Formal Verification of Computer Narratives

Christopher J.F. Pickett
School of Computer Science, McGill University
Montréal, Québec, Canada H3A 2A7
cpicke@sable.mcgill.ca

December 2nd, 2005

COMP-525 2005
1 Introduction
2 Interesting Temporal Properties
3 Representation
4 Verification
5 Conclusions and Future Work
Narrative verification?

- Modern computer games typically have some kind of narrative backbone.
- Unfortunately, plot holes, non-sequiturs, and dead ends abound in the narratives of even *massively commercial* games.
- It would be nice to prevent against these.

Who cares?

- Game designers that want certain narrative properties.
- Large numbers of “playtesters” that try to find bugs by hand.
- Players that want information about the current game state.
- Computer scientists, who like solving problems.
Motivation

What can we do?

- Identify some interesting temporal properties.
- Represent computer narratives as finite state machines.
- Generate models and check properties automatically.
> look
It’s pitch black, and you can’t see a thing!

> inventory
You are carrying:
  a lamp

> light lamp
The lamp flickers to life.

> look
You’re in a musty old cellar. Prakash is here.
Outline

1 Introduction

2 Interesting Temporal Properties

3 Representation

4 Verification

5 Conclusions and Future Work
Reachability: is there a path from $s$ to $t$?

- Binary yes/no: check $\mathcal{M}, s \models EFt$
- Actual path: counterexample produced by checking $\mathcal{M}, s \models \neg EFt$

Some interesting reachability questions:

- Can I lose? $\mathcal{M}, s_{\text{current}} \models EF\text{lose}$
- How do I win? $\mathcal{M}, s_{\text{current}} \models \neg EF\text{win}$
- How do I unlock this door? $\mathcal{M}, s_{\text{current}} \models \neg EF\text{door.unlocked}$
- Are there any zombie states: $\mathcal{M}, s_{\text{initial}} \models \neg AG(\text{EFwin} \lor EF\text{lose})$
Distance: shortest path from $s$ to $t$

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

But $\delta(s, t)$ is just length of reachability counterexample produced by $\mathcal{M}$, $s \models \neg EF t$, since BFS is used.
Separation (\(\gamma\))

Separation: longest acyclic path from \(s\) to \(t\)

\[
\gamma(s, t) = \max\{n|\exists s_0, \ldots, s_n. s = s_0, t = s_n, (s_i \rightarrow s_{i+1} \land (s_i \neq s_j \text{ for } 0 \leq j < i) \land (s_i \neq s_k \text{ for } i < k \leq n)) \text{ for } 0 \leq i < n\}
\]

Not obvious how \(\gamma(s, t)\) can be efficiently checked in general.
Reoccurrence radius: longest path from $s$ to any other reachable state $t$

$$\rho(s) = \max\{\gamma(s, t) | t \in S \land M, s \models EFt\}$$

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

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

Intuition: find the shortest path length such that extending it any further will always create a cycle; this is the longest acyclic path length.

This is a propositional formula, and can be checked by SAT.
A game is pointless: $\mathcal{M}, s_{\text{current}} \models \neg \text{EF} \text{win}$
A game is $p$-pointless: it takes a maximum of $p$ steps from pointlessness to actually losing.

How to calculate $p$?

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

If $\mathcal{M}, s_{\text{pointless}} \models \neg \text{AG(EFlose)}$, then $p = \infty$.
Otherwise, $p = \rho(s_{\text{pointless}})$.

Why? lose is a sink node, and reachable on all paths, so any putative acyclic longest path not including lose can be extended to include lose.
1 Introduction
2 Interesting Temporal Properties
3 Representation
4 Verification
5 Conclusions and Future Work
Represent computer narratives using Petri Nets.

- Compact encoding of large state space.

Consider “1-safe” Petri Nets:

- \( S \), a set of places
- \( T \), a set of transitions
- \( F \subseteq (S \times T) \cup (T \times S) \), a flow relation
  - No arc connects two places or two transitions
- \( M_0 : S \rightarrow \{0, 1\} \), an initial marking, or distribution of tokens over \( s \in S \).
  - A 1-safe Petri Net has only 0 or 1 tokens in any place.
  - Markings are equivalent to states.

A transition is enabled if all source places contain tokens.

An enabled transition can fire, removing tokens from all source places and filling all destination places.
Petri Nets
Narrative Flow Graphs

We extend 1-safe Petri Nets to *Narrative Flow Graphs* (NFGs). Need some additional elements:

- $M_{\text{win}}$, a sink marking that signifies winning
- $M_{\text{lose}}$, a sink marking that signifies losing
- $L$, a set of labels
- $I : T \rightarrow L$, a mapping of transitions to input labels
- $O : T \rightarrow L$, a mapping of transitions to output labels
- $T_{\text{actions}} : t \in T . \ I(t) \in L$, action transitions
- $T_{\text{internal}} : t \in T . \ \neg(I(t) \in L)$, internal transitions

The narrative flows from $M_0$ to $M_{\text{win}}$ or $M_{\text{lose}}$ via some series of $t_a \in T_{\text{actions}}$ and $t_i \in T_{\text{internal}}$.

