Compiling and Optimizing A High Level Game Language

GR@M Technical Report No. 2006-1

Félix Martineau Clark Verbrugge
School of Computer Science, McGill University
Montréal, Québec, Canada, H3A 2A7
{fmarti10,clump}@cs.mcgill.ca

June, 2006

1 Introduction

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 recent work [23] we explored a design for a high level computer narrative language that allows for formal analysis of game narratives. *Programmable Narrative Flow Graphs* (PNFGs) provide a high level, user-friendly interface to a low level formalism, the *Narrative Flow Graph* (NFG) [29]. The 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. The framework presented in [23] also featured a solver module, using the NuSMV formal model checking software [8]. This software uses a brute force approach on a BDD boolean representation of a generated NFG to find the minimal solution to the game narrative, among other properties. Because of this brute force nature the size of the state-space of the narratives we wish to solve is always a key issue.

In this work we 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 analyse 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.

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.

In the next section, we discuss related work on narrative analysis and representation. In Section 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 in Section 4. Section 5 describes our different NFG optimizations, and Section 6 gives experimental results showing the impact of the optimizations on the output NFG. In Section 7 we conclude and describe future work in the area of computer narrative analysis.

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]. Among the most famous titles, we find examples such as Adventure (also known as ADVENT or Colossal Cave) [10], Zork [18], 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), [28] and `rec.arts.int-fiction` (or r.a.i-f) [14], the first focusing mainly on playing games, while the latter deals with the creation of new IF works.

In recent years, attempts have been made at analyzing IF from a theoretical approach, arguing that it is a legitimate literary art form [20]. When we look at what has been written on IF from this a 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" [15]. 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 [30], and have been proposed as narrative patterns for computer game design [25]. Within the realm of computer games, two different types of narrative can be observed: embedded, and emergent narrative [17], where the former is "pre-generated narrative content that exists prior to a player's interaction with with the game", and the latter refers to the narrative that "...arises from from the set of rules governing interaction with the game system" [26].

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 [19]. 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 [29]. 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 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 [29].

Interactive Fiction toolkits are very popular among the IF community, and the most popular authoring kits are probably Inform [22] and TADS [24, 12]. Inform is based on the tool used by the company Infocom, who published the most successful works of IF during the eighties, while TADS features an object-oriented language. The different authoring tools allow programmers to create complex story-lines, 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 [27]; 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 [9]. 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 [16]. Reusing existing Petri Net reductions and applying them to a domain-specific representation can be rewarding, as demonstrated in [11], where reductions suggested by Berthelot [6] are applied to Task-Interaction Graph-based Petri Nets.

The work we present in this paper directly extends our initial definition of *Programmable Narrative Flow Graphs* (PNFGs), a high level language that is easily mapped to a Petri Net model [23]. We review this material in the next section, and refer the reader to our earlier work for additional discussion and references to work on narrative problems, different representations of narratives, development frameworks for narratives, existing Interactive Fiction toolkits, and also how to map specific domains to a Petri net representation.

3 Narrative Representation as a PNFG

Interactive Fiction games 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 overcome 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) [29], a special class of 1-Safe Petri Nets that provides a simple syntax and operational semantics for describing narratives. In a nutshell, NFGs are, like Petri Nets composed of nodes (*places*) and *transitions*, in a bipartite, directed graph. 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. NFGs also include certain structural and simple correctness properties appropriate for representing computer game narratives; formal definitions and further details on NFGs can be found in [29]. Figure 1 shows a basic NFG representation for the trivial narrative used as a motivating example in [19].

