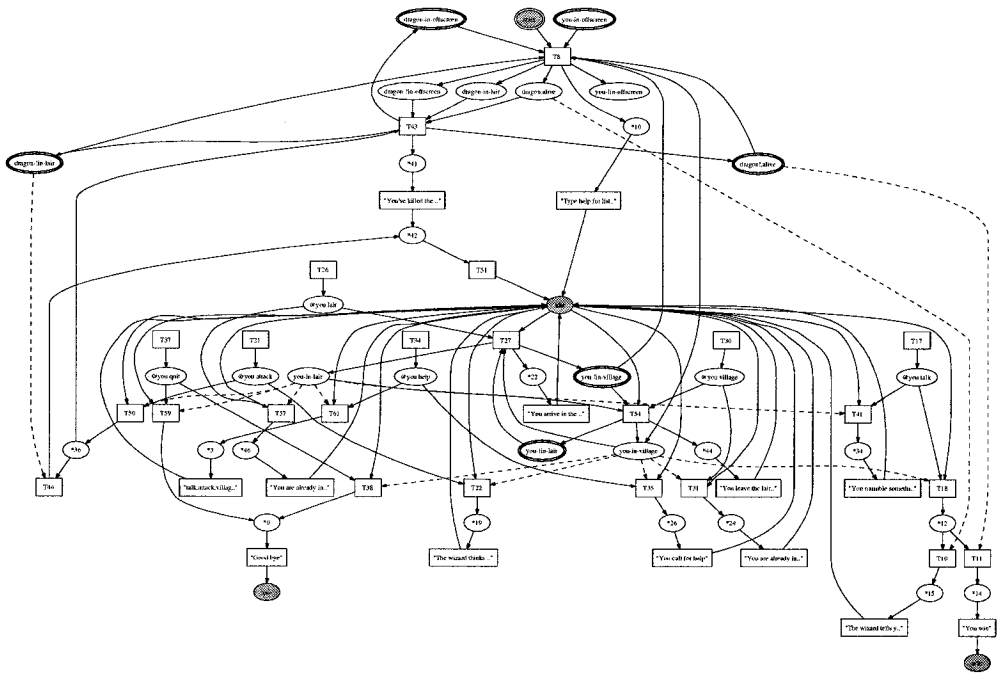


Figure 3.1: A NFG for the trivial narrative *The Wizard* [21].

3.2 PNFG Data and Declarations

Representing a narrative directly in NFG form can be very tedious; as can be seen from Figure 3.1 the size and the complexity of the graph can make the task overwhelming, even for a relatively small narrative. As a more practical means of developing narratives, the *Programmable NFG* (PNFG) language, first introduced in [29], is a high-level representation of a game narrative that is translated to a corresponding NFG. This allows for a much more intuitive representation of the narrative, while also offering a translation mechanism that is structured and efficient.

The design of the PNFG language has been based on the more popular IF toolkits, albeit stripped down to only essential features. Through *object*, *room*, and *action*

declarations, the user can express the basic game narrative structure. Using additional constructs such as *states*, *counters*, and *timers*, it becomes possible to represent complex IF games. For each of these language components, we must always have a way to translate them to a valid NFG representation. In following sections, we will describe this translation, along with the PNFG syntax and the general structure of a PNFG program.

3.2.1 Objects & Rooms

Objects and rooms are the two most basic components of narrative games; as described above, in IF games the player usually moves from one room to another and is required to interact with different objects in order to eventually win the game. In the PNFG language, objects can be declared quite simply, as shown in Figure 3.2. This particular statement will be translated into a unique game object called "`dagger`." In the PNFG language all object declarations must be performed statically; this does not allow for infinite or arbitrary numbers of objects, but is nevertheless appropriate and adequate for most IF games, and has so far not proven to be an obstacle to complex game development.

```
object dagger { }
```

Figure 3.2: *A simple object declaration.*

Rooms in the PNFG language have an almost identical declaration syntax to objects. The major difference between objects and rooms is simply that rooms can function as containers, and can hold objects, even other rooms. The player herself is in fact typically described using a room declaration in order to allow her to have an inventory. Containment is presumed to form a tree structure in the PNFG language, with every object and room having exactly one parent (container) room at any one time. To guarantee this property holds at all points in the game, the PNFG language pre-defines a reserved `offscreen` room where all objects and rooms initially reside, and to where they can be moved when they are no longer part of active game play. The `offscreen` room is unique in that it cannot be moved or contained itself.

The mapping of object containment from the PNFG space to the NFG is achieved through the creation of two unique NFG nodes for each game object or room in each possible containing room. For an object A and a room B, a node representing "A in room B" and a node representing "A not in room B" will be generated. These nodes function with complete complementarity, and the NFG constructed will guarantee that if the node "A in room B" is active (contains a token), the node representing "A in room R" is inactive (does not contain a token) for all other possible rooms R. This means that the narrative begins with the nodes "x in `offscreen`" active for all objects and rooms x, except of course `offscreen` itself.

3.2.2 Sets

Many operations in an IF game will be identical for some number of different objects or rooms. *Drop* and *take* actions, extremely common actions in IF games, for instance tend to be quite similar or identical for a large subset of game objects. To lessen the amount of coding redundancy subsets of objects (and/or rooms) can be declared in a PNFG program, and elements of the set referred to by abstract set variables. Use of the set variable is then internally expanded according to the semantics associated with the context of the use of the set variable: such variables can be *bound* or *unbound.* In an unbound context set variables are merely macros for replicating a program command over some number of distinct objects or rooms; when bound by an enclosing statement and scope, however, a set variable refers to a particular element of the set used as part of its declaration. We will further discuss the use of set variables, bound and unbound in Section ??.

An example of both bound and unbound set variable declarations is shown in Figure 3.3. Notice that set definitions can refer to other sets as well as individual objects and rooms, and can also be constructed through subtraction as well as addition of elements or other sets. To avoid declaration-ordering constraints forward references, and recursive definitions are permitted, although infinite and contradictory set constructions are of course not allowed. Actual set contents are computed at compile time using a (least) fixed point algorithm.

```
carryable = { dagger, banana }
uncarryable = { widget, kleinbottle }
stuff = { everything, -you }
everything = { carryable, uncarryable, you }
...
stuff $mystuff;
carryable $c;
```

Figure 3.3: *Set declarations and set variables.*

3.2.3 States

Having both rooms and objects allows a game programmer to express some very simple narratives, and is in fact sufficient to achieve our desired level of expressiveness. The PNFG language, however, also offers *State* declarations to be associated with rooms and objects as an alternative way of representing current and changeable properties of the game. States are binary, and can be set to *true* (+) or *false* (-), as we will discuss in Section 3.4. Figure 3.4 shows a room declaration with two states being declared, `trapOpen` and `lit`.

```
room bedroom {
   state {trapOpened,lit}
}
```

Figure 3.4: *A room with 2 declared binary states.*

Similar to containment, in the translated NFG output each individual object or room state declaration will be represented by two nodes, one for each binary value. For example, the states declared in Figure 3.4 would be represented by four nodes, `-bedroom.trapOpened`, `+bedroom.trapOpened`, `-bedroom.lit`, and `+bedroom.-lit`. Again, since both state values are mutually exclusive the two nodes forming the pair representing a particular object or room state cannot both be active at the same time. All states begin with the false node active.

A few special states are defined to indicate the player winning or losing the game. A reserved object name game is used for this, and includes `win` and `lose` states. When

win or lose is set to true (+), it means that the game has been won or lost, respectively, and game play is automatically terminated.

3.2.4 Counters and Timers

Many IF games require *counting:* some typically small and finite number of steps or events must occur in order to trigger a subsequent event. Using states it is quite possible to build finite counters by composing a series of states for each possible counting value; e.g., `x.value0`, `x.value1`, `x.value2`, `x.value3` for a counter with range $0 \ldots 3$, with the game programmer ensuring that at most one of these positive states is true at any one time.

Counters automate and abstract this process, and allow the programmer to declare variables which can be set, incremented, or decremented by a constant value within a given range. This eliminates potential programmer error in use of counters, and also allows for easier optimization of the ensuing NFG code generation. Figure 3.5 shows an example of a counter declaration for a counter `you.lives` that can assume a value in the range $0 \ldots 3$.

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

Figure 3.5: *A counter definition for the inclusive range 0..3.*

Counters are trivially represented in an output NFG by generating an equivalent set of states, initializing the state representing the minimal value to true and all others to false. Operations on counters are then required to ensure the corresponding set of states continues to guarantee that exactly one of the states is true. This unary representation strategy is not necessarily optimal, and we intend to explore binary representation as an optimization in later work.

With counters the programmer can specify exactly how the value will be increment or decremented, according to the desired behaviour. The PNFG language also supports *timers*, which are in fact special counters that are automatically incremented

after each action the user executes. How this is achieved will become more obvious in the following section. Timers, however, act only as further syntactic sugar on counters in order to avoid some code duplication.

3.3 PNFG Execution

Interactive Fiction games, or turn based adventures, are typically made up of three different phases: the prologue where the game is initialized, the cycle of waiting for the user commands and processing them, and finally, an epilogue phase [24]. The PNFG compiler generates a similar structure, and the general control flow of a PNFG game is shown schematically in Figure 3.6.

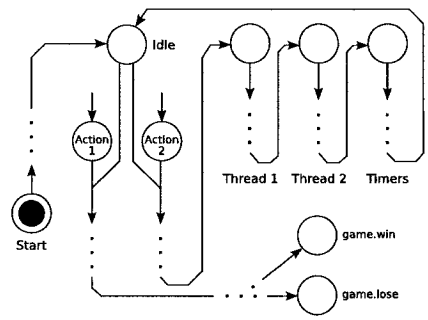


Figure 3.6: *The general NFG structure for a PNFG program.* The entry points for the main phases of execution are prologue, user commands, user threads, timers, and epilogue. *Taken from* [29]

At the beginning of the narrative the *start* node is active and triggers the prologue or game initialization. This first stage terminates at the *idle* node, where control flow waits for user input. When a user command is received it is processed, moving control into the appropriate *action* or set of execution statements. After an action has been completed the game may terminate in a win or loss, or pass control to a set of user-defined and internal *threads*. Threads are not concurrent in our language, and are in fact used primarily to append fixed execution behaviours to the end of each user-initiated action. A primary internal thread is responsible for incrementing timer

values as discussed in Section 3.2.4. Finally, control returns to the idle node, where it waits for the next user input command.

User commands or actions are composed of PNFG statements, and have a similar appearance to standard procedural languages such as C or Java. Figure 3.7 shows a code snippet for the action triggered by the user command "kill npc" in one of our example games, *The Return to Zork - Chapter 2*, where the player can kill a non player character (NPC). Executing the action results in each statement being processed according to the execution semantics we will define below. In the case of this example action we first define a set of items at line 02. Then we check whether or not the player has the `knife` in her inventory. If she does, it means she can actually kill the npc; in the actual game the "Guardian" appears and strips the player of all her inventory items as punishment, and further sets some player states indicating that the player has performed this violent act. Later actions in the game branch on these states, resulting in a permanent (and undesirable) impact on the player.

```
01  (you,kill,npc) {
02     stuff = { knife, rock, vine, ...}
03     if(you contains knife) {
04        "You kill the npc";
05        " . . . The Guardian appears . . . ";
06        "The Guardian :  I must relieve you of your belongings";
07        for(stuff $s) {
08           if(you contains $s){
09              move $s from you to offscreen;
10           }
12        }
13        -?you.friendly;
14        +you.killer;
15  } } } }
```

Figure 3.7: *A sequence of PNFG statements corresponding to a "kill npc" command.* Statements are referred to by number in the text.

3.3.1 Basic control flow

Each statement in a PNFG program is expected to be executed in the order specified. Since Petri Nets transitions do not enforce this sequentiality by definition, a general structuring principle is used to enforce correct control flow through the use of *context* nodes. The context nodes form a set of nodes that are guaranteed to have exactly one node active. Transitions generated for individual statements rely on an input context being active to allow the statement transition(s) to fire, and must guarantee the activation of a single output context to feed to the subsequent statement.

The `start` node is in fact a context node (the only initially-active context node), and allows control to flow through the game prologue to `idle`, also a context node. The `idle` node then passes control to the first statement of an executed action and is expected to receive control back from the last statement in each action, or the last statement in the last thread if any threads (or timers) are defined.

3.4 Basic PNFG Statements

The example of Figure 3.7 illustrates most of the core operations available in the PNFG language. This section describes each of the basic operations, as well as how each of these actions is translated into some number of transitions in the underlying output NFG.

3.4.1 Output Statements

One of the most important components of Interactive Fiction is the actual output produced when the player enters a command. With output statements, we allow the narrative programmer to print messages, and thus communicate with the game player. They are created by declaring a string constant, as shown in statements 04, 05, and 06. The strings are sent "as-is" to the game console, although there is some rudimentary syntax to allow output to mention objects referred to indirectly by set variables; Figure 3.8 shows a generalized example. The corresponding NFG pattern is a simple transition expressing the output string, as shown in Figure 3.9.

```
stuff $s;
...
"You are carrying too much.";
"Drop the ${s}.";
```

Figure 3.8: *Syntax for an Output statement.*

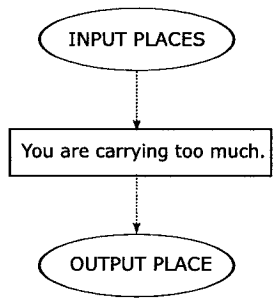


Figure 3.9: *NFG structure for the first output statement in Figure 3.8*

3.4.2 Set Statements

Several approaches are available to the programmer for changing the value of a particular state inside the game. As shown in line 14 a simple set of the `you.killer` state is expressed as `+you.killer`. This basic statement is considered a *blind* operation, in the sense that if the state is already true prior to execution of the statement the NFG output transition will be unable to fire and the execution will stall, unable to activate the appropriate output context, as shown in part *a* of Figure 3.10. Using the blind set statement is perfectly valid when the value of the state is certain, but in the case when it is not surely false on input a *safe* set operation is also available. An example of a safe set is shown at line 13, with a schematic NFG generation as shown in part *b* of Figure 3.10; here two transitions are generated, one for the case of the incoming state being false, and one that acts as an identity in the case that the incoming state is already true. Since states will be either true or false this guarantees exactly one transition will fire, and the output context will become appropriately activated. Finally, a *toggle* operation can be used to flip the state value, whatever its incoming

status. The NFG translation for all three forms of the set statement are displayed in Figure 3.10.

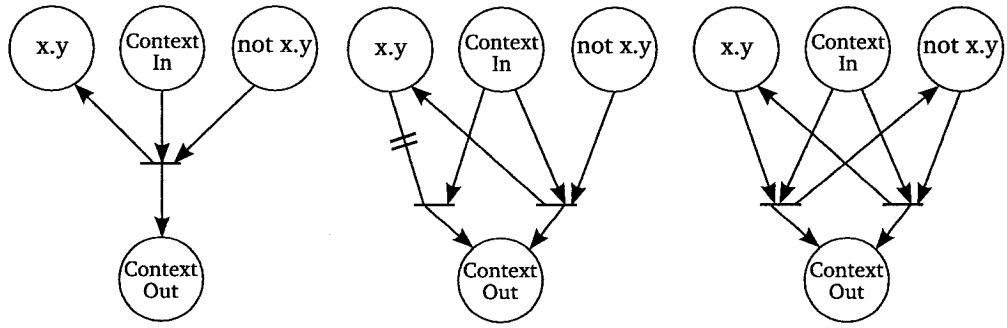


Figure 3.10: *NFG structure for the 3 main variations of the set statement.* Similar operations are defined for the symmetric unset operations, `-x.y` and `-?x,y`. Lines with two short lines intersecting represent context edges (*Section 3.1*)

3.4.3 Move Statements

Movement of objects is accomplished in a similar fashion to state manipulation; statement 09 provides an example of a move statement, and Figure 3.11 shows the generated NFG. In general moving x from y to z involves deactivating the nodes corresponding to "y `contains` x" and "z `does not contains` x" and activating the nodes representing "y `does not contains` x" and "z `contains` x." For move statements safe versions are not provided, primarily to help ensure efficient code generation. Safe versions of state changes are relatively efficient, requiring only two transitions to implement effectively. For move statements, however, all potential locations of an object x would have to be accommodated, and this could result in a much larger NFG output. Similar safe effects can be achieved through the use of an enclosing `if`-statement, as we describe below.