Internal transitions take priority, and firing is sequential.
Narrative Flow Graphs

- `lamp_unlit`
- `idle`
- `carrying_lamp`

"light lamp" (input)

"The lamp flickers to life." (output)

- `lamp_lit`
- `burn_fuel`
- `fuel_10`

idle

fuel_9
Narrative Flow Graphs

- lamp_unlit
- idle
- carrying_lamp

"light lamp" (input)

"The lamp flickers to life." (output)

- lamp_lit
- burn_fuel
- fuel_10

- idle
- fuel_9
Narrative Flow Graphs

lump_unlit -> idle -> carrying_lamp

"light lamp" (input) -> "The lamp flickers to life." (output)

lamp_lit -> burn_fuel -> fuel_10

idle -> fuel_9
initialize NFG
while (!won && !lost)
    while (some $t_i \in T_{\text{internal}}$ enabled)
        fire $t_i$
    wait for user input
switch (input)
    case "query win":
        $\mathcal{M}, s_{\text{current}} \models \neg \text{EF} \text{win}$
    case "query lose":
        $\mathcal{M}, s_{\text{current}} \models \neg \text{EF} \text{lose}$
    case "query moves":
        print $l(t_a)$ for each enabled $t_a \in T_{\text{actions}}$
    case ($l(t_a)$ for some enabled $t_a \in T_{\text{actions}}$): fire $t_a$
    default:
        "Sorry, try something else."
Writing NFGs directly is painful and error-prone.

The *Programmable Narrative Flow Graph* (PNFG) language and compiler attempts to alleviate this concern.

Programmer writes narrative in imperative language.

Compiler generates NFG for interpretation and verification.
object lamp {
  state { lit }
  counter { fuel 0 10 }
}

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

...
For now, focus on simple reachability of win and lose.

- SPEC !EF win = 1

Translation from Petri Net (NFG) model is straightforward

High sensitivity to variable ordering:
- Might quickly run out of memory
- Might spend hours and not make any progress
- Might produce a solution in a few minutes

Player can query paths to win and lose at any point.
Exploiting PNFG Structure

Efficient encodings of Petri Nets in NuSMV is critical.

- Exploit structure of code generated by PNFG compiler
- Instead of modelling individual PN places, model tokens instead.
  - Split net into disjoint $S_{mutex}$, each with only 1 token.
  - Modelling by places: $|S_{mutex}|$ booleans per token.
  - Modelling by tokens: $\lceil \log_2 |S_{mutex}| \rceil$ booleans per token.
Three main applications:

1. Objects can only be in 1 room at a time. Cost per object is $\lceil \log_2 |R| \rceil$, where $R$ is the set of rooms.

2. Scalars can only assume 1 value at a time. Cost per scalar is $\lceil \log_2 |C| \rceil$, where $C$ is the set of unary counter values.

3. Program control can only be in 1 state at a time. Cost of $pc$ is $\lceil \log_2 |P| \rceil$, where $P$ is the number of nodes in CFG.
## Experimental Results

<table>
<thead>
<tr>
<th>Narrative Title</th>
<th>Cloak of Darkness</th>
<th>Cloak of Darkness</th>
<th>Return to Zork Ch. 1</th>
<th>Return to Zork Ch. 2</th>
<th>The Count</th>
</tr>
</thead>
<tbody>
<tr>
<td>Source</td>
<td>.nfg</td>
<td>.pnfg</td>
<td>.pnfg</td>
<td>.pnfg</td>
<td>.pnfg</td>
</tr>
<tr>
<td>Rooms</td>
<td>3</td>
<td>4</td>
<td>10</td>
<td>21</td>
<td>22</td>
</tr>
<tr>
<td>Objects</td>
<td>3</td>
<td>1</td>
<td>19</td>
<td>36</td>
<td>29</td>
</tr>
<tr>
<td>Places</td>
<td>69</td>
<td>303</td>
<td>1275</td>
<td>1876</td>
<td>15378</td>
</tr>
<tr>
<td>Transitions</td>
<td>167</td>
<td>462</td>
<td>3341</td>
<td>8030</td>
<td>82371</td>
</tr>
<tr>
<td>Booleans</td>
<td>69</td>
<td>27</td>
<td>98</td>
<td>117</td>
<td>212</td>
</tr>
<tr>
<td>Verifiable</td>
<td>yes</td>
<td>yes</td>
<td>no</td>
<td>no</td>
<td>no</td>
</tr>
<tr>
<td>Steps to Win</td>
<td>6</td>
<td>6</td>
<td>6</td>
<td>22</td>
<td>180</td>
</tr>
</tbody>
</table>
1 Introduction
2 Interesting Temporal Properties
3 Representation
4 Verification
5 Conclusions and Future Work
There are several interesting temporal properties in computer narratives.

Petri Nets can represent large systems efficiently, and are suitable for modelling these narratives.

Software verification is hard; exploiting structure in a high-level language can help.
Future Work

- Improve NuSMV model generation, see if we can compute reachability for larger problem sizes.
- Extend PNFG compiler to accept LTL and CTL specifications.
- Check other properties besides reachability, such as degree of pointlessness.