Wordgrind: a Logic Programming Language for Creating Quality-Based Narrative

Jan Wanot
Royal Holloway University of London
Egham, Surrey, United Kingdom
jan.wanot.2017@rhul.ac.uk

Abstract

We present a tool for writing interactive fiction based on the *storylet model* (Kreminski and Wardrip-Fruin 2018), which allows for crafting an interactive narrative out of many discrete "chunks", the order and presence of which is based on player choices and the structure of game logic. To facilitate creation of the latter, our tool is equipped with an integrated declarative logic programming capability, similar to a logic programming language such as Prolog. The created narrative systems can be exported to a browser-readable format, which allows for easy distribution and future preservation.

Introduction

Quality-based narrative, also known as the *storylet model* (Kreminski and Wardrip-Fruin 2018) allows for structuring narrative content in a way that enables the content to be selected and ordered in a dynamic way depending on the game state and player choices. Our tool incorporates a full logic programming capability that allows for creation, manipulation and selection of content in an intuitive and efficient way, along with more advanced state tracking then offered by most other QBN interactive fiction authoring systems.

The functionality of Wordgrind as a tool can be separated into three discrete aspects bound together by the underlying logic programming engine:

- A textual templating system, allowing parts of longer text chunks to be substituted, reordered or omitted based on programmatic conditions, as well as allowing for creating multiple variants of a given text chunk to be created based on logical terms.
- A content selection system, allowing both for presenting various available context-dependent player choices, as well as more automated "actions", which are executed automatically as soon as their preconditions are met.
- A dynamic database of current game state, presented in a form of structured logic programming terms, along with convenience features for simpler state management.

All of these aspects are enabled by the logic programming functionality, which allows for a declarative style of writing game logic while being completely flexible. Despite requiring little to no conventional programming experience, Wordgrind can be used for creating interactive systems of great complexity.

Example

Suppose we want to create an interactive system with a world model evocative of traditional parser-based interactive fiction: a series of rooms that the player can navigate, containing items that can be pick up and placed in the inventory. It should be noted that while unlike more specialized IF creation tools such as Inform 7 (2006), Wordgrind itself contains no built-in preference for that model of interaction compared to any other alternative (i.e. one could just as easily use Wordgrind to create a dialogue system, a social simulation system, or a resource management game), such a system can still be created very simply and without much boilerplate code by utilizing some of the logic programming features of the language in conjunction with the storylet model.

```
  Predicates:
  - <kitchen> is a room
  - <living room> is a room
  - <bathroom> is a room
  - <bedroom> is a room

  - <kitchen> is connected to <living room>
  - <living room> is connected to <bathroom>
  - <living room> is connected to <bedroom>

  - ?A and ?B can be walked between:
      or:
        - ?A is connected to ?B
        - ?B is connected to ?A

Unique facts:
  - Player is in ?_

Initial state:
  condition:
     - Player is in <bedroom>
     - <soap> is in <bathroom>
     - <TV remote> is in <living room>
     - <knife> is in <kitchen>
  displays:
    You wake up in a cold sweat. You had nightmares
    all night, and now you have a horrible headache.


default:
  Choices:
    Walk from ?A to ?B:
      available when:
        - Player is in ?A
      such that:
        and:
          - ?B is a room
          - ?A is a room
          - ?A and ?B can be walked between
      displays:
        You walk from ?A to ?B.
      causes:
        - Player is in ?B

    Pick up ?Item:
      available when:
        - Player is in ?Place
        - ?Item is in ?Place
      displays:
        You have picked up ?Item.
      causes:
        - Player has ?Item
        - removes: ?Item is in ?Place

    Drop ?Item:
      available when:
        - Player is in ?Place
        - Player has ?Item
      displays:
        You have dropped ?Item.
      causes:
        - ?Item is in ?Place
        - removes: Player has ?Item
```

Our example scenario consists of four rooms (kitchen, living room, bathroom and a bedroom) which are declared using the “{} is a room“ predicate. We establish connections between rooms using the “{} is connected to {}“ predicate, but for actual choices given to the player we will be using the “{} and {} can be walked between“ predicate, to allow player to walk in both directions (we assume there are no one-way passages in this system for convenience).

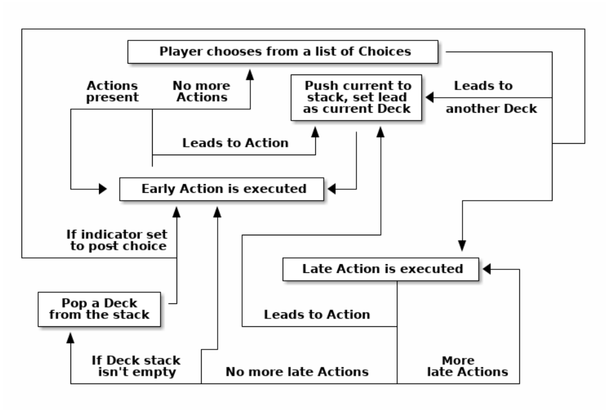
Since the locations of the player and the items are dynamic, we use database terms rather than a predicate to represent them. The initial condition of the system, along with the starting text, is placed in the initial state section. In addition, we declare the player location to be a unique fact, meaning that the old value in the database will be deleted automatically when the new one is inserted, without the need to declare so explicitly.