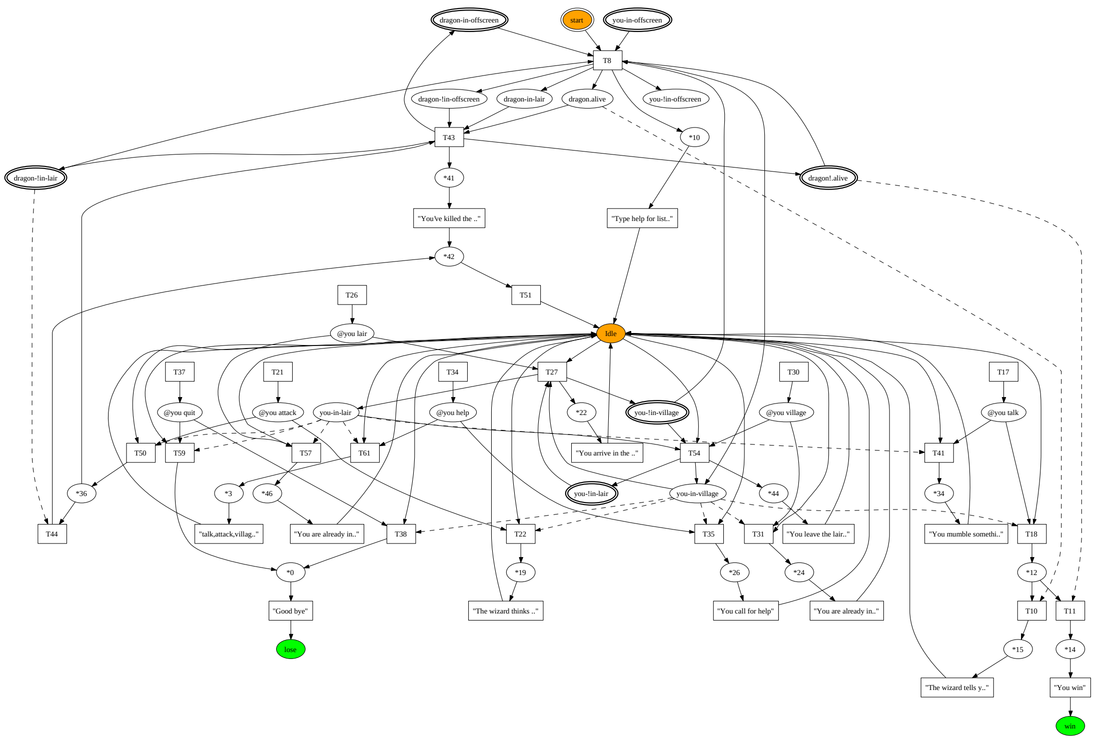

Figure 1: A NFG for the trivial narrative *The Wizard* [19].

3.1 PNFG Data and Declarations

Representing a narrative directly in NFG form can be very tedious; as can be seen from Figure 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 [23], 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 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 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 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.

3.2.1 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 4.

An example of both set and set variable declarations is shown in Figure 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: *Set declarations and set variables.*

3.3 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 4.2. Figure 4 shows a room declaration with two states being declared, `trapOpen` and `lit`.

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

Figure 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 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.

3.4 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 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 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.

4 PNFG Execution and Code Generation

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 [21]. The PNFG compiler generates a similar structure, and the general control flow of a PNFG game is shown schematically in Figure 6.

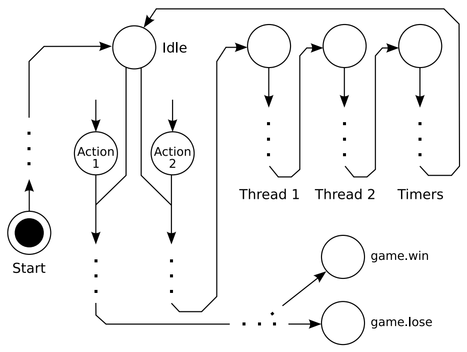

Figure 6: *The general NFG structure for a PNFG program.* The entry points for the main phases of execution are prologue, user commands, user threads, timers, and epilogue. *Taken from* [23]

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 3.4. 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 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 7: *A sequence of PNFG statements corresponding to a "kill npc" command.* Statements are referred to by number in the text.

4.1 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.