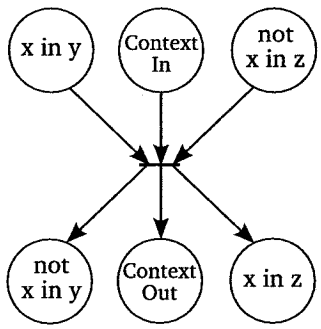


Figure 3.11: *NFG structure for statement, "*`move x from y to z`*".*

3.4.4 Counter operations

As discussed in Section 3.2.4 counters are represented in unary, in a manner quite similar to basic object/room state variables. Modifying a counter is thus a simple manner of adjusting the appropriate subset of unary value states. Figure 3.12 shows an example of code generation for a counter increment following the semantics of the well-known "++" C/Java operator; a transition is generated to move each unary value to the next higher value, predicated on the unary value having a true state. Exactly one of these transitions will actually be executed at runtime.

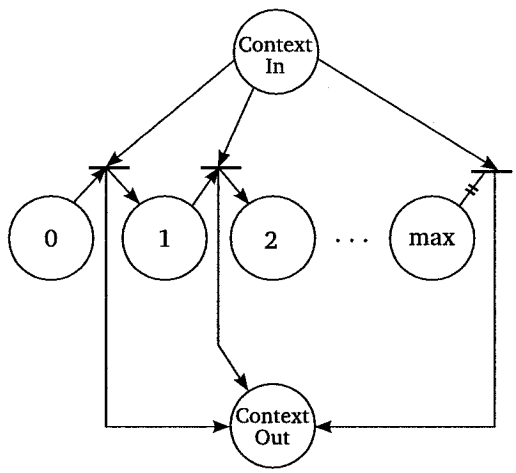


Figure 3.12: *Counters.* NFG structure for a counter increment.

Code generation for increment or decrement of a counter by an arbitrary constant value follows the same pattern, and increment or decrement by a variable value (another counter) would be easy to add. The latter operations are not currently defined in the PNFG language, primarily to avoid the temptation by programmers to use non-trivial counter ranges and operations, given the potentially large code generation that can result in our simplistic, unary compilation strategy.

A final concern in code generation for counters is how to handle overflow and underflow. Various approaches are possible, including error-generation or implicit application of modulus; in our case we have elected to make overflow and underflow operations identity functions.

3.4.5 If Statements

Branching is an essential feature of any significant programming language. In the PNFG language the narrative programmer can test for properties such as containment, state values, and counter/timer values. The statement at line 03, for instance, checks whether or not the knife is contained in the player's inventory, and if so will execute lines 04–14. The NFG representation of a simple `if`-statement is shown in Figure 3.13. Conditional statements in general introduce distinct control flows for the two branches, with only one of the corresponding contexts active after the test. A final output context is then generated for the merge of the two branches.

The use of set variables in conditionals adds an extra complexity. The intention of a statement "`if (x contains $y) {...}`" is that the body of the if-statement would be executed if x contains any of the objects represented by the set variable `$y`. Moreover, within the if-statement body the variable `$y` would then be "bound" to a particular set member, and could be referenced and used as a normal object/room reference. The NFG representation for this kind of set variable binding through if-statements can be seen in Figure 3.14. Of course, this kind of compilation schema leads to redundant structures inside the generated NFG; under certain conditions this redundancy can be optimized away, as we will discuss in Chapter 4.

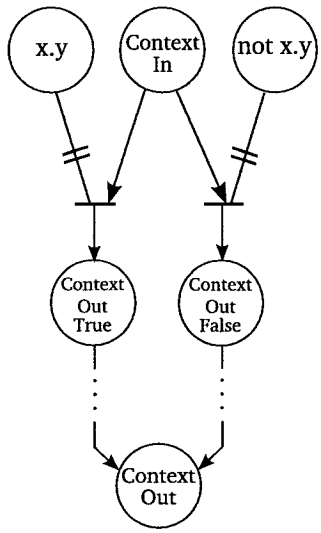


Figure 3.13: *NFG structure for a statement,* "`if (x.y) {...} else {...}`". Negative state tests ("`x!.y`") and positive/negative containment tests are structurally identical.

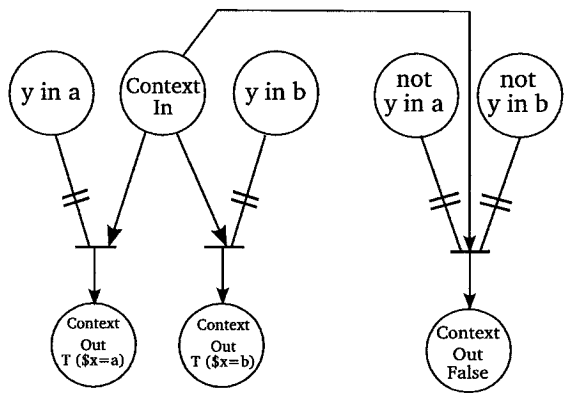


Figure 3.14: *Using variables.* NFG structure for a statement, "`if ($x contains y) ...`" where "`$x`" is an element of the set "`{a,b}`". Branch bodies and the following merge are not shown.

3.4.6 Actions

The statements described above can be executed as part of three different constructs of the PNFG language, namely the initialization of the game, actions that are triggered by the user, and *threads* that are automatically executed after each action. We will now describe how actions can be constructed.

In most IF systems the user enters commands using a natural language interface, and discovering the appropriate language for an action can be a central, if often vexing part of the assumed game play. The PNFG language does not model this interface, and instead user commands are represented and derived from a simplified, canonical language. Actions in the PNFG language are defined by either a (subject,verb) or (subject,verb,object) declarations. When user input is received and matches an action declaration its underlying PNFG statements are executed. In its current version, the PNFG language assumes the subject of the action is always the player, represented by the room "`you`". Thus the user input "kill npc" triggers the action `(you,kill,npc)`. The ability to define and use other subjects is intended to support concurrent game play, and is part of our future work.

Basic action declarations such as in Figure 3.7 can be executed at any time, and are considered to have a global scope. A nice syntactic feature of PNFG allows for actions to be "scoped" to individual rooms by nesting their declaration within the room declaration. The action is then only available to the game player when she is located in that particular room. This feature turns out to be particularly useful when it comes to encoding the "map," or room connectivity inside the game, as shown in Figure 3.15.

Encoding room-specific actions is straightforward; an action with subject s defined in room r is semantically identical to embedding the action within a statement "`if (`r `contains` s`) {...}`".

```
room lighthousefront {
   (you,look) {
      "You are standing in front of the";
      "lighthouse.  From here you can travel";
      "in the four cardinal directions.";
   }
   (you,go,north) {
      "You walk up to the mountain pass.";
      move you from lighthousefront to
         mountainpass;
   }
   (you,go,east) {
      "You step behind the lighthouse.";
      move you from lighthousefront to
         lighthouseback;
   }
   ...
}
```

Figure 3.15: *Room-specific actions.* These actions shadow global actions with the same user command specification, while the subject (you) is in the declared room (*taken from* [29]).

3.5 Syntactic Sugar Components

The PNFG language, with the constructs presented thus far, allows for the expression of complex narratives, but certain parts of these narratives can be very tedious to write and usually involve code duplication. In this section, we present additional "syntactic sugar" components whose goals are to improve the usability of the high level language and to limit the amount of code duplication that needs to be done. Our contribution includes the proposal of these components and their implementation in the language. Note that we will continue to refer to statements from Figure 3.7.

3.5.1 Variables & Sets

Most of the basic PNFG statements can also accept set variables as object/room specifiers instead of specific objects. This contributes to reducing code redundancy inside the PNFG source file. In their simplest, unbound form the use of set variables causes the corresponding statement to be replicated, one copy for each possible instantiation of the set variable, all sequentially linked in an arbitrary order. In the case of bound set variables, and as discussed in reference to the if-statement (and for-statement below), a set variable will represent a single, specific object or room, and compilation is identical to the case where the set variable is suitably substituted by the object/room name.

3.5.2 For & Forall Statements

Statement 07 shows an alternative method for replicating sections of code over multiple objects, the *for*-statement. The for-statement in the PNFG language will execute the body statements for each member of the set it receives in its declaration, binding a corresponding set variable for use in the body of the for-statement. The for statement in Figure 3.16a can be seen as a sequence of body executions, each with the set variable substituted by a different set element. The PNFG language also has a `forall` (Figure 3.16b) statement that allows for parallel execution via concurrent activation of transitions. This latter variation is provided primarily as part of future

work on concurrency in IF games.

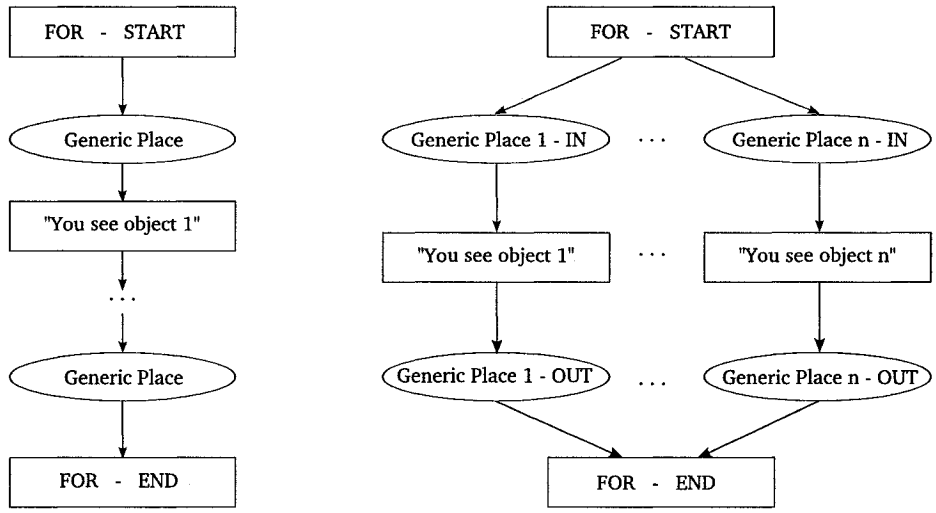


Figure 3.16: a) *Using for* execution of each case is sequential. b) *Using for-all* execution of each case is parallel.

3.5.3 Enter & Exit.

When a player moves from one room to another, a good game design strategy is to have an output statement that informs the player that she is now in the other room, and/or has left the previous room—this acts as confirmation or essential feedback for the activity. This can of course be hand-coded at every player (you) movement statement; enter and exit blocks, however, simplify the effort by allowing us to define statements that are automatically executed when the player enter and exits a room. This not only reduces the code duplication, it also leads to code that is much easier to understand for the programmer.

Enter/exit blocks have a relatively simple syntax, consisting of just a keyword and compound statement declaration, within the declaration scope of the relevant room, as shown in Figure 3.17. Semantically, an exit block is executed just prior to

the actual move, while the enter block is executed immediately after the movement is performed.

```
room oldMillBack {
   enter {
      "You arrive in the backyard of the mill";
   }
   ...
   exit {
      "You go back inside the mill.";
   }
}
```

Figure 3.17: *Enter and Exit blocks.* The statements in the enter block get executed when the player enters the room, while those in the exit block get executed when the player leaves the room.

3.5.4 Threads

Threads can be used to define sequences of PNFG statements that will be executed after a player action has been fully executed. In the absence of threads these blocks of statements would have to be copied after each and every action, which would introduce severe code redundancy. Threads are usually executed unconditionally after each action (and order of thread execution is undefined); *conditional* threads, however, are also available and execute only if a specific boolean condition is satisfied. Conditional threads effectively behave like "triggered" events, a common behaviour found in narrative games. Examples of unconditional and conditional threads are shown in Figures 3.18 and 3.19 respectively. Note that the behaviour of a conditional thread can also be easily mimicked by an unconditional thread by moving the condition inside the thread body.

```
thread {
   you.moves++;
   if (you.moves==55) {
      "You have no more time.";
      +game.lose;
   }
}
```

Figure 3.18: *Threads.* After each move made by the player a counter is incremented; if the limit of moves has been reached the game ends with a loss.

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

Figure 3.19: *A conditional thread declaration.* This thread only executes when the state `bomb.active` is true (*taken from* [29]).

3.5.5 Timers

In the previous section we addressed the issue of compiling counters into a corresponding narrative flow graph. It is also convenient to support the concept of self-incrementing counters, or timers. Many IF games contain sections where the player has a limited number of moves to complete a certain task. Reproducing this behaviour with counters alone meant the programmer had to insert a counter increment statement after each action or to have a thread for incrementing counters, as shown in Figure 3.18. Timers obviate that manual specification, but are merely counters incremented automatically by an internal, system-defined thread. An example of a timer declaration is shown in Figure 3.20. Timers begin at the minimum declared value, increment by 1 each turn after all user threads have executed, and as with counter overflow timers that reach the maximum value remain at that value from that turn onward.

```
timer {
   moves 0 60
}
```

Figure 3.20: *Declaration of timers.* This block declares a counter for the number of moves made, beginning at 0 and reaching a maximum of 60.

3.5.6 Functions

Having functions in our high-level language allows the programmer to isolate certain narrative behaviours. Functions are very useful when the programmer wishes to have actions that are available in more than one room but not all rooms, or just to reuse a particular behaviour. We have defined a set of replacement rules for each type of PNFG statement that allows one to pass parameters to defined functions, and customize their behaviour. Figure 3.21 illustrates the definition of a function that uses one parameter. At compile-time, each call to the function `takePicture` will be replaced by its statements, and *photo* at line 03 and 05 will be replaced by the parameter used in the function call.

```
01 function takePicture(photo) {
02   if(you contains camera){
03      if(you contains photoalbum && photoalbum !contains photo){
04         "You took a picture";
05         move photo from offscreen to photoalbum;
06 } } }
```

Figure 3.21: *Function declaration.* This function can then be called in any action, and the actual function call will be replaced by the function body.

3.5.7 Default Actions

Actions defined within rooms are not typically intended to be available when the player is in other rooms. A player attempting to execute an action specific to room r in a different room s will thus find the action is silently ignored. This is fine and correct from a compilation perspective, but is clearly not particularly user-friendly—at the very least there should be feedback to the player indicating that that the command entered is (currently) invalid.

This can be easily accomplished by generating appropriate negative feedback actions in all rooms other than the ones in which a given room-specific action is defined. This is highly-repetitive, however, and so the PNFG also provides *default actions* to automate the process of emitting negative feedback. Default actions may be thought of as global actions, filling in the gaps introduced when there is no room-specific definition for an action in room s for a room-specific action defined in room r. The basic default action is to emit a simple response "What?" to an invalid user command. Syntax to allow further variation on the default actions would be straightforward to include, and is part of our future work.

3.6 Summary

The goal of the PNFG language as a whole is to be able to analyze properties of game narratives, and to make the creation of complex narratives a much easier task than writing an NFG from scratch. The basic framework presented so far allows a

programmer to represent complex narratives which can then be compiled down to a Narrative Flow Graph. Many linguistic components are provided to reduce the amount of redundant code inside the PNFG source file, and improved the overall usability of the PNFG language. Although these are effective and useful, unfortunately they are insufficient to generate highly minimal NFGs for the purpose of verification. In the next section we explore a variety of low-level NFG optimizations designed to help further reduce the compiled output size and improve verification possibilities.

Chapter 4
PNFG Optimizations

The PNFG allows for a straightforward expression of game narratives, but the generated NFGs tend be very large, to the extent where the NFG generation process for our largest example narrative would not terminate in a reasonable amount of time. Another side effect of producing large structures is that it becomes increasingly difficult to analyze them. In this chapter, we present the different optimizations we have applied to the PNFG translation process in order to reduce the size of the generated NFG/Petri Net. By reducing the size of the output we can analyze properties faster, while also increasing the limit on the size of narratives we can analyze using the NuSMV driven narrative solver we presented in earlier work [29].

Optimizations we will present fall into two categories, *safe* optimizations (Section 4.1) and *unplayable* optimizations (Section 4.2). Safe optimizations simply mean that we are reducing the size of the compiled output while still functionally generating the same narrative, with identical execution behaviour. Unplayable optimizations imply that we remove certain statements and behaviours that are not necessary in order to correctly execute and verify the game, but which result in a game that is difficult for human beings to actually play. An example of an unplayable optimization we present is the removal of all output statements.

4.1 Safe Optimizations

4.1.1 Redundant Transition Removal.

PNFG statements can have complex interdependencies, not always fully captured and made disjoint by the language definition. Thus the corresponding NFG generation cannot easily take advantage of that dependency, and occasionally identical, redundant transitions can be generated. This sometimes occurs due to our simplistic code generation for compound conditional testing, and can also occur as a consequence of the application of other optimizations, particularly sequence collapsing (described below). Removal of redundant transitions is shown in Figure 4.1, and is a well-known, standard optimization on Petri Nets. In our case we must further ensure that any textual output associated with otherwise identical transitions is also identical—this would represent, however, an unusual and rare situation (concurrent output is technically possible in our system as a consequence of the use of the `forall` statement, but its value is unclear).