Our scenario consists of a single default deck with three choices: one for walking between the rooms, and two for interacting with items. Note that the “Walk from {} to {}“ choice makes use of the logical predicates we have defined before, whereas the other two choices work only by matching the terms available in the dynamic database. Simply augmenting the precondition/effects syntax with variables is sufficient to achieve non-determinism and allow for a single element to serve as a template for an entire class of interactions.

Here is an example interaction with the system:

```
You wake up in a cold sweat. You had nightmares all
night, and now you have a horrible headache.

> Walk from bedroom to living room

You walk from bedroom to living room.

> Walk from living room to kitchen
> Walk from living room to bathroom
> Walk from living room to bedroom
> Pick up TV remote

You have picked up TV remote.

> Walk from living room to kitchen
> Walk from living room to bathroom
> Walk from living room to bedroom
> Drop TV remote

You walk from living room to bathroom.

> Walk from bathroom to living room
> Pick up soap
> Drop TV remote
```

Content selection overview



Player chooses from a list of Choices
Actions present
No more Actions
Push current to stack, set lead as current Deck
Leads to another Deck
Leads to Action
Early Action is executed
If indicator set to post choice
Late Action is executed
Pop a Deck from the stack
Leads to Action
If Deck stack isn't empty
No more late Actions
More late Actions

Since content selection is one of the most important aspects of every interactive storytelling system, we implement a relatively robust solution that allows for authoring both the pattern of interaction with the player and more the automated simulation-like systems (such as NPCs or dynamic state of the world) using the same storylet-based content system.

In Wordgrind, content is divided into pieces of content called *Elements*. Elements can be either *Actions* or *Choices*, and are grouped into *Decks*.

Each Element contains *preconditions* which must be met in order for the element to be selected. These preconditions take a form of a list of (partially of fully instantiated) logic terms, which are matched with the dynamic database. If a given term unifies with a term in the database, this precondition is considered to be fulfilled. Optionally, a precondition can be negative, in the sense that it is only considered to be fulfilled if no term in the database unifies with it. In addition to these preconditions, which are dependent on the dynamic database, Elements can optionally also have direct logic statements embedded inside of them, which must also be satisfied in order for the Element to be selected.

Elements also contain *effects* which are the changes to the game state that occur as a result of a given Element being executed. Like preconditions, the effects also take the form of a list of logic terms. Each effect represents a term to either be added or removed from the database. Certain kinds of terms can be marked as *Unique* in a specific section of the file, meaning that only one term with a specific top-level functor name can be held in the database at one time. This means that when a new such term is added, the previous one is automatically removed. This is useful for types of state which should not contain duplicate values, such as player location, or a plot-tracking stat value.

The execution of the selected Elements depends on their type. Actions are executed automatically as soon as they are selected, in order of their priority level. Choices are presented to the player, and only the one that is chosen is executed. In addition, Actions can be further divided into those that are selected and executed before and after the player makes a choice.

All Elements are grouped into Decks, with those not explicitly marked with one being grouped into the default Deck. One Deck is active at the time, with content selection always happening within the Elements from the currently active Deck. Each Element can specify a Deck that it "leads" to, and once that Element is executed, that Deck becomes active. This is intended as an easier way of grouping together content, as well as a method of optimizing selection time in more sophisticated interactive systems.

In the current version of Wordgrind, Decks operate on a stack-based system - when a new Deck becomes active, it is pushed onto the stack on top of the old one, and when an Element that does not lead to a new Deck is executed the active Deck is popped from the stack and the previous Deck becomes active. This is intended to make Decks more reusable and enable easier creation of nested menu structures. However, as of writing, the details of this particular system are still being refined.

Logic programming aspects

As previously mentioned, Wordgrind owes much of its expressivity to the logic programming language that extends throughout the whole tool. Logic variables and terms can be substituted into any piece of textual content shown to the player, which allows for simple text templating being integrated directly into the language.

Along with each Action and Choice Elements being able to specify its own logic, Wordgrind file includes a section for predicates, allowing for code reusability. All predicates in Wordgrind, along with all the structured data terms used in the predicates and the dynamic database, are in the form of natural language sentences with variables and data terms "embedded" inside of them. This solution, while only deviating from the standard logic programming convention on the syntax level, provides a simpler foundation for the templating system, as well as enables a more "literate programming" style suited to non-technical content creators.

Both the logic terms embedded in the Action and Choice Elements as well as the standalone predicates are compiled directly to high-level JavaScript code, which is intended to be human-readable. Using a method inspired by the Mercury logic programming language, each predicate is compiled into a separate JavaScript function, where the non-deterministic aspect is handled through continuation passing (Henderson and Somogyi 2002). This one-predicate-one-function correspondence allows for easier human comprehension of the generated code, which is valuable when debugging or embedding the generated code within external systems and frameworks.

