The “Board Game Model”
A High Level Overview

In some ways, the word “model” describes only half of what I’ve been working on this semester. I have worked on two things: The model and an interpreter/engine.

The Model

The model can be thought of as a “game description language.” It is a way to think about games and express their rules. I tried to make the model very abstract so that it could accommodate a wide variety of games.

Games can be described in a number of ways. For instance, Tic-Tac-Toe can be described as follows:

- Method 1: This game takes place on a 3-by-3 grid, where players take turns placing pieces of their color into an empty cell. The first player to get three in a row wins.

- Method 2: This game has 19 buckets (9 red, 9 yellow, and 1 green). One player owns 5 yellow buckets and the green bucket. The other owns just 4 yellow buckets. To make a move, the player must own the green bucket. A legal move consists of giving the green bucket to the other player along with putting a yellow bucket the player owns into an empty red bucket. A player wins instantly if buckets they own are in red buckets (1,2, and 3), (4, 5, and 6), (7, 8, and 9), (1, 4, and 7), (2, 5, and 8), (3, 6, and 9), (1, 5, and 9), or (3, 5, and 7).

As you can see, method 1 is much more concise. It has more built-in constructs, such as taking turns, a grid, and three-in-a-row. Both methods have a concept of color. Method 2 allows potentially greater flexibility in the games described. My game description language is similar to that of method 2.

The Interpreter

The interpreter (or “engine”) is a piece of software. It reads in text files formatted according to the model’s requirements and then allows people to play the game as described in the files. The interpreter must parse the files, converting the text into internal objects the rest of the code can use. Then it must allow users to interface with the game state it stores, updating the game state as they take actions, along with regulating when these actions are or are not allowed.

For any given model, a multitude of different engines could be coded with differences both in how they keep track of game state and in the interface they offer.

Some Comments

While the model I created doesn't have constructs like a grid, it may be nice to add them eventually. Every addition to the model requires additions to the engine. If the model is going to be used for game generation, higher-level constructs could be useful for creating things more quickly, but adds redundancy and removes some of the elegance a lower-level game description level offers. These thoughts are of a speculative nature at this point as I have not yet actually worked on automatic game-generation with the model.

An Example

Below is a game played with 15 pebbles. Players take turns removing 1-3 pebbles. If one player takes the last pebble(s), the other player loses.

Actions.txt

```
Action TakeOne Pebble p1 Container c
      require in p1 c
      effect remove p1
      effect player player.left

Action TakeTwo Pebble p1 Pebble p2 Container c
      require in p1 c
      require in p2 c
      effect remove p1
      effect remove p2
      effect player player.left

Action TakeThree Pebble p1 Pebble p2 Pebble p3 Container c
      require in p1 c
      require in p2 c
      require in p3 c
      effect remove p1
      effect remove p2
      effect remove p3
      effect player player.left
```

Instances.txt

```
Container c
Pebble
      in c
Pebble
```

```
      in c
Pebble
      in c
Pebble
      in c
Pebble
      in c
Pebble
      in c
Pebble
      in c
Pebble
      in c
Pebble
      in c
Pebble
      in c
Pebble
      in c
Pebble
      in c
Pebble
      in c
Pebble
      in c
Pebble
      in c
Pebble
      in c
```

WinLose.txt

```
LoseCondition Container c
      require contains Pebble c < 1
```