4.2 Basic PNFG Statements

The example of Figure 7 illustrates most of the core operations available in the PNFG language. This section described each of the basic operations, as well as how each of these actions is translated into some number of transitions in the underlying or output NFG.

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 8 shows a generalized example. The corresponding NFG pattern is a simple transition expressing the output string, as shown in Figure 9.

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* operations, 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

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

Figure 8: *Syntax for an Output statement.*

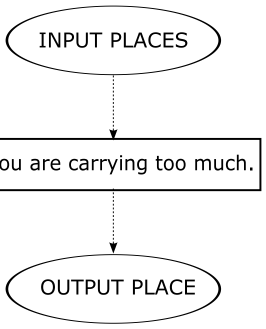

Figure 9: *NFG structure for the first output statement in Figure 8*

will stall, unable to activate the appropriate output context, as shown in part *a* of Figure 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 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 10.

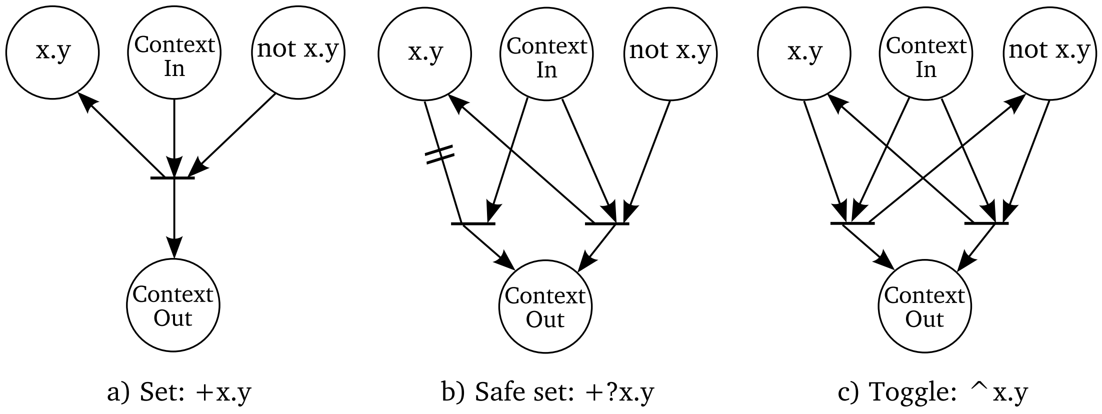

Figure 10: *NFG structure for the 3 main variations of the set statement.* Similar operations are defined for the symmetric unset operations, `-x.y` and `-?x,y`.

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 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.

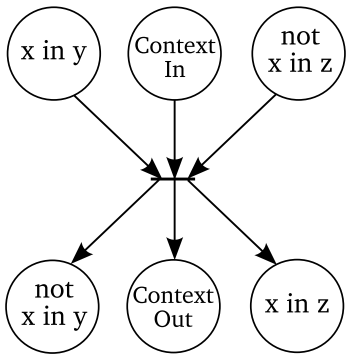

Figure 11: *NFG structure for statement,* "`move x from y to z`".

Counter operations. As discussed in Section 3.4 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 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.

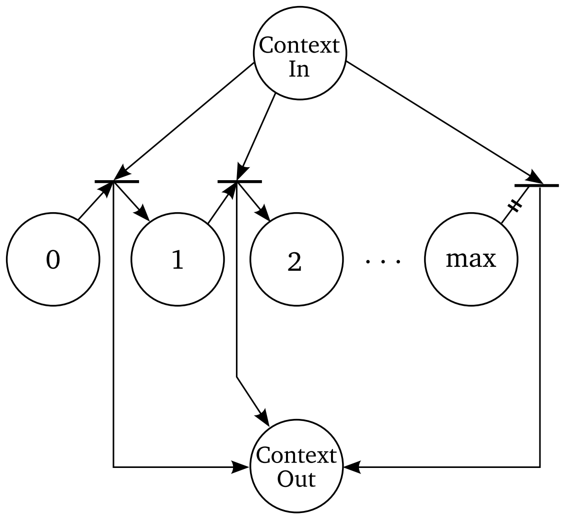

Figure 12: *Counters.* NFG structure for a counter increment.