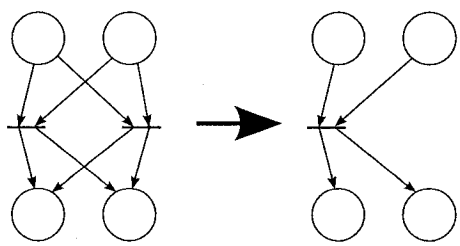


Figure 4.1: *Redundant Transition Removal.* The two transitions on the left have the same inputs and outputs, and so accomplish the same task. One of the transitions is thus sufficient, and the other can be safely removed.

4.1.2 Dead Code Removal.

Generated code that cannot affect execution behaviour of the output NFG is functionally useless, and can be safely removed. Such "dead code" can take the form of isolated places (nodes) or transitions, nodes that cannot contain a token, and which

do not constrain the firing of any connected transitions, and symmetrically transitions which cannot fire, but do not constrain the presence or absence of tokens in connected nodes.

In actual Petri Nets, one must be very careful of the constraints mentioned above. A node with no input transitions and no initial token can still have meaning by preventing its output transitions from actually firing. Given our code generation strategy, however, all transitions must be *live* (eventually, potentially fireable from the initial marking/token-assignment), or the game execution will stall—the "flow" of a token through the context nodes must continue for the game to run properly. Thus a node which can never contain a token is necessarily "dead," and can be removed, as well as any output transitions connected to such a node. Similar logic applies to nodes that cannot be "emptied" of tokens—if a transition cannot fire, and is the only output from a node that contains a token, then, due to the 1-safe nature of the output, no input transitions to that node can fire either. The schematic nature of these cases is shown in Figures 4.2 and 4.3.

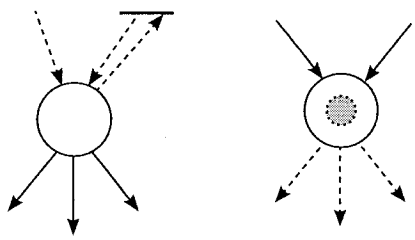


Figure 4.2: *Dead Code Removal.* Nodes are dead if all inputs transitions are dead and no token exists, or if all outputs transitions are dead. Dashed arrows represent a link to a dead transition.

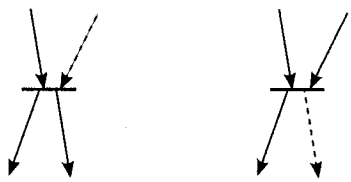


Figure 4.3: *Dead Code Removal.* Transitions are dead if any input or output node is dead. Dashed arrows represent a link to a dead node.

The removal of these dead portions of the code is done iteratively. First of all, we mark all places that are trivially dead—ones without any tokens and without any inputs (or that have only input transitions that are also output transitions), and ones which contain a token but have no outputs (that are not also inputs). We then repeatedly identify dead transitions as ones that cannot fire because an input or output place is dead, and nodes which are dead because all input transitions are dead or all output transitions are dead. Once all dead transitions and nodes are identified they are removed from the NFG output.

4.1.3 Collapsing Sequences.

As discussed above, all sequences of transitions that share a context node must be executed in turn—failure to do so would result in execution stalling, violating the general principle we use of "moving" a single token from context node to context node in order to enforce control flow.

This code generation property suggests a simple, and quite effective optimization for reducing the state space of the generated NFG: we can look for sequences of transitions connected by a single context node, and "collapse" them into a single transition. This process is shown schematically in Figure 4.4. Here *T1* and *T2* are sequentially connected only through the context *C-2*; *C-2* is itself not connected to any other transitions, *T2* has only *C-2* as a context input, and thus it is necessarily true that if *T1* fires so must *T2*. Token movements based on *T2* can thus be combined with the effects of *T1;* in particular, inputs of *T2* such as node *B* can become inputs of *T1,* and outputs of *T2* such as nodes *D* and *C-3* can become outputs of *T1*. The result is that transition *T2* and context node *C-2*, as well as any other nodes that are both outputs of *T1* and inputs to *T2* are now isolated, trivially dead code, and can be deleted.

There are a few minor complications to this process not shown diagrammatically. Any textual output associated with *T2* must be of course appended to *T1*, and nodes that are both inputs and outputs to both *T1* and *T2* must avoid being duplicated. Note that nodes cannot be just inputs to both *T1* and *T2,* nor can they be just

outputs to both transitions—in either case this would disallow *T2* from firing once *T1* had fired.

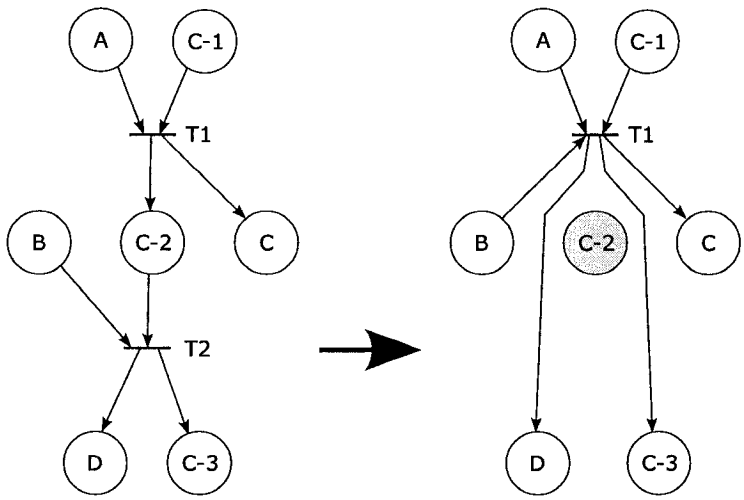


Figure 4.4: *Collapsing sequences of transitions. C-1, C-2,* and *C-3* are context nodes, while other nodes represent generic (non-context) input and output places.

4.1.4 Code Commoning.

Much of the "syntactic sugar" present in the syntax of the PNFG language is designed to help reduce duplication in code generation at the programmer/source level. A good example is the use of threads and timers, which would otherwise require many sequences of identical programmer code, and thus generated code at the end of each and every action specification.

In early versions of implementing our code generation we treated these structures very naively, simply appending each thread and timer update after each action in the NFG representation. Current code generation is more efficient, treating these blocks of code as *common code* that can be combined into a single output representation, but reused from multiple input paths. This is in fact a general optimization strategy, and is shown in Figure 4.5.

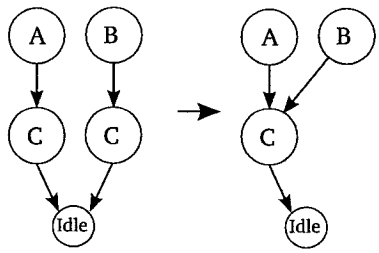


Figure 4.5: *Code Commoning.* *A, B,* and *C* represent arbitrary code blocks, with *A; C;* forming one action and *B; C* forming another action. In this case *C* can be easily commoned, with *A* and *B* redirected to a single instance of *C.*

Commoning the tails of actions has the advantage that the termination of each action is known, and identical—we simply return to the `idle` state. However, it is also possible to common arbitrary sequences of code within actions, albeit with a little more effort and cost. Figure 4.6 shows an example of how to common an intermediate sequence of code found to be identical in two different actions. Note that in order to exit from the common code and return to the appropriate sequence, an extra "context" node must be generated for each sequence. This extra context will be filled in upon entry to the common code, and consumed to branch properly on exit. Thus in order to common code *C* we need to minimally generate two extra nodes. Reductions in code size due to this optimization must therefore be balanced against the extra costs and complexities of additional code generation, as well as of course the very significant cost of locating such common code. For these reasons we have not yet implemented this optimization in its full generality, and a detailed, experimental investigation of the relative benefits and costs of general commoning is part of our future work.

As a specific form of the above generalization, however, we have experimented with code commoning for simple functions. Functions with no parameters are essentially common code where the programmer has already identified the common instance, and semantically suggested its value as common code as well. Parameters add greater complexity, and require extra contexts to identify the parameter variables if the function code is in fact common—a cost calculation becomes necessary to be

sure such common code really does result in a reduced output size. For these reasons we currently only generate common code for 0-parameter functions; other functions are implemented as macro-expansions.

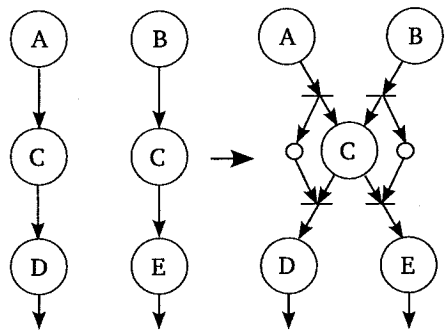


Figure 4.6: *Code Commoning.* Only a single instance of code block C is really required; however, extra nodes are necessary to ensure that upon exit from C control flows back to the appropriate code block, either D or E, depending on whether C was entered from A or B respectively.

4.1.5 No Not Nodes.

Translating our output NFG to a form consumable by the NuSMV solver relies on a few known facts in the PNFG language semantics. Specifically, different (and disjoint) sets of nodes form "mutexes," in the sense that of all the nodes in a given mutex set only and exactly one will have a token. Partitioning nodes into mutexes allows NuSMV to search the state space much more efficiently than with a naive input specification. Our context nodes, for instance, form a mutex set, as does the set of nodes corresponding to a given counter's value, and also the pair of nodes (positive and negative) generated for each object state variable.

For object locations, our default code generation produces two nodes for each object in each location, as outlined in Section 3.2.1. Thus simple mutex generation implies a mutex set for each object/location combination, and this is our default mutex generation strategy. Objects in a PNFG program, however, can only be one room at one time, and must always be in one room. As an alternative then, we could

specify mutexes based on this property, producing much larger mutex sets, and fewer of them.

In order to do this effectively we also need to change code generation: nodes representing the absence of an object in a location, or "not" nodes, are not defined or emitted. Moving an object x from location y to z then involves generating a transition relying on "x in y" as an input and producing "x in z" as an output, without requiring or modifying negative location state indicators. The main disadvantage of this technique is in the use of conditionals. In order to test whether x is *not* in y (or equivalently as the else-part of testing if x is in y), it is not possible to simply inspect the node representing "x not in y." Instead, a whole collection of transitions must be generated, each inspecting "x in z" for every $z \neq y$. This is shown schematically in Figure 4.7.

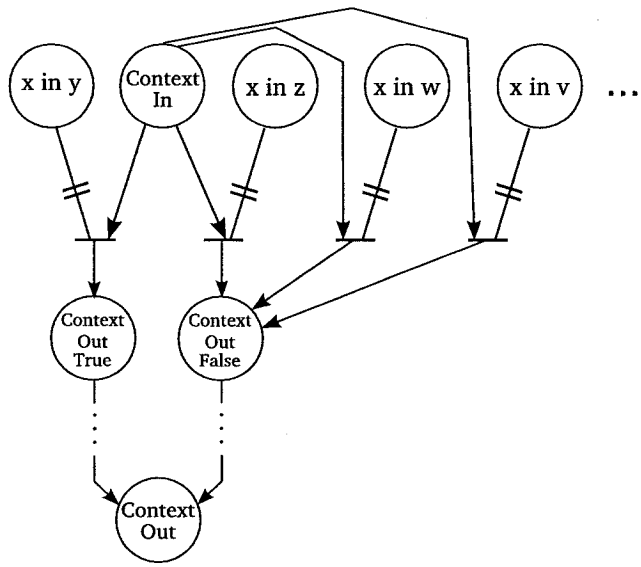


Figure 4.7: *No Not Nodes.* A conditional must test all possible locations to determine whether to follow the negative location branch.

Note that while the use of the "no not nodes" optimization (or alternative code generation strategy) may have a positive impact on the size and number of mutex sets, it has a negative effect on the number of generated transitions. In our admittedly

highly-limited experiments so far this optimization has a large measured effect on mutexes, but no obvious effect on NuSMV solution time, and so unlike the above optimizations it is not enabled by default.

4.2 Unplayable Optimizations

4.2.1 No Default Actions.

Default actions, discussed in Section 3.5.7, provide the player with feedback when entering a command that is not defined for the current room. This is convenient from a human perspective; for verification, however, it generates an excessive amount of choice. Each default action is a potential branch for verification to consider—these choices accomplish nothing, and so can be quickly ruled out, but the sheer number of default choices nevertheless has a very large impact on verification cost. From a verification perspective an extremely effective optimization is thus to no longer generate default actions. Results we will show in Section 7.2 indicate that this verification impact is not well-reflected in the size of the output NFG—mainly because default actions are trivially small.

4.2.2 No Output Statements.

For automated, computer verification actual console output is obviously unnecessary—a narrative can be analyzed and verified entirely by the actions accepted and the states reached, without need to examine or emit real output.

From this perspective we can remove all output statements from the PNFG. In practice this means we convert output transitions to simple null-effect transitions, allowing other optimizations (such as sequence collapsing) to perform more effectively. With no visible feedback human inspection of results naturally becomes more difficult, and actual game play by humans becomes extremely challenging, practically impossible in larger games. This optimization in particular is thus applied only to final, confidently bug-free versions of our PNFG compiler, optimizer, and example

narratives.

Unplayable optimizations are applied if an appropriate command-line option is specified. Prior, safe optimizations are applied by default. In terms of verifying narratives, however, it is important to know the extent to which each of these optimizations can or could contribute to making verification faster or more general. NFG size itself is not a perfect indicator, as suggested in our discussion of the elimination of default actions, but remains a primary heuristic. In the next section we investigate the impact of individual optimizations on the output NFG size.

Chapter 5
A Heuristic Narrative Solver

Previous attempts at solving game narratives [29], have used brute-force approaches on optimized representations of the narrative. As narratives become more complex, their corresponding state space grows exponentially and solving the game via a brute force approach requires significant computational resources. On the other hand, it might be possible to use extra information from a high level representation in order to reduce the search space. With that idea in mind, we developed a simple heuristic search that looks for a game solution. Our long term goal is to be able to derive complete solutions, even for complex narratives.

In Section 5.1, we define the heuristic we use to simplify the search, and formally define our algorithm in Section 5.2. We will further explain the workings of the heuristic solver by applying it to an example narrative in section 5.3. At this point, the solution produced by our heuristic solver cannot be guaranteed to be optimal or complete. We will discuss these limitations in detail in Section 5.4. Section 5.5 mentions the possible ways to extend our current approach. We also compare the heuristic solver to the other approach we have been using to solve narrative games in Section 5.6.

5.1 The Heuristic Defined

The approach consists of looking at the pnfg source code of the narrative and finding where the game can be won and finding a sequence of actions that leads back to the initial state of the game. Our heuristic tries to build this sequence of actions by looking at the if statements in the high level PNFG source file. Overall, this approach is heuristic in nature because it assumes the conditions that need to be satisfied in order to win the game are independent from each other, and that they can be satisfied independently during gameplay. The heuristic will not perform well when dependent conditions exists, because it will not be able to properly order each action to produce a valid solution.

5.2 Algorithm Definition

```
solver()
01  followCondition(game.win)
02  if the room (r) of the first action of
    the solution is not the initial room
03     followCondition(r contains you)

followCondition(conditional statement)
04  For each condition of the conditional statement:
05     Find the statement that make the condition satisfiable
06        For each statement found:
07           add the action in which
             it is contained to the solution
08           if the statement is in a conditional C
             and C has not been seen
09              mark C as 'seen'
10              followCondition(C)
11           else return
```

Figure 5.1: Heuristic Solver Algorithm

We start by looking for the statement that sets the state `game.win` to true

(*+game.win*). If a game can be won, then this statement must be present in at least one action definition, inside the PNFG source file. The function call *followCondition(game.win)* will look at the bodies of all room actions of the game, searching for the *+game.win* statement (Statement 05).

When this statement is found, the action that contains it is added to the solution (Statement 07). We then check whether or not that statement was enclosed in a conditional statement (Statement 08). If that is the case, we must now look for the statement that, once executed, will satisfy the condition needed in order to go into the branch that contains the statement that sets `game.win` to true.

We continue following conditions until they have all been satisfied. To avoid looping infinitely, we follow a given condition only once (Statement 08). We then look at the room `r` that contains the last statement to have been added to our solution and compare it to the room where the player starts the game. This initial room will be the room `s` in the statement *move you from offscreen to* `s` that must be present in the `start` action of the PNFG source file. If `r` is different from `s` (Statement 02), our solution is not complete. We must then find actions that will move the player from the starting room of the game to the room where the first action of our list can be executed. This is done by creating a new condition *(room contains you)* and calling the *followCondition* function (Statement 03). As a result, we will have constructed the full solution from the starting room to winning the game.

