Dia: a Domain Specific Language for Scripted Dialogues and Cutscenes

VLADIMIR KOBZEV, University of Twente, The Netherlands

Modern computer games often rely on presenting a fictional narrative to the player. Translating a written story manuscript for a modern computer game into working interactive narrative is a cumbersome task that requires time investments from story writers, programming teams and technical artists. This paper proposes a Domain Specific Language (DSL) called Dia. Dia aims to simplify the process for a game development team through simple and non-intrusive grammar, as well as extensive functionality for data and function binding.

Additional Key Words and Phrases: Computer Game, Digital Storytelling, Narrative Design, DSL, Tools

1 INTRODUCTION

Game development is a time consuming process prone to a variety of problems. Aside from planning-related difficulties such as unrealistically large scope identification, and its further expansion through feature creep, other usual obstacles are technological and tool problems that greatly increase development time. This potentially leads to the project exceeding allocated budget and, in some cases, risking cancellation [17, 23]. Tool problems generally revolve around the difficulty of using the tools or lack of specialized tools in general [17, 23].

One of the major elements in modern computer games are scripted sequences of events in which the player is presented with the narrative of the game. These events may possess a varying degree of interactivity. The non-interactive events are generally referred to as "cutscenes", in which the player is simply shown a series of events, like a movie [7]. Some games introduce a degree of interactivity into such events, generally through interacting with Non-Player Characters (NPCs) by means of selecting from a list of predetermined options. Such events are referred to as dialogues or dialogue trees. Dialogues interrupt the general flow of the game to let the player interact with other characters and advance the story. Dialogues can also be integrated into cutscenes, such that a series of predetermined actions are interrupted by a choice query from the player.

There are different approaches to creating scripted events in computer games. The most obvious one is to simply hard-code them into the game source code, which may be cumbersome and much less accessible to those unfamiliar with programming [4, 19]. The other involves incorporating a scripting language into the game logic. Scripting languages can be either general-purpose or domain-specific, depending on the requirements of the development team, and can be either text-based, such as Lua or Python, or graphical, such as Unreal Blueprints (formerly known as Kismet) [4, 19]. While visual languages are known to be easier to learn, their application to narrative design may be limited, as storytelling may involve large amounts of data, which, according to Myers [14] is difficult to represent with a visual language. Because of this, the focus will be primarily on text-based languages.

Several existing languages and tools that deal with digital storytelling and narrative design were analyzed, such as SAGA [1], Ink [11], RenPy [11], Twine [2, 11] , and ScriptEase [20]. Among the languages listed, Ink and RenPy are both text-based, and are known to have been used to develop a large number of games, including commercial ones [9, 18]. Ink is a text-based DSL that is simple enough for a non-programmer to learn, due to its simple syntax, and construct complex dialogues and narratives. However, the language provides no support for text markup, and the function bindings not only appear too verbose, but also support a very limited number of built-in types for parameters and return types [10]. This leads to Ink users inventing custom tag and message formats that are meant to be manually parsed after Ink compilation, which can be error-prone. RenPy, on the other hand, provides a lot of expressive constructs for text markup [21], but exists as its own Python-based engine for Visual Novels (VNs). This is convenient if the goal is to design a classic visual novel, but at the same time comes with its own technical limitations, such as being limited strictly to 2D Visual Novels. RenPy also requires at least some degree of Python knowledge, which is a general-purpose programming language that takes time for a non-programmer to learn. This results in a gap in which there is no language that is easy to learn, is capable of being adapted to different interactive genres, and provides extensive data and function binding.

The general problem of tool development and tool usability, as well as the problem of readability and accessibility of languages lead to the following research questions:

- **RQ1**: What grammar constructs are needed for a scripted event to be expressed, such as a dialogue or a cutscene?
- **RQ2**: How do constructs for data and function binding, as well as markup affect source code readability?

Thus, the goal is to design a DSL that is both easy to use and is flexible enough to adapt to a variety of games with different degrees of interactivity. The ease of use in this case is based on using familiar syntactic constructs from popular markup languages, and readability comparable to other popular text-based DSLs for game dialogues. Flexibility is achieved by offloading any complex logic and data implementation strictly outside the language domain, while retaining the ability to reference it. The language will be called Dia, short for "Dialogue".

First, to help establish a grammar for the language, the paper describes how domain analysis was conducted. Then, an in-depth language description is presented. Afterwards, a comparison between Dia and another DSL, is shown with regards to readability and feature differences. The DSL for the comparison purposes was chosen to be Ink, as it is a text-based DSL with known successful application. Finally, the paper concludes with a discussion of findings and their relation to their respective RQs.

TScIT 39, July 7, 2023, Enschede, The Netherlands

2 DOMAIN ANALYSIS

One of the first steps to DSL design is the identification of the domain in which the language operates. This process is referred to as Domain Analysis [12]. The particular formal method of Domain Analysis that was selected for this language was Bunge-Wand-Weber (BWW) ontology [22]. This widely accepted method was chosen as it is described as the leading ontology for domain analysis [13]. The ontology is modeled using "things", which are elementary units that can either be primitive or composite, consisting of several other things. Sets of things that contain common properties are referred to as classes [22]. Subclasses are classes that inherit properties of its parent class [15]. Finally, things that are coupled together form a system, and the things that interact with the things within that system, but are not a part of the system, are a part of the system environment [22].

The following entities were identified to be relevant in the domain of scripted sequences in computer games, according to the BWW ontology. Similarly to Goncharenko and Zaytsev [3], words in *italics* refer to kinds of BWW ontological constructs, and words in **bold** are the domain-specific concepts. When things affect the history of each other they are said to be coupled [22]. The coupling of the identified entities is represented in Figure 1.