The non-deterministic capability is an important feature of the system in itself. Since each predicate can have a number of valid "results" (bindings of variables which are considered true), it is very easy to create a number of different available Choices or Actions working from the same template, by introducing inside of it a variable with several possible valid bindings. For instance, when implementing a system where a player character can move freely between rooms, instead of manually writing a Choice for each connection, one can create a single Choice for walking between two locations designated with variables, and then create logic that ensures that this option is only available if these two variables represent two directly connected rooms. This feature is similar to the "parameterized storylet" concept introduced by (Kreminski and Wardrip-Fruin 2018).

Related work

While both interactive fiction tools making usage of logic programming and those that allow for QBN exist, to our knowledge those two concepts had not previously been used together in one authoring tool. Our design is partially influenced by the Dialog interactive fiction language (2018), especially with regards to the usage of natural language with embedded logic terms and variables as a way of structuring data (in contrast to the more traditional functor-like notation used by the Prolog family of logic programming languages).

Despite the fact that it is a rule specification language rather than an interactive narrative tool, Ceptre (Martens 2021) offers an approach to content selection similar to ours, with *stages* and *rules* being the equivalent of Decks and Elements in Wordgrind. In place of preconditions and effects commonly seen in QBN systems, Ceptre instead uses linear logic to describe changes to the game state happening due to the application of its rules.

The exclusion logic used within the Praxis logic language, one of the components of the Versu storytelling system (Evans and Short 2014), is used within that tool to solve a similar problem as the Unique facts feature in Wordgrind, namely the necessity of repeatedly removing and adding a persistent stat property every time it needs to be changed. Exclusion logic however allows for more sophisticated control, as well as the ability to easily express tree data structures.

While the concept of QBN has existed for over a decade, it wasn't until recent years that a number of tools for creating

QBN fiction have emerged, including SimpleQBN (2020), StoryletManager (2021) and Tiny QBN (2019). A popular data format for an existing interactive fiction tool Twine, Harlowe, had recently added support for a storylet-based selection mode as well (2021). Most of these tools so far only support simple methods of content selection based on integer-based state values and preconditions, or otherwise lack the ability to use structured data for game state. The main exception is SimpleQBN, which uses the MongoDB database query language that permits more advanced forms of state and condition tracking. Of the mentioned systems, only StoryletManager has support for parameterized storylets, a feature which emerges in Wordgrind naturally as a direct result of augmenting content with logic variables and terms.

Conclusions & Future Work

In this paper we introduced Wordgrind, a tool for authoring storylet-based interactive fiction based on logic programming, and demonstrated both its content selection mechanism and how it uses logic to augment its functionality.

While our tool in its current implementation has sufficient functionality to demonstrate the underlying concepts, more work will be needed in the future to ensure both technical stability and user convenience sufficient for a public release. In particular, the question of syntax had only been mentioned in this paper in broad strokes, as it is one of the aspects of the tool that will require heavy user feedback and iteration. The current implementation uses a simple YAML file as a document format, however it is most likely that a custom parser will be used at some point in the future.

The current logic programming system, while complete enough for most purposes, still lacks some features that we hope to make it into the final version. In particular, logical negation and if/else statements are still to be implemented, as well as syntax and library functions for dealing with list data structures. In the farther future, we also hope to implement a mode-based compilation system, similar to the one used by the Mercury programming language (Henderson and Somogyi 2002), which would allow for better generated code performance.

References

2018. Dialog. http://www.linusakesson.net/dialog/index.php. Accessed: 2021-08-16.

Evans, R., and Short, E. 2014. Versu—a simulationist storytelling system. *IEEE Transactions on Computational Intelligence and AI in Games* 6(2):113–130.

Henderson, F., and Somogyi, Z. 2002. Compiling mercury to high-level c code. In *Proceedings of the 2002 International Conference on Compiler Construction*. Grenoble, France: Springer.

2006. Inform 7. http://inform7.com. Accessed: 2021-09-27.

Kreminski, M., and Wardrip-Fruin, N. 2018. Sketching a map of the storylets design space. In *ICIDS 2018: Interactive Storytelling*, 160–164. Dublin, Ireland: Springer.

Martens, C. 2021. Ceptre: A language for modeling generative interactive systems. *Proceedings of the AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment* 11(1):51–57.

2020. Simpleqbn. https://github.com/videlais/simple-qbn. Accessed: 2021-08-16.

2021. Storyletmanager. https://github.com/dmasad/StoryletManager. Accessed: 2021-08-16.

2019. Tinyqbn. https://github.com/JoshuaGrams/tiny-qbn. Accessed: 2021-08-16.

2021. Twine harlowe. https://twine2.neocities.org/#macro_storylet. Accessed: 2021-08-16.