In the event that a particular condition is composed of a logical operator (`AND` or `OR`) the two parts (*c1* satisfied by *s1* and *c2* satisfied by *s2*) will be treated independently. In the case of an `OR`, our solution will suggest doing *s1* or *s2*. When it is an `AND`, we have no way of determining the order to place *s1* and *s2*. We cannot guarantee the validity of the solution anymore. This is the behaviour we were referring to in Section 5.1 when mentioning the solver does not perform well when there are dependent conditions.

From this heuristic approach, we see the emergence of a naive game decomposition. Our heuristic assumes that the entire game can be decomposed into two sections, the first being `A`: Get to the room where we can start executing a series of actions that lead to winning the game, and `B`: execute these actions to win the game.

5.3 Solving an Example Narrative

Consider the following game narrative in Figure 5.2. We will now demonstrate how our heuristic solver would go about finding a solution for this narrative. Internally, we are building a solution tree that reflects the different uncertainties in our solution that appear when we deal with conditional statements that use logical operators.

```
01 start {
02    move you from offscreen to street;
03 }
04 room street {
05    (you, enter){
06       move you from street to pub;
07 } }
08 room pub {
09    (you,takeoutmoney) {
10       if(you.cool || you contains wallet) {
11          move money from offscreen to you;
12    } }
13    (you,takeoutwallet) {
14       move wallet from offscreen to you;
15    }
16    (you,relax) {
17       +you.cool;
18    }
19    (you,talk) {
20       "Hello ladies";
21       if(you contains money) {
22          +game.win;
23       } else {
24          +game.lose;
25 } }
```

Figure 5.2: Small narrative that uses a conditional statement.

We first start with a root for our solution tree.

Figure 5.3: Initial configuration of solver graph

Searching for *+game.win*, we find it in the action `(you,talk)` of the room pub, so we add this action to the solution (Figure 5.4). Thus it means that (you,talk) in the room *pub* is the last command the player will need to execute to win the game. Because the *+game.win* statement is enclosed in an if statement, we now look for the statement that will satisfy the condition (`you contains money`).

Figure 5.4: The action *(you,talk)* is added

To satisfy the previously identified condition, we need to find any statement that moves the object `money` from any room 'r' to 'you'. Looking at the example in Figure 5.2, the only statement that satisfies the condition is '*move money from offscreen to you*', from executing the action 'takeoutmoney' in the room 'pub' (Figure 5.5).

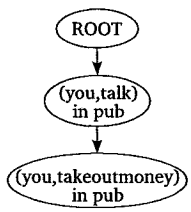


Figure 5.5: The action *(you,takeoutmoney)* is added

Again, that statement is enclosed inside an if statement, but this time, the if statement has two conditions. We simply consider the two conditions independently, and we will add the actions that satisfy the condition as branches in our solution, illustrated by Figure 5.6.

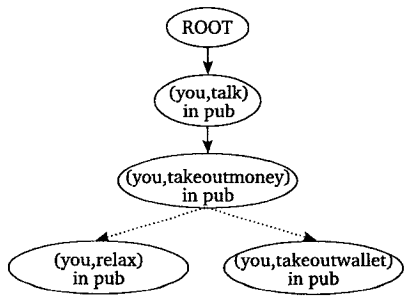


Figure 5.6: The actions that satisfy the conditions of the OR have been added.

This effectively concludes the first section of our solution, and we must now determine the initial room. In our example, the room `street` is our initial room. We now look at the rooms where the first actions of our current solution take place and determine whether or not they correspond to the initial room. In both cases, the player must be in the room `pub`. We create a condition *(pub contains you)* and find that the action `enter` in room `street` will satisfy the condition. Since the action is in the initial room, the solver terminates and we are left with the solution in Figure 5.7.

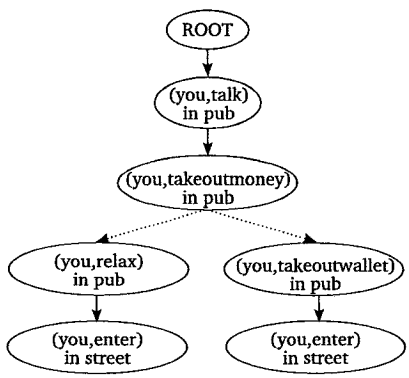


Figure 5.7: The action to get from the initial room to `pub`.

5.4 Limitations

In its current form, the solver does not keep track of the current game state, and this information would certainly allow it to perform better. For example, the solver will not be able to detect a looping sequence of statements for the solution, and a given non-movement action cannot appear more than once in the game solution. A side-effect of this limitation is that we maintain a list of conditional statements we have already visited (Statement 04 in Figure 5.1). We avoid visiting the same condition twice, but in practice, a certain condition might need to be evaluated many times, especially when dealing with counters.

The solver ignored all threads and timers, as well as global actions. This limitation was mainly motivated by the fact that most of the example narratives available to us do not use these features, or if they do, they play secondary roles in the narrative and are not required to win the game. Also, the heuristic solver in its current form was meant to be a proof of concept exploring the possibilities and overall usefulness of high level information as a way of solving games, as opposed to the brute force approach of the NuSMV model checker.

5.5 Possible Extensions

Maintaining information about the game state as we search for a solution would allow us to generate solutions for more complex narratives. Since this algorithm is a result of wanting to make use of extra information available to us, maintaining a game state for each part of the solution seems like the next logical step if we were to extend the solver.

Another possibility would be to use our heuristic approach to generate hints in real-time during gameplay. Using the current game state, the solver could check which conditions need to be met in order to move closer to the game winning action and then indicate the action that needs to be executed next. The number of conditions that have been satisfied at a given point in the game could be used as a measure of the player's progress in the game.

Finally, an easy way to resolve some uncertainties in our solution would be to verify the solution by simply playing it automatically. As we have seen, the solution tree can have more than one action at a given level, but it might be the case that only one of these actions applies when we play the game.

5.6 Heuristic Solver Versus NuSMV

The heuristic solver presented in this section was developed as an alternative to our other method of building narrative game solutions automatically, using the NuSMV symbolic model verifier [9]. NuSMV uses a very efficient Binary Decision Diagram (BDD) [8] representation to reduce the state space of the input it receives, which allows it to analyze larger problems. One of the properties it can derive is reachability, which is exactly what we try to accomplish when looking at a NFG representation in order to find paths that lead to winning.

The problem we rapidly encountered with this approach is that the size of NFGs grows to be very large as the narrative increases in size and complexity. As a result, the NFG we feed to NuSMV is simply too big, and cannot be analyzed formally. Still, neither of the two approaches should be ignored and we will now analyze their different strengths.

NuSMV is a highly generic solver and has been applied to a wide variety of problem domains, which gives us a lot of related work from which we can probably find some ways to further optimize our representation. Also, NuSMV will produce the game solution which has the fewest number of moves, while our heuristic solver cannot guarantee that the solution it finds is minimal, or even in the right order. Using this model checker makes it very easy to identify dead code inside the representation we give it, since dead code is also a reachability problem.

NuSMV is not able to find solutions for large narratives, because we are using the full representation of the game as the input. For our heuristic solver, we focus on the dependencies between conditions that originate from the *+game.win* set statement. This represents the fundamental difference between the two approaches: NuSMV uses

a rigorous approach to find the minimal solution, while our solver uses a heuristic algorithm to give a solution that represents an indication of what the player needs to do in order to win the game. Also, the heuristic solver operates on the PNFG source file, which allows it to use an abstract representation much smaller than the corresponding NFG, especially with the PNFG language improvements we presented in section 3.5.

Another important difference between the two approaches are the resources needed for their execution. The heuristic solver only operates on the PNFG source file, which makes it very fast and very compact. NuSMV deals with a compact representation of the full state-space of the narrative. As a result, the model checker has very high memory requirements, and is very slow to produce a solution, even for small narratives. We will show the differences in execution time in our experimental results (Section 7.3).

In its current implementation, our heuristic solver is still in its early stages, but it plays a very important role in our narrative analysis research. It allows us to determine the benefits of using high level information to rapidly find a solution, as opposed to a costly brute-force method. The best solution is probably a combination of both approaches, where we would use a rigorous solver that operates on a highly reduced version of the game narrative to produce the minimal solution, as shown in Figure 5.8.

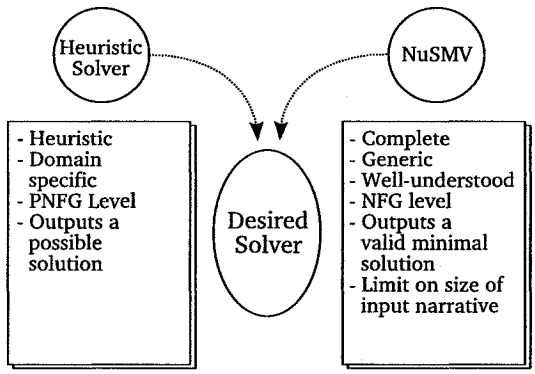


Figure 5.8: The different advantages of each the two solving approaches we use.

Chapter 6
Narrative Metrics

The underlying goal of developing the PNFG framework has always been to derive properties about game narratives. We have developed a metrics framework that uses information found in the PNFG high-level representation of narratives in order to derive interesting properties about these games. For example, we wish to analyze the number of possible actions available to the player during the game, in order to look at the convexity of the game. We will present this framework in Section 6.1, and then look at the different metrics we measure in Section 6.2.

6.1 Metrics Framework

Below we discuss the design of our metrics framework and how it operates with the PNFG system. This interaction allows us to use high-level information about narratives that was previously ignored when doing analysis. The framework itself gives us a representation that is much easier to work with than dealing directly with PNFG source code, or the corresponding NFG. We will now go over the details of this framework by explaining the *GameTree* representation we use to derive metrics, and the role of each component of our framework.

An overview of the metrics framework can be seen in Figure 6.1. The high-level intermediate representation used internally by the PNFG compiler is also used by the metrics framework in order to build a GameTree which is then fed to the metrics

analyzer. From this representation of the game, we can then derive different metrics, produce an html report containing all the results, as well as a graphic representation of the GameTree, which is generated using the `dot` [15] tool.

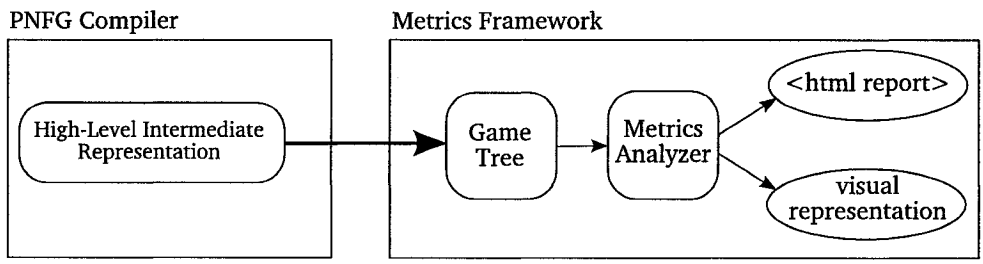


Figure 6.1: Metrics Framework Overview.

The need for a new representation of the narrative originated from the metrics we wanted to measure. Deriving metrics from a Narrative Flow Graph is certainly possible, but we wanted to have a representation that still contained high-level information in an organized manner to make the metrics development much faster.

6.1.1 The GameTree Representation

The first stage of metrics analysis consists of building a GameTree, which corresponds to a tree representation of all possible action paths from the start of the narrative to winning or losing the game. This structure is composed of *GameNodes*, representing the game state in terms of the values of all variables in the game, and *GameEdges*, which symbolize actions executed by the player. The relationship between these elements is shown in Figure 6.2. We will now describe the different components that are used to generate this representation (Figure 6.3). As we go along, we will also explain how the generation of the GameTree takes place. Note that currently, the generation process of the GameTree ignores default actions, threads, and timers, which contribute to a large state explosion and prevent us from deriving any metrics.

GameTree Loader

Internally, the GameTree is represented by a set of GameNode objects. The GameTree Loader is responsible for building the GameNodes that represent the game

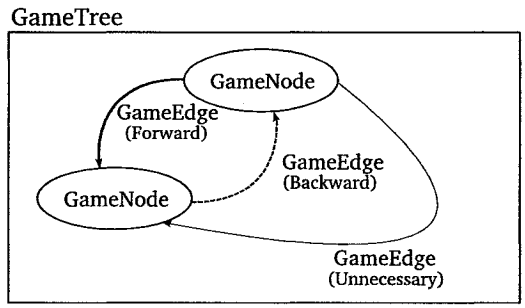


Figure 6.2: Elements of a GameTree.

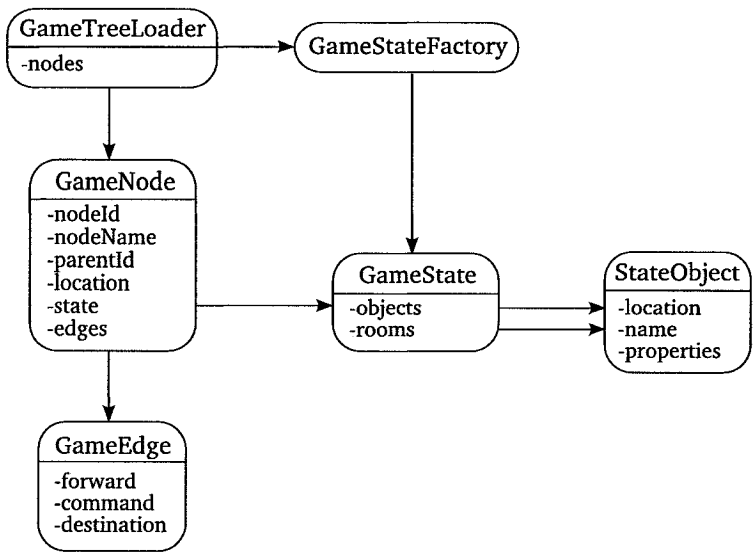


Figure 6.3: Metrics Framework Components.

narrative. We use the high-level intermediate representation of the PNFG compiler, which contains all the game construct definitions. Starting from the `start` room, it will do a breadth-first search traversal of the game space. In this GameTree we are building, the edges represent the execution of an action, and the nodes represent a particular game state.

A very important detail to note, we do not allow a particular GameState to be repeated in our GameTree. We discuss handling these redundant states in our description of GameEdges.

Game State Factory

When an action is executed, it may lead to a new GameState or to a state that has previously been encountered. So, before a GameNode is actually created, we need to evaluate the GameState that this node will represent. This is done by the Game State Factory, which receives the current GameState, and updates it by evaluating each statement of the action. In practice, we are playing the game and evaluating every possibility by interpreting the statements of the action. We start by creating a GameState that contains all the game objects, and all the game rooms with the default values for each of their properties. We also keep track of all the GameStates seen so far, and if a GameState as already been encountered, it is discarded, and no GameNode is created. This ensures termination of our tree-building algorithm.

GameState and StateObject

The GameState is composed of two sets representing all the objects, and all the rooms respectively. Inside these sets, we store StateObjects, which act as containers for the current state of either an individual room, or an individual object. Each StateObject has a name and a specific location that corresponds to the room in which it is currently contained . Also, a StateObject has a set of properties where each entry has the name of the property and its value. This representation of the GameState, shown in Figure 6.3 is structured very similarly to the PNFG source code, and retains its straight forward organization.

GameNode

As mentioned previously, a GameNode represents a unique GameState. It has a link to its parent, as well as a list of GameEdges, which represent the possible actions

that can be executed at this particular node.

GameEdge

Having a GameEdge object defined explicitly gives us more flexibility, and allows us to have edges with more information in the GameTree we are producing. We have defined three different types of edges: *forward, backward, and unnecessary* (see Figure 6.2). When a GameNode is successfully created, we add a forward edge that goes from that new node's parent to the new node, and store that edge in the parent's list of edges. When a GameState is found to be identical to another one previously created, we first consider the GameState's level in the GameTree. If it is found to be at the same level than the original state, we create an unnecessary edge, since the same GameState can be reached using another path in the graph. If the redundant state's level is lower than the original state, we create a backward edge that goes to the GameNode that has the original GameState. Since we traverse the game in a breadth-first search manner, this ensures that for a given state, we will create the GameNode for its first occurrence in the closest position to the root of the tree. By definition, the total number of forward edges will always be one less than the total number of GameNodes.

GameTree Generation

In our description of each GameTree component, we have briefly explained the details of the generation of the tree. You will find a complete description of this process in Figure 6.4.

6.2 Narrative Game Metrics

In the following section, we will present the different metrics we are currently able to measure using our metrics framework.