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.

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 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.

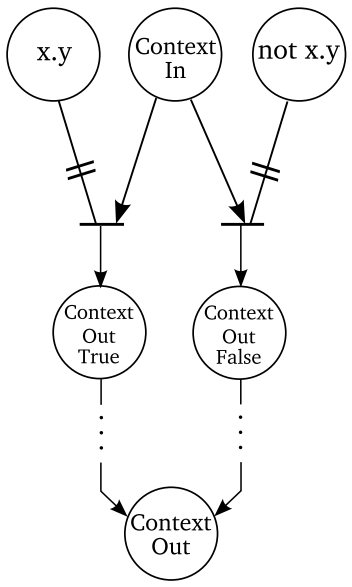

Figure 13: *NFG structure for a statement,* "`if (x.y) {...} else {...}`". Negative state tests ("`x!.y`") and positive/negative containment tests are structurally identical.

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 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 Section 5.

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

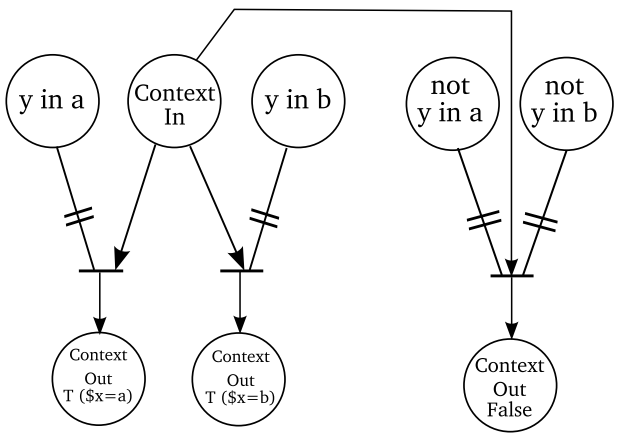

Figure 14: *Using variables.* NFG structure for a statement, "`if ($x contains y) ...`" where "`$x`" is an element of the set "{`a,b`}". Branch bodies and the following merge are not shown.