6.2.1 Edges to Nodes ratio

The edges to nodes ratio is a metric that allows us to quickly have an idea of the shape of the GameTree, and also hints a certain complexity in the narrative. A higher node

```
buildGameTree()
01  Build initial GameNode g
02  loadFromAction(start,g)
03  while(queue is not empty)
04      loadFromAction(dequeue())

loadFromAction(Action a, GameNode g)
01      Using a, update the GameState s, found in g
02      if s already exists
03          if the level of s is equal to that of
            the original state o
04              create an 'unnecessary' edge e from g to
                the GameNode associated with o
05          else
06              create an 'backward' edge e from g to
                the GameNode associated with o
07      else
08          create new GameNode g' with g as parent
            and s as state
09          create a 'forward' edge e from g to g'
10          for all the actions a' available in g'
11              enqueue( (a',g') )
```

Figure 6.4: GameTree Building Algorithm

count indicates a large narrative, while the number of edges represents the number of actions. Therefore, as the ratio becomes bigger, it means that there are less actions that have an impact on the game state.

6.2.2 Forward to Backward edge ratio

Our metrics framework allows us to easily differentiate between edges that lead to a new game state, and those that bring the player back to a previously encountered state. If there are much more backward edges, it probably equates to a complex game, since many actions will not make the narrative progress.

6.2.3 Convexity

The concept of convexity as a way of analyzing game narratives has been presented in [30], but no actual data has been presented to show that convexity can be actually observed in game narratives. The idea behind convexity is that number of possible choices the player has starts out small, increases, and then converges to winning of the game.

In [30] the concept was presented from a very high-level perspective, where the term *convexity* represented what was being referred to as the *classical game structure*, as shown in Figure 6.5. An actual measurement for convexity was not provided, and the authors did not address the fact that most narratives have "useless" actions that do not move the player closer to winning the game. While convexity looks to analyze the number of choices available to the player, the description from [30] does not mention if that number corresponds to the number of choices from the entire narrative, or if it is limited to one play sequence. It could also represent the average number of choices the player has after having entered a certain number of commands.

It was also argued there that this convexity structure could be serialized throughout a game as a series of "levels", "acts", or "worlds" (Figure 6.6). One of our goals in analyzing game narratives has been to look for chapters inside games, and the detection of these chapters could be possible using such a convexity metric, that focuses on shape of the narrative. Since no standard measurement of convexity exists, we will

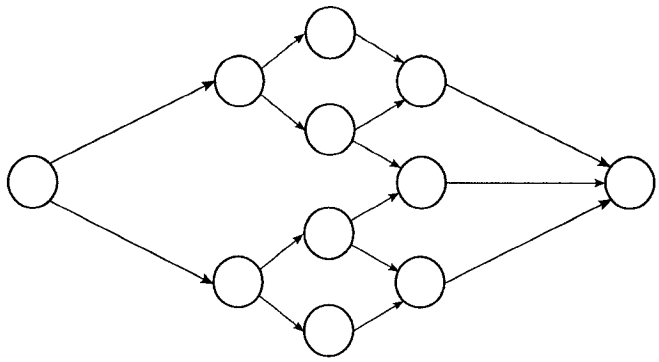


Figure 6.5: A convexity, as defined in [30]

use two different measurements. First, we count the total number of GameNodes at each level in the tree, where a level is the number of actions that have been executed to get to a certain GameNode. The second one consists of looking at the convexity of individual winning paths, and we will discuss the details of this approach in the next section.

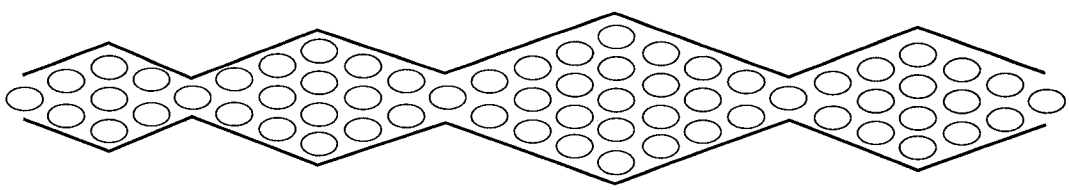


Figure 6.6: A series of convexities, as defined in [30]

6.3 Analyzing Winning Paths

From our first experiments with measuring metrics, we found out that our example narratives have more than one solution, which we define as Game Winning Paths in terms of the GameTree. More precisely, a Game Winning Path is a sequence of actions going from the start node to a winning node, and in which all actions affect the game state. Also, a Game Winning Path cannot visit the same game state

twice. In the event where no such path could be found, we can declare the game as being unwinnable. The metrics framework we are using allows us to produce a tree representation of our game, and using forward edges, we can easily track down paths that lead to winning the game. Using these game winning paths, we look to further analyze the overall narrative.

6.3.1 Convexity of Winning Paths

When we analyze the shape of the GameTree, it can be hard to find a well defined pattern like that of Figure 6.5, because many "useless" actions give false positives when measuring convexity. By focusing on game winning paths individually, we look to analyze their convexity as well. This is the second convexity measurement we referred to in the previous section. To measure convexity, we again count the total number of GameNodes at each level in the tree.

6.3.2 Subsets and Supersets

In most case, there are many solutions to a game, but since we keep track of the full game state, many solutions can be very similar, to the point where they only differ due to the player having a non-essential item in his inventory or not. By keeping track of the subsets and supersets of each game winning path, it becomes much easier to draw links between each path. Figure 6.7 illustrates this relationship between paths, where a node represents a particular game winning path, and an arrow indicates the source path is contained in the destination solution.

6.3.3 Analyzing the Shortest Winning Path

While the length of the shortest Game Winning Path can give us an idea of the complexity of the game, we will also look at the effects of changing a side quest from being optional to mandatory. Our goal is to determine if we can isolate game quests by looking at the actions that make up the shortest game winning path.

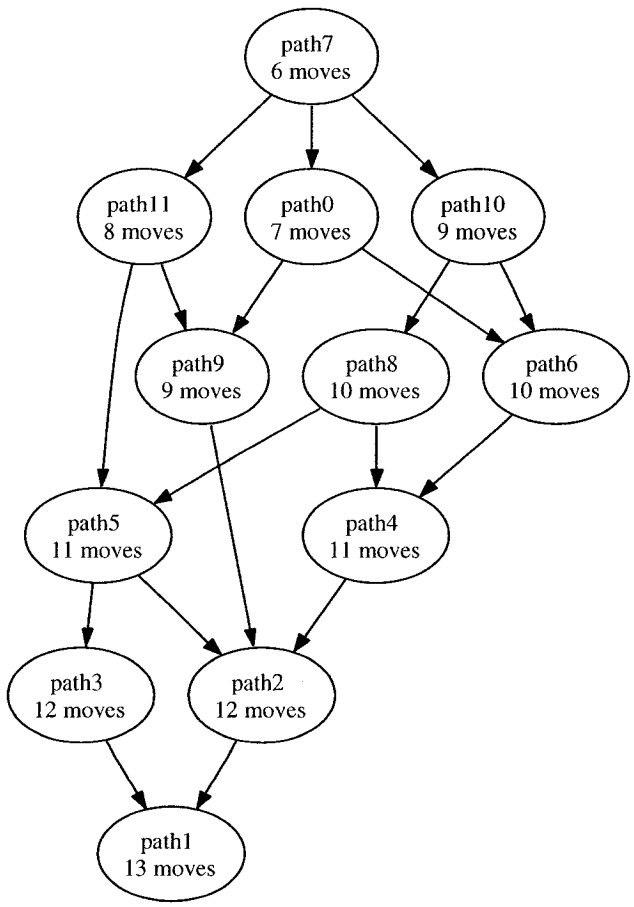


Figure 6.7: Dependencies of Game Winning Paths for *Return to Zork, chapter 1.*

Chapter 7
Experimental Results

The experiments we have done with our system consist of measuring the effects of the different optimizations, looking at the quality of the solutions produced by our heuristic solver, and measuring the different metrics we present. We also analyze the different Game Winning Paths, and make observations about the solutions of the games we used for our experiments. We will first describe the example narratives we have used for our experiments in Section 7.1, and then proceed with the results for our optimizations (Section 7.2), heuristic solver (Section 7.3), and metrics (Sections 7.4 and 7.5).

7.1 Example Narratives

To test our system we have used four different game narratives, ranging from from very small to fairly large. We will present each game along with their different properties and usefulness in our experiments. We will also present maps of each narrative, which have been generated using the PNFG compiler to get the list of actions that move the player, and the `dot` directed graph generation tool [15]. In theses maps, the nodes represent the different rooms in the game and an edge indicates that it is possible to move from room a to room b using the command that appears on the edge, near the source node. For example, in Figure 7.1 the player can move from the *cloakroom* to the *foyer* by entering the `e` command, assuming that the current game state satisfies

all conditions that apply to this movement. We will also call on Tables 7.1 and 7.2 (page 68) for data on the main properties of each narrative. We played through our implementations of theses narrative manually to ensure winning was possible.

1) *Cloak of Darkness* (`CoD`), acts as our base case in all of our experiments, and it was originally designed to be an example that would help programmers learn the syntax of various IF toolkits [14]. In the game (see map in Figure 7.1), the player starts by wearing a cloak, and as long as she is wearing it, the bar remains in darkness. The player can score a point by hanging the cloak in the cloak room at which point the bar becomes lit, and score another point by reading a message in the bar once it has been lit. Every time the player executes a non-movement action or an invalid movement action in the darkened bar, a counter is incremented. If the value of the counter is above a certain limit, reading the message cannot win the game.

Having only three game objects and three game rooms, CoD can be classified as being very simple. On the other hand, it has all the basic features we find in works of IF, such as room and object interaction, player movement, actions which have non-local effects, object with states, as well as some counting. The game map of CoD can be seen in Figure 7.1. We also include a more detailed version of CoD that heavily relies on the use of functions and global actions in `CoD(func)`. The PNFG and NFG data for both version of CoD can be seen in columns 1 and 2 of Tables 7.1 and 7.2.

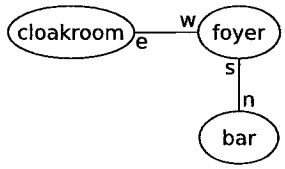


Figure 7.1: *Map for Cloak of Darkness. Taken from* [29]

2) *Return to Zork* (`RTZ`) [5] was released in the midst of the CD-ROM era of narrative games, and heavily relied on the use of video and sound to deliver to story. It is set in the same world as its prequel, the IF classic *Zork* [20]. While CoD was chosen for its overall simplicity, representing RTZ in PNFG source code was motivated by the following two reasons. First of all, it gave us the opportunity to verify the feasibility

and challenges of translating a multimedia game into a strictly text-based game. Also, RTZ has previously been divided into chapters [36], another narrative game feature that interests us. We will be using the first two chapters of the game.

2.1) *Return to Zork - Chapter 1* (`RTZ-01`) is a fairly simple narrative where the objective consists of building a raft in order to reach the town of West Shanbar, while avoiding the deadly *Road to the South* (See Map in Figure 7.2). We also include an alternate version of the first chapter, `RTZ-01 (bonding)`, where the object *bonding plant* must be present in the player's inventory in order to win the game. Also the bonding plant must be alive, which is accomplished in the game by digging the plant as opposed to cutting it. In the original game, the bonding plant is not required to finish the chapter, but the player needs to have it in order to eventually win the game in the final chapter. The data for both versions of the narrative can be found in columns 3 and 4 of Tables 7.1 and 7.2 respectively.

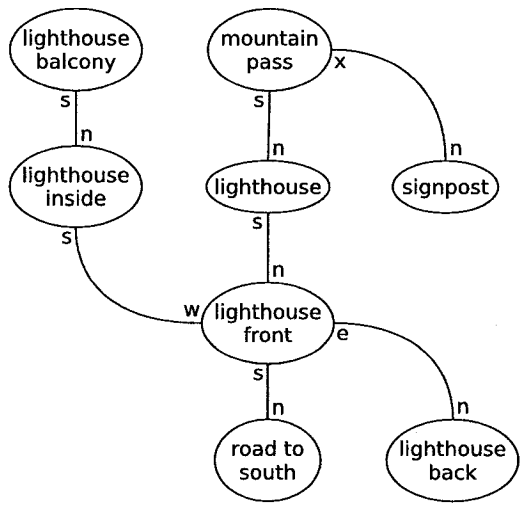


Figure 7.2: *Map for Return to Zork - Chapter 1 Taken from* [29]

The underlying goal of having this modified version is to analyze the impact of adding a requirement to winning the game. To accomplish the goal of having the bonding plant in his inventory, the player must first pick up a rock and throw it at the vulture that guards the plant. In the original `RTZ-01`, the action of picking up

the rock was implemented in the PNFG source file as the global action `take`. Since our heuristic solver and metrics framework ignore global actions, we added special versions of two global actions `take` and `drop` in the room `mountainpass`. These versions only deal with the object `rock`, and it allows our metrics framework to detect the needed actions to win the game. We will discuss the impact of making this side quest mandatory in Section 7.6.

2.2) *Return to Zork - Chapter 2* (`RTZ-02`) is more complex than the first chapter, because the player must take part in a drinking game, where a series of commands must be executed the right number of times in order to get a key that leads to the next chapter. The map of the narrative in Figure 7.3 reveals that the different drawers of a filling cabinet were represented as individual rooms. When a player consults a file, a set of output statements is used. This a very good example of actions, and rooms that do not affect the game state, because reading the different files is optional. Column 5 of Tables 7.1 and 7.2 show the basic PNFG and NFG data.

3) *The Count* (`Count`) [3] is a famous work of interactive fiction by Scott Adams where the player must confront count Dracula. It is particularly interesting for us, because it is fairly complex, to a point where we cannot currently solve it automatically with our system. In terms of number and rooms and objects, The Count is not much bigger than the second chapter of RTZ, but its NFG data from column 6 of Table 7.2 reveals that it is much more complex than any of the example narratives.

Many of the language enhancements we presented have been added in order to facilitate the representation of The Count in PNFG source code. As a matter of fact, we can see in column 6 of Table 7.1, that the narrative uses all the PNFG language features, such as threads, timers, and functions. Figure 7.4 shows a map of the game. The game itself contains properties we wish to be able to verify automatically using our system. For example, the game can become *pointless* when the player loses an essential item, or when a certain timer runs out. The Count also has some logic flaws, a narrative consistency problem we also wish to be able to determine. In terms of finding the solution automatically, the 180-step solution of The Count represents a long term target for our system, and will also motivate further exploration of more efficient ways to solve narrative games. While The Count is too large to be used in

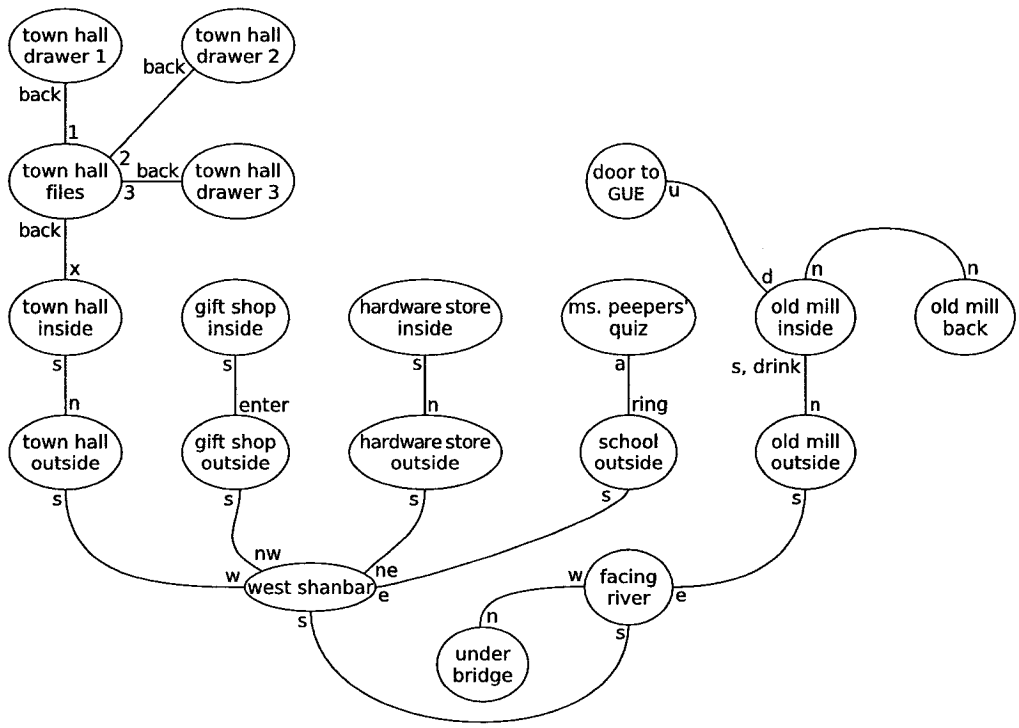


Figure 7.3: *Map for Return to Zork - Chapter 2 Taken from* [29]

most of our tests, the task of representing it with PNFG has already yielded several optimizations, and it will certainly continue to be a great source of improvements in the future.