`(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 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 15.

```
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 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* [23]).

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`) {...}`".

4.3 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. Note that we will continue to refer to statements from Figure 7.

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.

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 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 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.

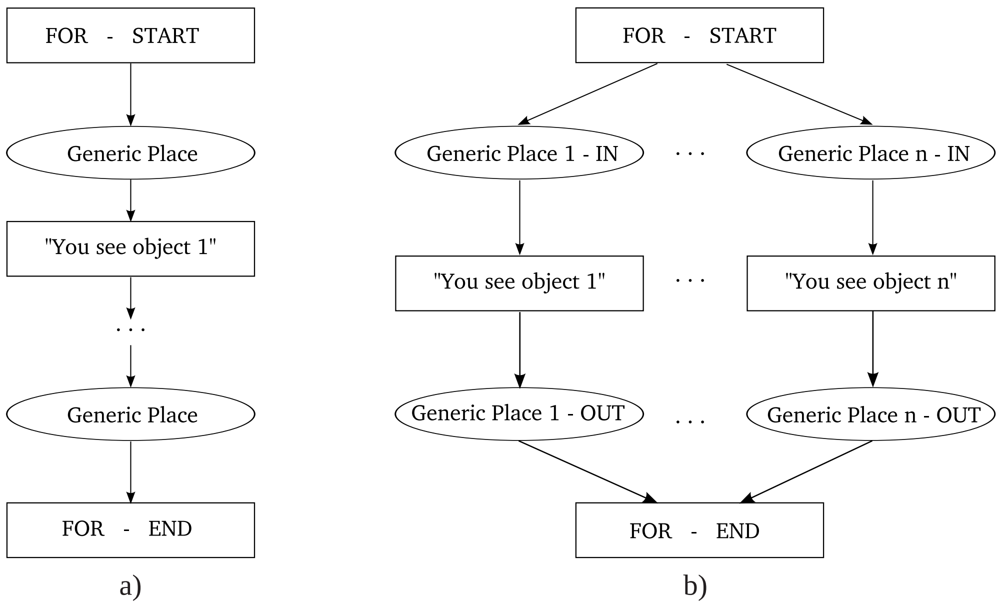

Figure 16: a) *Using for* execution of each case is sequential. b) *Using for-all* execution of each case is parallel.

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 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 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.

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 18 and 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 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 19: *A conditional thread declaration.* This thread only executes when the state `bomb.active` is true (*taken from* [23]).

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

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

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.

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. While effective and useful these are, unfortunately, 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.

5 Optimizations

In this section, 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 in earlier work [23].

Optimizations we will present fall into two categories, *safe* and *unplayable* optimizations. 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 an execution semantics that is difficult for human beings to actually play.

5.1 Safe Optimizations

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 22, 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).

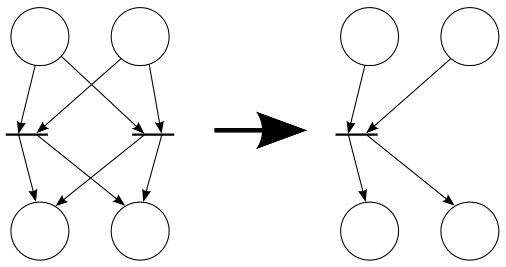

Figure 22: *Redundant Transition Removal.* The two transitions on the left have the same inputs and outputs, and so accomplish the same task. One of the transitions is thus sufficient, and the other can be safely removed.

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 Figure 23.

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.

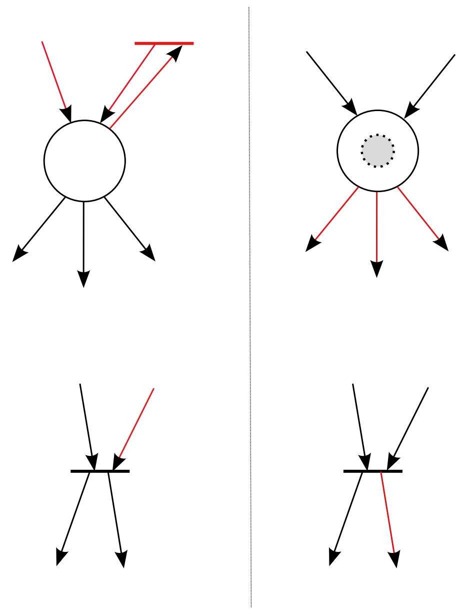

Figure 23: *Dead Code Removal.* Nodes are dead if all inputs are dead and no token exists, or if all outputs are dead. Transitions are dead if any input or output node is dead.

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 24. 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.

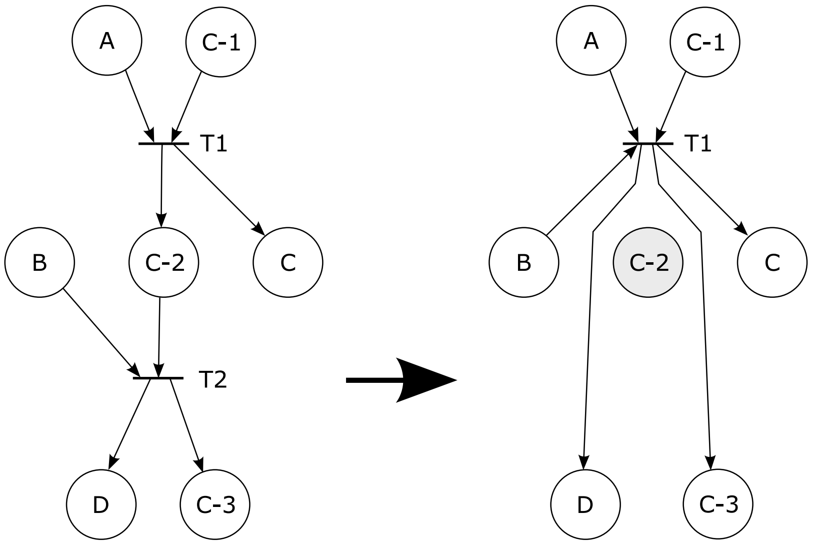

Figure 24: *Collapsing sequences of transitions. C-1, C-2,* and *C-3* are context nodes, while other nodes represent generic (non-context) input and output places.

Code Commoning. Much of the "syntactic sugar" present in the syntax of the PNFG language is designed to help reduce duplication in code generation as well as a 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 25.

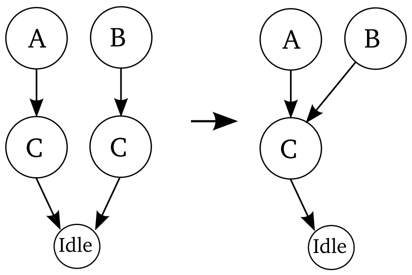

Figure 25: *Code Commoning. A, B,* and *C* represent arbitrary code blocks, with *A; C;* forming one action and *B; C* forming another action. In this case *C* can be easily commoned, with *A* and *B* redirected to a single instance of *C.*

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 26 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.

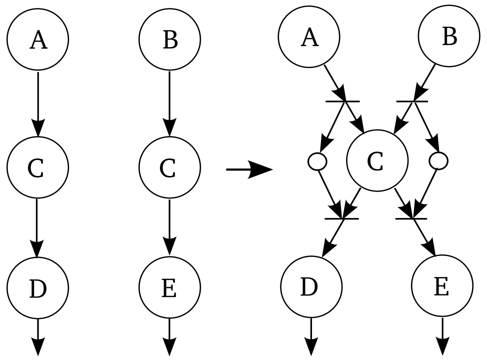