Using Alternate Versions of Example Narratives

For the optimization results, we only considered the "complete" version of each narrative. We have therefore ran our experiments on `CoD(func)`, `RTZ-01`, `RTZ-02`, `The Count`. The alternate versions of `CoD` and `RTZ-01` will be used in our results for the heuristic solver and metrics. We needed `CoD` for the solver and metrics experiments because these two modules currently ignore global functions, which `CoD(func)` uses extensively.

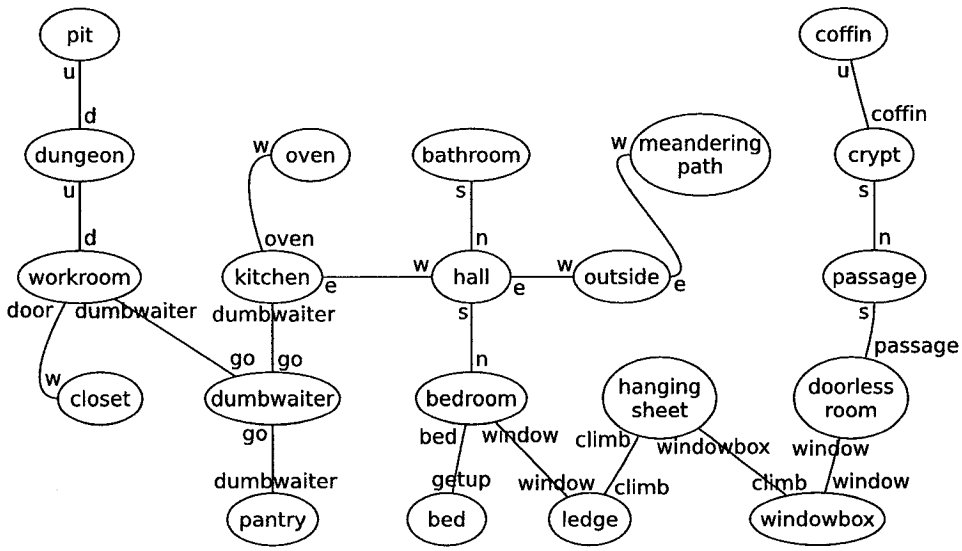


Figure 7.4: *Map for The Count. Taken from* [29]

7.2 Optimization Results

General statistics about each narrative are shown in Tables 7.1 and 7.2. The "BDD Booleans" gives the sum of $\lceil \log_2 |m| \rceil$ for all mutex sets m in the NuSMV representation; this gives a rough sense of narrative complexity (in conjunction with the number of transitions), and also an indication of the size of the state space that may have to be searched for each game. BDD boolean values themselves are most dramatically affected by the *no not nodes* optimization, and so for the optimizations we investigate below we concentrate on transition and node changes.

Methodology for results

Since the optimizations build incrementally on one another, and thus have large interdependencies, a separate analysis of each optimization would be misleading, and moreover would not sum to the total effect of applying all optimizations together. Instead we generated an NFG using all the implemented safe optimizations (except for "no not nodes"), and then tested the impact of each one by removing it from that all-inclusive optimization. We also considered the incremental impact of available

property	CoD	CoD (func)	RTZ-01	RTZ-01 (bonding)	RTZ-02	Count
rooms	4	4	10	10	21	22
objects	1	1	19	19	36	29
threads	0	0	0	0	0	8
timers	0	0	0	0	0	4
functions	0	20	2	2	2	1
global actions	1	15	5	5	4	29
PNFG lines	218	535	563	583	1133	1966
steps to win	6	6	6	11	19	180

Table 7.1: *Basic PNFG data on example narratives.* The number of steps to win has been calculated from the optimal solution, except in the case of *The Count*, where we cannot guarantee our solution is optimal, because we cannot solve it automatically yet.

property	CoD	CoD (func)	RTZ-01	RTZ-01 (bonding)	RTZ-02	Count
places	123	274	965	944	1764	12603
transitions	147	335	1498	1470	3806	40762
BDD booleans	25	29	181	175	241	686
verifiable	yes	yes	no	no	no	no

Table 7.2: *Basic NFG data on example narratives.* Each graph was generated using all the optimizations available. We will present the effects of each optimization in Section 7.2

unplayable optimizations such as the removal of output statements and exclusion of default actions.

Effects of Redundant Transition Removal

Table 7.3 shows the impact of the basic redundant transition removal optimization. Unsurprisingly, since this optimization affects only transitions, there is no change in the number of nodes. Number of transitions, however, is significantly improved, up to a little over 31% in RTZ chapter 2. Unfortunately, the largest narrative (Count) does not show an equally large or larger improvement, mainly due to its more complex

	CoD (func)	RTZ-01	RTZ-02	The Count
Places	274	965	1764	12603
Difference	0%	0%	0%	0%
Transitions	347	1882	5538	46921
Difference	3.5%	20.4%	31.3%	13.1%

Table 7.3: Effects of redundant transition removal on NFG Size

control structure.

Effects of Dead Code Removal

	CoD (func)	RTZ-01	RTZ-02	The Count
Places	278	986	1800	12714
Difference	1.4%	2.1%	2.0%	0.9%
Transitions	338	1564	3951	41021
Difference	0.9%	4.2%	3.7%	0.6%

Table 7.4: Effects of dead code removal on NFG Size

Dead code removal has a disappointingly minimal impact. From Table 7.4 the lack of dead code removal only results in a narrative (NFG) between 1% and 4% larger than a fully-optimized version. In a general sense the programmer will of course write statements that are designed to be executed at some point, and so dead code should be minimal. Most identified dead code is likely due to programmer errors or imprecision in specification of sets. As well, even for objects or game events that are not actually used in a significant or important way, the existence of feedback messages or other error handling within the game will result in otherwise unnecessary nodes and transitions being conservatively identified as live. More aggressive "useless" as opposed to actually *dead* code identification would likely have a much larger impact, and is one of our main directions for future work.

Effects of Collapse Sequences of Transitions

	CoD (func)	RTZ-01	RTZ-02	The Count
Places	440	1742	3421	20885
Difference	37.7%	44.6%	48.4%	39.7%
Transitions	501	2275	5463	49044
Difference	33.1%	34.2%	30.3%	16.9%

Table 7.5: Effects of Collapse Sequences of Transitions on NFG Size

The sequence collapsing optimization has a fairly large impact on game narratives. Table 7.5 shows that the number of nodes are reduced by 38%–48%, and transitions by 17%–34%. This very nice effect is of course to be expected given our naive code generation—NFG output for each statement is generated in isolation, and so non-branching sequences of PNFG statements will naturally translate to sequences suitable for optimization. Better results are obtained on both *Cloak of Darkness* and the *Return to Zork* games. This is to some extent likely due to the fact that these games, as opposed to *The Count,* have a lot of dialog, often coded as multi-line output statements.

Effects of Code Commoning

	CoD (func)	RTZ-01	RTZ-02	The Count
Places	319	3401	15201	—
Difference	14.1%	71.6%	88.4%	—
Transitions	413	6452	25209	—
Difference	18.9%	76.8%	84.9%	—

Table 7.6: Effects of code commoning on NFG Size

Table 7.6 shows that code commoning can provide quite spectacular results. Size increases of up to 89% of nodes and 85% of transitions are possible without commoning,

and for *The Count* code commoning is in fact essential for producing any output in a reasonable time.

The main source of this benefit is in how the code for actions is generated. For simplicity in code generation, each room includes a specific copy of the code for every possible action—in larger narratives with many rooms and many possible user commands this can have a very significant cost. Code commoning allows the bodies of identical actions to be reused, effectively resulting in only one action body for each distinct action, irrespective of other conditions such as player location that control whether the action can be executed. Commoning in fact has a greater impact not shown in Table 7.6—threads and timers also benefit from a form of commoning, and this effect is not included in the data above.

From these encouraging results we wish tohe investigate other possible uses of code commoning in order to reduce the size of the NFGs we generate. The main difficulty at this point is to efficiently find patterns inside game narratives that can be commoned.

Effects of Commoning Functions

	CoD (func)	RTZ-01	RTZ-02	The Count
Places	309	965	1764	12647
Difference	11.3%	0%	0%	0.3%
Transitions	446	1498	3806	41124
Difference	24.9%	0%	0%	0.9%

Table 7.7: Effects of commoning functions on NAG Size

As we can see in Table 7.7, useful reductions can be made to the generated NFG when we apply code commoning to functions, even when limiting that strategy to functions without parameters. This does not apply in all cases of course—there must obviously be a significant number of user-defined functions in the PNFG source file in order to achieve good results. The two examples from *Return to Zork* have no

such functions, *The Count* has only 1 function used in only 2 places, while *Cloak of Darkness* makes extensive use of functions for most game activities. The performance of the optimization directly mirrors this pattern of function usage.

Effects of Unoptimizing

	CoD (func)	RTZ-01	RTZ-02	The Count
Places	654	1772	3441	21342
Difference	58.1%	45.5%	48.7%	40.9%
Transitions	812	2742	7343	56442
Difference	58.7%	45.4%	48.2%	27.8%

Table 7.8: Effects of turning off all the above optimizations, except for basic code commoning, on NFG Size.

Table 7.8 gives data for our narratives when compiled with all optimizations turned off, excepting basic code commoning. The latter optimization is included regardless since it has such a large impact, and is necessary to compile *The Count* at all.

Lack of optimizations results in output on the order of twice as large as optimized output. Thus, while the optimizations described above have quite variable effects, and depend greatly on narrative programming style and choices, the overall effect is quite significant, and well worth applying.

The next two Subsections describe the effects of "unplayable" optimizations. These are presented in a positive, rather than negative form, added in rather than subtracted out from a default usage. This represents their intended application as extra effects applied only during analysis rather than as part of the normal compilation process.

Effects of No Default Actions (Incremental)

Default actions are generated to provide simple, error feedback to the player when they enter an action that is undefined in their current situations. This has different

	CoD (func)	RTZ-01	RTZ-02	The Count
Places	274	963	1762	12601
Difference	0%	0.2%	0.1%	0%
Transitions	335	1400	2687	40181
Difference	0%	6.5%	29.4%	1.4%

Table 7.9: Effects of no default actions on NFG Size (*incremental*).

effects depending on how the game is defined. In the case of *Cloak of Darkness* and to only a slightly lesser extent *The Count,* there is sufficient error handling already built into the game specification to obviate much of the use of automatically-generated default actions. In *Return to Zork* chapter 2 in particular, almost no explicit error handling is provided by the game programmer, and default actions have a significant impact. Note that default actions are quite small, consisting of a single output statement, and are also subject to commoning. The large impact in *Return to Zork* chapter 2 can be more correctly attributed to the combination of a large number of rooms and a large number of room-specific actions.

Effects of No Output Statements (Incremental)

	CoD (func)	RTZ-01	RTZ-02	The Count
Places	221	658	1246	9897
Difference	19.3%	31.8%	29.4%	21.5%
Transitions	283	1191	3288	38056
Difference	15.6%	20.4%	13.6%	6.6%

Table 7.10: Effects of removing output statements on NFG Size (*incremental*)

Even though the narratives would be unplayable by a player we can see from Table 7.10 that the removal of all output statements leads to a significant reduction of both the number of nodes and the number of transitions in the generated NFG. The impact of this optimization certainly motivates further exploration of these types of

simplifications, which can be described in broad terms as the removal of everything that is useless for verification.

This topic of identifying useless components of computer game narratives is in itself a very relevant field of computer game analysis. Being able to identify what is meaningful or not when it comes to winning or losing a game can tell us a lot about game design patterns that are found in many games, including the general structure of individual game tasks, and how "chapters" or other logical divisions may be incorporated or identified. This is certainly an area of work we wish to further explore.

7.3 Solver Results

The heuristic solver we developed uses a very simple approach and makes an attempt at finding a solution for a narrative game, and assumes a rather simple narrative structure in order to generate this solution. The following tables contain the solutions produced by our heuristic solver. We first present the optimal solution, and compare it against the solution given by the solver. The strategy used by our heuristic solver consists of first finding the actions that need to be executed in order to satisfy the winning conditions. The second part consists of adding the movement actions in order to get to the rooms where the actions must be executed. Running times for optimal solutions were obtained using the NuSMV model checker. We will use the notation --- when no solution was produced by NuSMV. This notation is also used for the validity of the optimal solution, because it is valid by definition. An invalid solution means entering the commands does not lead to winning the game. The running times were obtained on a 1.2GHz AMD Athlon machine with 512MB of memory.

Solution for CoD

In CoD, the player starts out in the Foyer, and must go west to the cloakroom (`w`), where she can then hang the cloak on the hook (`put`). She must then move to the Bar (`e`, `s`) and read the message on the floor to win the game (`read`). If the player

	Optimal Solution	Heuristic Solution
01	w	w
02	put	put
03	e	e
04	s	w
05	read	wear
06		drop
07		e
08		s
09		read
Valid	—	no
Time(ms)	1397	176

Table 7.11: Solutions for *CoD*.

enters two or more invalid actions in the darkened Bar, reading the message will cause him to lose the game. The much longer heuristic solution in Table 7.11 is due to the structure of the PNFG source code. The `game.win` statement is nested inside two conditions, and they can both be satisfied by the executing the `put` command. However, since our heuristic solver does not keep track of the game state as it builds the solution, it looks to satisfy both conditions independently. This also explains why the set of suggested actions can contain actions which satisfy the same condition. The very important detail to note from the results for `CoD` is the running time for both approaches. It shows the heuristic approach is much faster than NuSMV. By working with the PNFG source instead of the entire state space, the heuristic solver can find a solution very rapidly.

Solution for RTZ-01

In the first chapter of Return to Zork, the player must build a raft to reach the city of West Shanbar. After reaching the back of the lighthouse (`s`, `s`, `e`), the player can build the raft (`cut vines`, `tie vines`), and finally ride it to win the narrative (`ride`

	Optimal Solution	Heuristic Solution
01	s	s
02	s	s
03	e	e
04	cut vines	cut vines
05	tie vines	tie vines
06	ride raft	ride raft
Valid	—	yes
Time(ms)	—	185

Table 7.12: Solutions for *RTZ-01*.

`raft`). This version of the first chapter has an optional side-quest where the player can get a bonding plant. It is possible to lose the game by removing the bonding plant while the vulture is still present (`cut bondingplant` or `dig bondingplant`), killing an npc, or venturing on the *Road to the South*. As we can see from Table 7.12, the heuristic solver was able to generate a valid and optimal solution. This is explained by the fact that `RTZ-01` uses a simple structure. For example, all its conditional statements verify only one property at a time, and the narrative itself does not use advanced PNFG language features such as counters, threads, or timers. Being able to generate a valid solution in 185 milliseconds, while the NuSMV checker does terminate when analyzing `RTZ-01` gives us great confidence in the potential of our heuristic method of solving narrative games.

Solution for RTZ-01 (bonding)

`RTZ-01 (bonding)` adds the side-quest of getting the bonding plant as a requirement for winning the game. The bonding plant must also be *alive* in order for the player to win. This is done by first scaring away the vulture by throwing a rock at it (`take rock`, `throw rock`), and then moving closer to the signpost (`examine`). At this stage, the bonding plant can be retrieved, and it is crucial that the player digs the plant rather than cutting it (`dig bondingplant`). Then, the player must go back to the

	Optimal Solution	Heuristic Solution
01	take rock	s
02	throw rock	s
03	examine	e
04	dig bondingplant	cut vines
05	n	tie vines
06	s	n
07	s	n
08	e	n
09	cut vines	examine
10	tie vines	dig bondingplant
11	ride raft	n
12		drop rock
13		take rock
14		throw rock
15		examine
16		cut bondingplant
17		n
18		s
19		s
20		e
21		rideraft
Valid	—	no
Time(ms)	—	235

Table 7.13: Solutions for *RTZ-01 (bonding)*.

mountainpass (n), and execute the solution from RTZ-01. As we saw in Table 7.11 for CoD, we have a heuristic solution that is considerably longer than the optimal one, because the solver first looks to satisfy the condition of building the raft, and then backtracks to get the bonding plant. For that particular objective, we can see two flaws in the heuristic solution. First, executing the action `dig bondingplant` before scaring away the vulture will result in losing the game, and we will explain why it gets added before `take rock` and `throw rock`.

At that point in its execution, the solver is looking to satisfy the condition of the player having the bonding plant in his inventory. It first finds `cut bondingplant`, and considers the condition that the vulture must not be present, and therefore adds `take rock` and `throw rock` to satisfy the condition. By doing so, the vulture condition gets marked as being 'seen' (*see Section 5.1*), therefore getting ignored when later finding `dig bondingplant` as another action that satisfies the bonding plant condition.

Also, executing the `cut bondingplant` action prevents the player from winning because the bonding plant must be alive in order to win the game. Because it satisfies the condition of having the bonding plant in the player's inventory, it gets added to the solution.

Solution for RTZ-02

When the player starts the second chapter, she first needs to enter the Old Mill (`s`, `e`, `n`). Once inside, the drinking game starts, and to pass this test, the player must fool the npc by pretending to drink exactly three times (`toast`, `empty drink`, `drink`, ...). After those three drinks, the npc will be willing to hand over his key (`askAboutKey`). The player must then open a secret trap door, by removing a chock that blocks the water mill outside (`n`, `remove chock`). She can return inside the mill to find a hole in the floor which leads to a door (`n`, `climbdown`). Finally, the door can be unlocked and opened to win the game (`use booskey`, `n`).

As we can see in Table 7.14, the heuristic solution is only missing the steps from the drinking game, and is otherwise complete. Much like the problems encountered when attempting to find a solution for RTZ-01 (bonding), the omission is due to

	Optimal Solution	Heuristic Solution
01	s	s
02	e	e
03	n	n
04	toast	askAboutKey
05	empty drink	n
06	drink	remove chock
07	toast	n
08	empty drink	climbdown
09	drink	use boosKey
10	toast	n
11	empty drink	
12	drink	
13	askAboutKey	
14	n	
15	remove chock	
16	n	
17	climbdown	
18	use boosKey	
19	n	
Valid	—	no
Time(ms)	—	192

Table 7.14: Solutions for *RTZ-02*.

a condition being marked as 'seen'. This strategy allowed us to avoid running into conditional loops, while not having to maintain information about current game state. In this particular case, we are dealing with a group of statements that must repeated a certain number of times. Also, if the player was to drink too much, or drink without emptying his drink, she would have to restart the drinking game from scratch. This particular narrative trait is more complex than the other example narratives, but it nonetheless represents a good objective for our heuristic solver.

Heuristic Solver and The Count

We do not present any results for the example narrative *The Count*, because it uses features of PNFG language that are not supported by the heuristic solver. Namely, global actions, threads and timers play a crucial role in the PNFG implementation of the narrative. In its current form, the solver simply ignores these constructs. Extending the solver to support these two features is part of our future work. Table 7.1 shows the solution length is around 180 steps, much larger than any of our other examples narratives. In order for the solver to produce a solution for a large narrative like The Count, we will need to maintain information about the current state of the game as we build the solution.

Closing Remarks

While the solver in its current form is very simple, we can already appreciate the value of high level information contained in the PNFG source file in order to automatically solve narrative games. The solutions it can produce are not always valid, but we must remember they were all generated in less than 300 milliseconds, while the NuSMV solver was only able to produce a solution for `CoD`. The amount of time taken to build the solutions is a good indicator of the potential of this heuristic method, and also shows that the solver itself is still in a preliminary phase. While, heuristically building solutions that are optimal may prove infeasible for very complex narratives, this approach is certainly able to find large parts of the solution, as we have seen for `CoD`, `RTZ-01(bonding)`, and `RTZ-02`. In most of these cases, we see that the solution generated by the solver contains all the steps of the optimal solution. Hence, even if the solution presented is not valid it can still be of great use to the player, and the solver could be used as the basis for a very useful hint generator.

7.4 Metrics Results

Our analysis of game narratives has previously been limited to verification. We now extend our analysis to measuring different game metrics. Using the Metrics framework

presented in section 6.1, we first build a GameTree, and use this new representation to evaluate each metric. In the generation of our GameTrees, we only considered actions which affected the state of the game. To determine these actions, we build a set of properties that are modified for each action, and then ignore actions whose "write set" are empty. We will now present results for each metric, and also show a graphical representation of a GameTree in Figure 7.11, on page 92.

Edges to Nodes Ratio

Calculating the Edges to Nodes ratio allows us to get a general idea of the shape of the generated GameTree. In this representation, each Node corresponds to a unique Game State, and the edges symbolize an action execution. When there are more edges than actions, it means that there exists groups of edges that lead to the same node. Therefore, a higher ratio should correspond to a more complex narrative game, since many actions lead to visited game states.

	CoD	RTZ-01	RTZ-01 (bonding)	RTZ-02
Number of Nodes	33	414	1896	17148
Number of Edges	90	1110	5970	62220
Number of Win Nodes	4	12	12	96
Number of Win Edges	2	12	12	96
Number of Lose Nodes	2	114	444	6420
Number of Lose Edges	2	170	556	6756
Edge to Node ratio	2.73	2.68	3.15	3.63

Table 7.15: Nodes to Edge ratio metric results

From Table 7.15, we see that among the examples we considered, the larger narratives tend to have a bigger Edges to Nodes ratio, and more actions that are not needed in order to win the game. Also, it also indicates that our larger narratives are more complex. Including the number of win nodes in Table 7.15 also tells us if there are a lot of ways to win the games. We also note that in most of our example narratives,

the number of lose nodes is much larger than the number of win nodes, which also represents a measure of the level game difficulty.

Game Edges

By looking at the different types of edges we have in our generated GameTrees, we can refine the general view we had with the Edges to Nodes ratio. It is particularly interesting to find from Table 7.16 that around 20% of the edges in `RTZ-01` are unnecessary, which means that they play the same role as an already existing forward edge. It might be possible to reduce the size of our narratives by removing these unnecessary actions from the narrative, provided that their lack of usefulness can be generalized to all cases.

	CoD	RTZ-01	RTZ-01 (bonding)	RTZ-02
Forward Edges	32 (35.56%)	413 (37.21%)	1895 (31.74%)	17147 (27.56%)
Unnecessary Edges	4 (4.44%)	232 (20.90%)	1072 (17.96%)	8695 (13.97%)
Backward Edges	54 (60.0%)	465 (41.89%)	3003 (50.30%)	36378 (58.47%)

Table 7.16: Game Edges metric results

We can also use the data on different types of edges to analyze the shape of the GameTree. The data from Table 7.16 points to the same conclusions as Table 7.15, because the proportion of Forward edges decreases as narrative complexity increases. In the case of `RTZ-01`, we have to keep in mind it represents the beginning of the entire *Return to Zork* game, which might explain why the ratio of forward edges is higher. As it has been mentioned in [30], a *Classical Game Structure* suggests fewer choices for the player when she starts the game.

Convexity

The convexity metric also deals with the shape of the GameTree, but in a much more precise way. With this metric, our goal is to find whether or not the number of possible choices presented to the player starts out small, increases, and then decreases as the

player approaches the end of the game. Figure 7.5 plots the level in the GameTree versus the number of GameNodes for each example narrative.

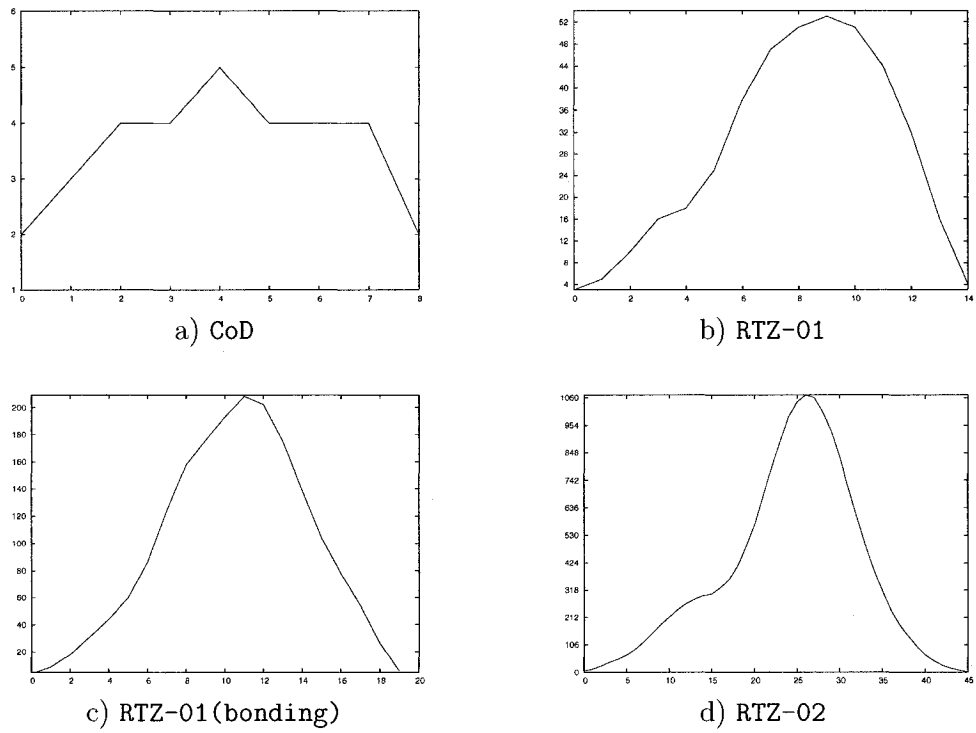


Figure 7.5: Narrative Convexity.

An important thing to note about this convexity measurement is that the end of the graph should not be interpreted as being necessarily the end of the game. The graphs represent the convexity of the entire narrative, a measure that considers all paths at the same time. In other words, the endpoint at level 14 of Graph 7.5b) means the longest path of actions that have a measurable effect on the game state has 14 actions.

The shapes of each graph in Figure 7.5 shows that the majority of GameNodes are located in the middle of the distribution. We can see a significant difference between the distribution of RTZ-01 and RTZ-02, where the latter's central concentration of

GameNodes is much more pronounced. The graphs indicate there are many middle-length paths, which can be interpreted as more choices for the player the middle of the game. This does not contradict the concept of convexity presented by Rabin in [30]. To get another point of view on the convexity of our example narratives, we will also analyze the convexity of individual game winning paths in the following section in order to get the number of choices the player has when actually playing the game and winning.

7.5 Results from Game Winning paths

In the following results, we only considered paths that are composed of Forward Edges. Table 7.17 shows some statistics about the Game Winning Paths we were able to measure using our metrics framework. The use of Game Winning Paths to measure game metrics allows us to get a closer look at the way the narrative is structured, without considering many sequences of actions all at the same time. As we can see, each narrative has Game Winning Paths, which verifies that they are in fact winnable.

	CoD	RTZ-01	RTZ-01 (bonding)	RTZ-02
Number of Paths	4	12	12	96
Number of Sets of Dependent Paths	2	1	1	1
Shortest Path Length	5	6	11	19
Longest Path Length	8	13	18	35

Table 7.17: Game Winning Paths

Dependency between Winning Paths

A set of dependent paths represents a group of paths that depend on each other. For example, the two paths in Figure 7.6 form a set of dependent paths because all the actions of path a) are found in path b). In that particular case, we say path

b) *depends* on path a). Furthermore, we can see from Table 7.17 that even though some narratives appear to offer many solutions to win the game, for three narratives, they all depend on the same minimal-length solution. We have also produced a graph representation of the dependencies between different paths, as shown in Figure 6.7 on Page 61.

```
Path a) w --> put --> e --> s --> read
Path b) s --> look --> n --> w --> put --> e --> s --> read
```

Figure 7.6: *Two dependent Game Winning Paths.*

In order to get a different perspective on Game Winning Paths, we have plotted the convexity of three Game Winning Paths for each narrative: the shortest, the median, and the longest paths (Figures 7.7, 7.8, 7.9, and 7.10, Pages 88 to 91). In the case of CoD, there was no such median length path. As we can see from these Figures, the solutions share the same general shape, but seem to be shifted laterally from one another. This is due to additional actions which affect the state of the game, but can be interpreted as not being necessary in order to win the game. Of course, this brings back the question of properly defining what is a necessary action, since a particular action might not be necessary to winning the game, but might appear essential for the player's comprehension of game objectives.

This very strong dependency we are observing between the different paths originates from the fact we are considering the full game state in our game tree, without making any distinction with respect to objects that actually matter in order to win the game. Therefore, modifying the value of one property can lead to the creation of an isomorphic subtree, where each GameState differs only by the value of this property. This effect is particularly easy to spot, when we look at the GameTree of *Return to Zork - Chapter 1* in Figure 7.11 on Page 92.

Convexity of Winning Paths

We can also look at the graphs in Figures 7.7, 7.8, 7.9, and 7.10 to analyze the convexity from a different perspective. In these graphs we can see the number of

choices the player will have when executing actions that lead to winning the game. This may be closer to the definition of Convexity as presented in [30].

From the different Figures, we cannot see any specific trend indicating that there are much less moves available to the player when the game starts, then more moves in the middle of the game, and less moves near the end of the game. We must remember that the notion Convexity is a *desired* property in games, and the graphs in Figures 7.7, 7.8, 7.9, and 7.10 indicate that our example narratives do not follow that trend. On the other hand, in every narrative, there is a point where very few actions are available, and this may very well indicate the presence of chapters. In the next section, we will look at another potential indicator of sections in narrative games.

7.6 Measuring the impact of adding a mandatory Game Quest

The division of games into different sections, or quests, is an area of narrative analysis that is of great interest to us, because we would like to use these divisions as a way of reducing the size of the narratives we analyze. In this section, we compare two versions of *Return to Zork - Chapter 1*. They differ only in that in one of them the player must have the living bonding plant in his inventory in order to win the game. Aside from that added condition, the games are identical. Table 7.18 presents the data we have collected on each version. In this section, we are comparing two versions of `RTZ-01(bonding)`, we are not using `RTZ-01`. We also compare the shortest game winning paths of each narrative in Table 7.19.

As we can see from Table 7.18, the only significant difference between the two version is the number of win nodes in the GameTree. This difference is due to the fact that the "optional quest" version has two actions that affect the game state (*take rock* and *drop rock*) but are not necessary to win the game. Also, it is important to note that all paths depend on the same one, which is another example of the what we saw in our metrics results for *Dependency between Winning Paths* in Section 7.4. The

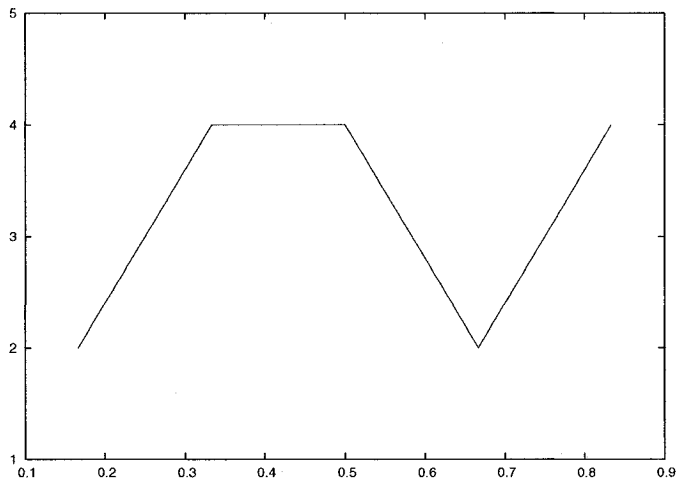
Bonding plant Quest	NFG			GameTree				
	nodes	trans.	BDD booleans	nodes	edges	Edge to Node ratio	win nodes	lose nodes
Optional	940	1464	175	1944	5970	3.07	60	444
Mandatory	944	1470	175	1896	5970	3.14	12	444

Table 7.18: Effects of adding a mandatory quest on NFG and GameTree properties

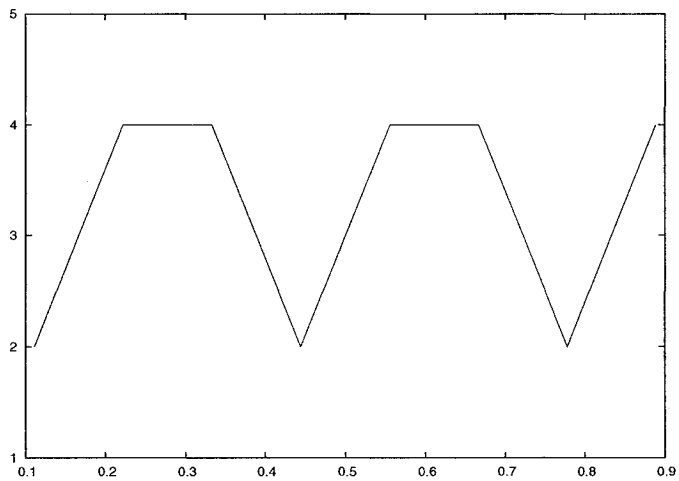
Bonding Plant Quest	Shortest Game Winning Path
Optional	s - s - e - cut vines - tie vines - rideraft
Mandatory	take rock - throw rock - examine - dig bondingplant - n - s - s - e - cut vines - tie vines - rideraft

Table 7.19: Effects of adding a mandatory quest on the shortest game winning path

more interesting results come when we look at the shortest game winning paths in Table 7.19. Here, we can easily identify the impact of adding an additional condition on winning the game, and can easily determine the actions that compose each quest . Quest 1 would be composed of the actions `take rock - throw rock - examine - dig bondingplant - n`, while Quest 2 would be made up of the actions `s - s - e - cut vines - tie vines - rideraft`. From these results, we see a possible strategy for identifying sections of game narratives would consist of removing conditions from the PNFG source file, and determining the impact on the game solution. On the other hand, this method requires the minimal solution to be efficient, and we could not use it on narratives that are larger than our current size limits. Still, these results will help orient our efforts in the future in order to come up with a solid quest identification mechanism.