Figure 26: *Code Commoning.* Only a single instance of code block *C* is really required; however, extra nodes are necessary to ensure that upon exit from *C* control flows back to the appropriate code block, either *D* or *E*, depending on whether *C* was entered from *A* or *B* respectively.

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. 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 27.

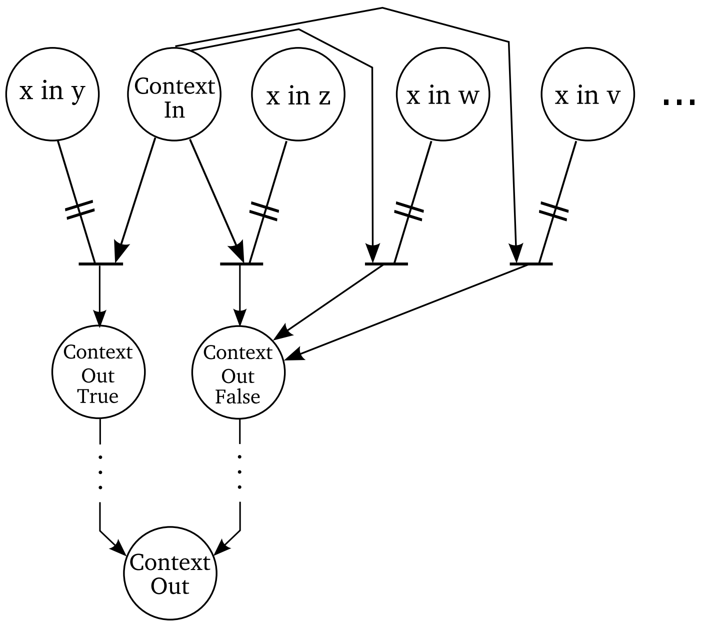

Figure 27: *No Not Nodes.* A conditional must test all possible locations to determine whether to follow the negative location branch.

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.

5.2 Unplayable Optimizations

No Default Actions. Default actions, discussed in Section 4.3, 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 show below in Section 6 show that this verification impact is not well-reflected in the size of the output NFG—mainly because default actions are trivially small.

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.

6 Experimental Results

Experimentation on the effect of optimizations is performed on four different narratives of increasing complexity and size. *Cloak of Darkness* [13] (CoD) represents a small narrative that serves as a tiny, but non-trivial sanity test for IF games. *Return to Zork* [5] (RTZ) is a larger, more recent game based on the popular Zork IF game—we consider the two initial chapters of RTZ as separate tests, the initial chapter is considered small, while the second chapter is of moderate size. A final, complete game representing a large narrative is a port of the well-known Scott Adams' IF game, *The Count* [3].

General statistics about each narrative are shown in Table 1. 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.

	Rooms	Objects	Nodes	Transitions	BDD Booleans
Cloak of Darkness	4	1	274	335	29
Return To Zork Part 1	10	19	965	1498	181
Return To Zork Part 2	21	36	1764	3806	241
The Count	22	29	12603	40762	686

Table 1: *Basic data on example narratives.* Number of rooms includes "you," the player. Nodes and transitions represent the total output NFG graph size, under all safe optimizations (except *no not nodes*).

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

6.2 Effects of Redundant Transition Removal

	COD	RTZ-01	RTZ02	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 2: Effects of redundant transition removal on NFG Size

Table 2 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 control structure.

6.3 Effects of Dead Code Removal

	COD	RTZ-01	RTZ02	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 3: Effects of dead code removal on NFG Size

Dead code removal has a disappointingly minimal impact. From Table 3 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.

6.4 Effects of Collapse Sequences of Transitions

	COD	RTZ-01	RTZ02	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 4: Effects of Collapse Sequences of Transitions on NFG Size

The sequence collapsing optimization has a fairly large impact on game narratives. Table 4 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.

6.5 Effects of Code Commoning

	COD	RTZ-01	RTZ02	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 5: Effects of code commoning on NFG Size

Table 5 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 5—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 the 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.

6.6 Effects of Commoning Functions

	COD	RTZ-01	RTZ02	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 6: Effects of commoning functions on NFG Size

As we can see in Table 6, 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.

6.7 Effects of Unoptimizing

	COD	RTZ-01	RTZ02	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: Effects of turning off all the above optimizations, except for basic code commoning, on NFG Size.

Table 7 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.

6.8 Effects of No Default Actions (Incremental)

	COD	RTZ-01	RTZ02	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 8: Effects of no default actions on NFG Size (*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 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.

6.9 Effects of No Output Statements (Incremental)

Even though the narratives would be unplayable by a player we can see from Table 9 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

	COD	RTZ-01	RTZ02	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 9: Effects of removing output statements on NFG Size (*incremental*)

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 Conclusions and Future Work

In this work we have presented our current, most complete language specification for game narratives. This 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.

There are large number of interesting and useful directions to explore suggested by our current efforts. 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.

A further interesting investigation is in the use of high level information. Our optimizations here are all based on low-level, NFG improvements, albeit to some degree informed by our higher level code generation design. Greater use of high-level information, such as in identifying necessary command dependencies, general goals and sub-tasks within games, and so on, have the potential to greatly improve verification design and cost. Our most active future work involves developing techniques to measure and evaluate game properties using the structure of the input PNFG itself rather than just low-level NFG forms. This is intended to help guide optimization and narrative understanding from a higher level perspective.

Other interesting, future approaches to game analysis build on the notion of "unplayable" optimizations:

not all of the narrative is necessary when investigating a particular analysis goal. “Useless” statements, commands, objects, rooms, can in principle be identified and excluded from the analysis work as a means of improving analysis efficiency, while preserving a mapping back to the original narrative design. This technique also has great promise, although it does require sophisticated and non-trivial efforts to identify such structures, at least without human intervention.

References

[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] 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.

[9] 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.

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

[11] 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.

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

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

[14] 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.

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

[16] 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.

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

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

[19] 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.

[20] 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`.

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

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

[23] 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.

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

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

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

[27] 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.

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

[29] 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.

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