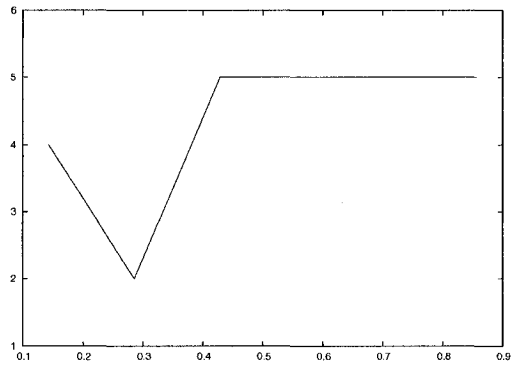


a) Shortest Path (5 moves)

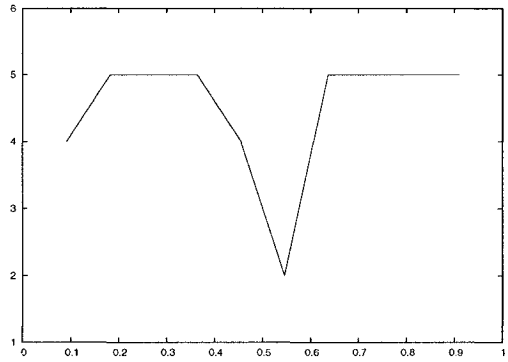


b) Longest Path (8 moves)

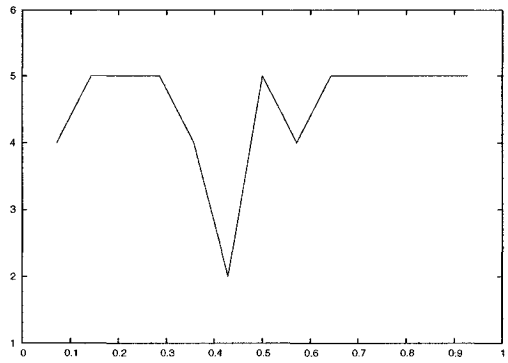
Figure 7.7: Two Game Winning Paths Normalized convexities for `CoD`.



a) Shortest Path (6 moves)

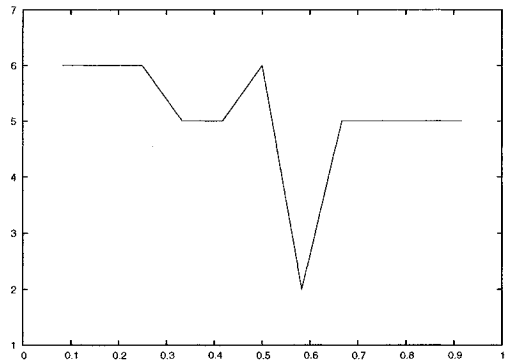


b) Median Path (10 moves)

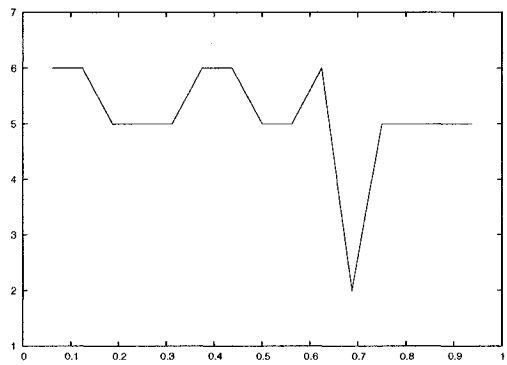


b) Longest Path (13 moves)

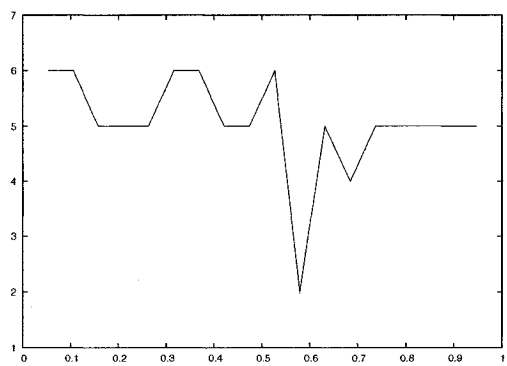
Figure 7.8: Three Game Winning Paths Normalized convexities for `RTZ01`.



a) Shortest Path (11 moves)

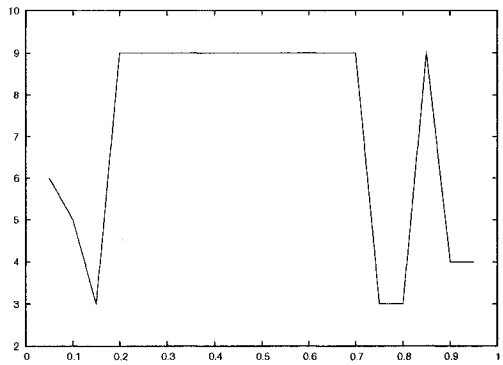


b) Median Path (15 moves)

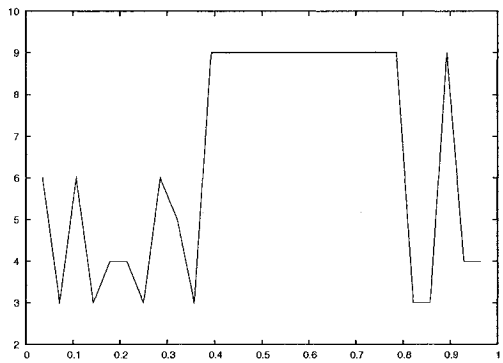


b) Longest Path (18 moves)

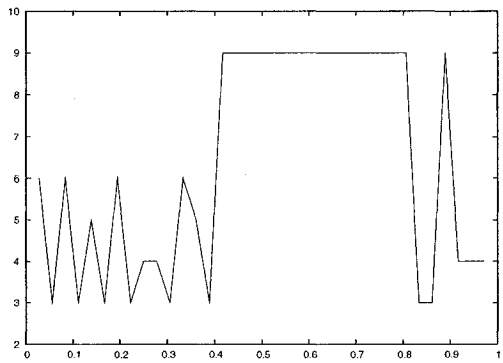
Figure 7.9: Three Game Winning Paths Normalized convexities for `RTZ01(bonding)`.



a) Shortest Path (19 moves)



b) Median Path (27 moves)



b) Longest Path (35 moves)

Figure 7.10: Three Game Winning Paths Normalized convexities for `RTZ02`.

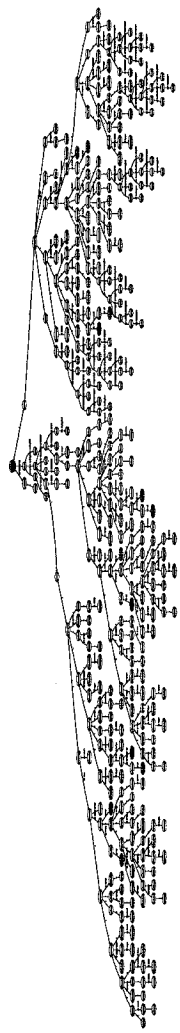


Figure 7.11: *GameTree for Return to Zork - Chapter 01.*

Chapter 8
Conclusions and Future Work

This work represents a very detailed description of the PNFG narrative game analysis framework. The PNFG language, its formal basis, and the accompanying code generation strategy are designed to allow for rigorous and algorithmic investigation of game narratives. Narrative problems are common in a variety of game genres, and by concentrating on the minimal, though narratively complex world of interactive fiction/adventure games we hope to produce practical solutions that are effective in many popular game genres.

Naively generated computer narratives have a surprisingly large state space. Even small games can result in structures that are far too large to search exhaustively, and the need for optimizations and other strategies to reduce the problem size is rather obvious. We have designed, implemented and tested several low-level optimizations that significantly reduce the output size. These have different effects, often dependent on the style of programming used to create the narrative, but collectively have quite a large impact. For larger narratives such as Return To Zork - Chapter 2 and The Count such optimizations may be necessary to even express the game, and are also an important, incremental step toward fully automatic verification.

We have also investigated the possibilities of using a very different strategy to deal with the large state spaces. Our Heuristic Solver has been designed to use high-level information found in our PNFG compiler intermediate representation to quickly derive the solution for narratives. While this exercise was a proof of concept, initial

results show great potential for this new approach at solving game narratives. In the future, we would like to extend the solver to make decision based on the game state, and to consider all the PNFG constructs, in order to produce better solutions for more complex games. A likely scenario would be to use the GameTree representation from the metrics framework to keep track of the game state.

Narrative game analysis can also benefit from high-level information, and we have created a new framework to measure different game metrics. By transforming our intermediate representation of the narrative in a GameTree, we can model the game on a representation that facilitates the task of deriving interesting properties about our example narratives. This new representation allowed us to obtain results for the larger narrative Return To Zork - Chapter 2, and we wish to continue extending the GameTree generation process in order to represent the full narrative, and analyze larger narratives.

The metrics framework presented opens the door to a new area of game narrative analysis by offering a representation that contains a very intuitive representation of the narrative. We certainly wish to pursue our research in this area and come up with new metrics for games. For example, we would like to keep track of the set of actions that are reachable by the player throughout the game and look for significant variations. We refer to this notion as continuity, and measuring this new metric, or looking at a game's overall complexity are but two examples of other metrics we wish to analyze using our framework.

Our current efforts suggest a large number of interesting and useful directions to explore. The overall complexity of large narratives shows the need for further optimizations and consideration of new ways to reduce the size of game narratives. For example, using a chapter decomposition for Return to Zork allows us to verify the first chapter; applying the same strategy to other narratives is highly desirable, and we are investigating automatic techniques that can help in this respect.

The detection of chapters is certainly a non-trivial problem, and will require some reflection on the notion of a game chapter itself, before something like the automatic detection of chapters can be investigated seriously. We have looked at the possible use of a convexity measurement on game winning paths as a way to identify chapters.

Also, the results we obtained from applying unplayable optimizations to our PNFG translation process give us motivation to pursue our efforts in order to further reduce the size of the NFGs we produce. We feel that identifying and removing useless components from the narratives shows great potential for future optimizations.

We have previously mentioned that the example narrative The Count represented a good long term goal for the size of narratives we would like to be able to analyze. As it stands now, the difference of size between our two biggest examples is relatively large, and we would like to analyze increasingly larger medium-size narratives, to eventually reach a game on the order of The Count. We believe representing new narratives in PNFG can yield improvements on all layers of the framework.

We believe the creation of narratives in PNFG code could benefit from an Integrated Development Environment. We have already explored this possibility internally, and a PNFG IDE offers the opportunity to assemble all the components of the PNFG framework and make them readily accessible to developers. Using the Eclipse Platform as a staring point is one of the possible scenarios we are considering for an eventual PNFG IDE.

Another interesting possibility for the PNFG framework would be to support concurrency in narratives, in order to allow multiple users to play simultaneously. The PNFG language already possesses some constructs that can support concurrency, and extending other components would allow us to analyze a different category of narratives.

Extending the PNFG language and the interpreter to support multimedia elements such as images and sound would make it very appealing to developers of 2D adventure games, and allow us to test the usability and overall usefulness of narrative analysis in the context of game development. This stage in the framework's evolution remains far off in the distance, but it certainly represents one of the many directions the framework may take in the future.

Bibliography

[1] The interactive fiction wiki. `http://www.ifwiki.org/index.php/Interactive_fiction`, 2005.

[2] D. Adams and S. Meretzky. The hitchhiker's guide to the galaxy. Infocom, 1984.

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

[4] M. Amster, D. Baggett, G. Ewing, P. Goetz, S. Harvey, F. Lee, R. Moser, P. Munn, J. Noble, J. Norrish, M. B. Sachs, M. Threepoint, R. Wallace, and J. Wallis. Plot in interactive works (was re: Attitudes to playing (longish)). discussion thread in rec.arts.int-fiction archives, October 1994.

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

[6] G. Berthelot. Checking properties of nets using transformations. *Lecture Notes in Computer Science: Advances in Petri Nets 1985*, 222:19–40, 1986.

[7] O. Bonnet-Torres and C. Tessier. From team plan to individual plans: a petri net-based approach. In *AAMAS '05: Proceedings of the fourth international joint conference on Autonomous agents and multiagent systems*, pages 797–804, New York, NY, USA, 2005. ACM Press.

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

[9] 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.

[10] R. Clarisó, E. Rodríguez-Carbonell, and J. Cortadella. Derivation of non-structural invariants of Petri nets using abstract interpretation. In *Proc. International Conference on Application and Theory of Petri Nets and Other Models of Concurrency (ICATPN'05)*, volume 3536 of *Lecture Notes in Computer Science*, pages 188–207. Springer-Verlag, June 2005.

[11] W. Crowther and D. Woods. The original adventure. The Software Toolworks, 1981.

[12] M. B. Dwyer and L. A. Clarke. A compact petri net representation and its implications for analysis. *IEEE Trans. Softw. Eng.*, 22(11):794–811, 1996.

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

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

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

[16] S. Griffiths, D. Glasser, J. Arnold, J. Barger, and D. A. Graves. [rec.arts.int-fiction] interactive fiction authorship FAQ. `http://www.plover.net/~textfire/raiffaq/FAQ.htm`, 2003.

[17] J. Juul. A clash between game and narrative: Interactive fiction. `http://www.jesperjuul.net`, Mar. 1998.

[18] S. Kim, S. Tugsinavisut, and P. Beerel. Reducing probabilistic timed petri nets for asynchronous architectural analysis. In *TAU '02: Proceedings of the 8th ACM/IEEE international workshop on Timing issues in the specification and synthesis of digital systems*, pages 140–147, New York, NY, USA, 2002. ACM Press.

[19] M. Leblanc. Feedback systems and the dramatic structure of competition. Game Developers Conference, 1999.

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

[21] C. A. Lindley and M. Eladhari. Causal normalisation: A methodology for coherent story logic design in computer role-playing games. In *CG'2002: International Conference on Computers and Games*, volume 2883 of *LNCS*, pages 292–307, July 2002.

[22] M. McNaughton, J.Schaeffer, D. Szafron, D. Parker, and J. Redford. Code generation for ai scripting in computer role-playing games. In *Challenges in Games Artificial Intelligence, AAAI Press*, pages 129–133, 2004.

[23] N. Montfort. Toward a theory of interactive fiction. In E. Short, editor, *IF Theory*. The Interactive Fiction Library, St. Charles, Illinois, Dec. 2003. Available online at `http://nickm.com/if/toward.html`.

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

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

[26] G. Nelson. Natural language, semantic aanalysis and interactive fiction. `http://www.inform-fiction.org`, Apr. 2005.

[27] J. Orwant. Eggg: Automated programming for game generation. *IBM System Journal / MIT Media Laboratory*, 39(3 & 4), 2000.

[28] B. Pell. METAGAME in symmetric chess-like games. Technical report, University of Cambridge, Computer Laboratory, 2003.

[29] 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, Montréal, Canada, Aug. 2005. Eurosis.

[30] S. Rabin. *Introduction to Game Development.* Charles River Media, June 2005.

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

[32] A. Rollings and E. Adams. *Andrew Rollings and Ernest Adams on Game Design.* New Riders Games, May 2003.

[33] K. Salen and E. Zimmerman. *Rules of Play : Game Design Fundamentals.* The MIT Pres, Oct. 2003.

[34] Ph. Schnoebelen and N. Sidorova. Bisimulation and the reduction of Petri nets. In M. Nielsen and D. Simpson, editors, *Proceedings of the 21st International Conference on Applications and Theory of Petri Nets (ICATPN 2000)*, volume 1825 of *Lecture Notes in Computer Science*, pages 409–423, Århus, Denmark, June 2000. Springer.

[35] J. H. Smith. The road not taken - the how's and why's of interactive fiction. `http://www.game-research.com`, 2000.

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

[37] S. van Egmond. [rec.games.int-fiction] FAQ. `http://www.faqs.org/faqs/games/interactive-fiction/`, 2003.

[38] 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.

[39] C. Volger. *The Writer's Journey: Mythic Structures for Writers.* Michael Wiese Productions, Oct. 1998.