System Environment **Game Logic** is the environment that concerns the business logic and data of the game itself. This environment is not analyzed in depth because different games may employ vastly different game engines, which in turn may affect the composition of this environment. For the sake of this analysis, it is sufficient to assume that **Game Logic** directly controls the system and possesses the data it may require.

Class **Script** represents a vector of **Events**, and is coupled to some **Context**. **Scripts** directly interface with and are under direct control of **Game Logic** to run the game dialogue or a cutscene. *Property* **Iterator** represents which **Event** is to be processed by **Game Logic** at a given moment. *Property* **Sections** defines groupings of **Events**.

Class **Context** represents information that describes the environment in which the **Script** is run. **Context** itself is populated either directly by **Game Logic** or at the request of its **Script**. **Script** and **Context** are separate entities because, technically, the same **Script** can be executed in a variety of different **Contexts**. *Property* **Identifiers** is a set of **Identifiers**.

Class **Identifier** is a referential binding to external functions or data in **Game Logic**.

Class **Event** is a statement that either represents a **Message** in a dialogue, a **Control** that directs execution flow, or an **Action** that notifies **Game Logic** about a context specific activity. All **Events** influence the state of the **Script** that contains them. *Subclasses* of **Event** are as follows.

Subclass **Message** is an **Event** that represents a unit of text that is expected to be printed on the screen at the time of execution. This subclass refers to and encompasses any kind of printed text information, whether it is a line of monologue, a passage exchanged by actors within a certain context, an email, etc. *Property* **Text** represents portions of text to be displayed. *Property* **Styles** represents how portions of text are displayed.

Subclass **Control** represents an **Event** that directs the execution flow. This redirection can either be *unconditional*, which simply points to a different **Section** of the script, or *conditional* that selects areas of the script depending on certain options, which are decided by **Game Logic**. A *conditional* **Control** has additional properties **Choices**, **Control Action** and **Outcomes**. **Choices** are conditions that select **Outcomes**, which are **Script** areas, and the **Control Action** is an external function **Identifier** that describes the kinds of **Choices** that are possible for the **Game Logic** to make.

Subclass **Action** is a kind of **Event** that serves as a binding to a function in **Game Logic** through a set **Identifier** in a **Context**. Actions can additionally influence state of other identifiers.

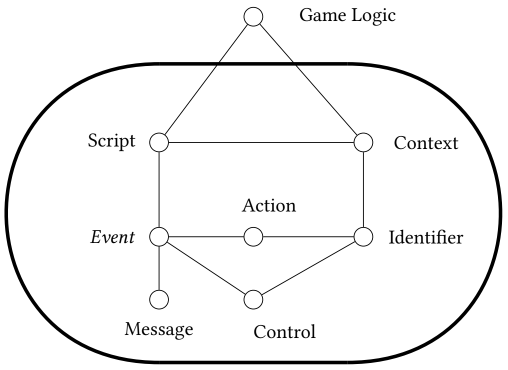

Fig. 1. Domain Entity Coupling

The capsule/stadium shape in Figure 1 represents the BWW system, with the nodes inside of it the individual entities. Lines on the graph are entity couplings. **Event** is shown in italics to highlight that it is used as an abstract class. Entity **Game Logic**, which resides outside the BWW system, represents the environment in which the system operates, and can directly interface with the system through its **Script** and **Context** nodes.

3 LANGUAGE DESCRIPTION

The identified ontology was applied to design the language features of Dia, and the domain entity model was used to ensure that specific language features properly interface with game logic. The following section provides a detailed description of the resulting language. First, subsection 3.1 will introduce basic language features to display and partition dialogue lines. Then subsection 3.2 will demonstrate how data and function binding is handled to interface with the game logic environment. Afterwards, subsection 3.3 will present constructs for built-in and custom text markup. Finally, subsection 3.4 will show how to enable branching narratives by using Options for control flow. A reference grammar is provided in Appendix A.

3.1 Messages and Sections

Dia is designed to be easy to learn for scriptwriters and flexible enough to extend for game programmers. The language utilizes many of the familiar constructs from other markup and programming languages to be more intuitive.

```
# Hello
> Hello World!
> This is a longer dialogue message. If a
  message is too long for a single line, it
  can be indented. > Messages can be chained
  within a single indent block too.
```

Listing 1. Simple Message Display

All messages begin with a greater-than > operator, and must be contained within a section. Messages that are separate but require to be rendered as a single message can be combined together with a plus + operator at the end of a message. This way the interpreter notifies the game logic that, while there is a message to be displayed in its current iteration phase, the next one is supposed to be combined with it.
Akin to Markdown, sections are marked with a pound # sign. Sections can accept Unicode characters, except for characters reserved for operators, but including whitespace. A section or its subsections span until another section or subsection is encountered, or until the end of the file. When the language interpreter encounters the end of a top level section, it halts and notifies the game logic, so that further action can be taken, whether it is to halt the dialogue entirely, or to process a section. All subsections are fall-through.
Execution can jump to a different section or a subsection with a double greater-than » operator. A valid path is either a subsection on the same level, a top-level section, or a point-delimited path from a top-level section. Sections can be returned from with a double less-than « operator.

```
# A
> This is a message from A.
## AA
> And now we have +
>> B.BA
> Came back from B.BA!
> Now execution will pause.

# B
> This will not be displayed.
# BA
> arrived at B.BA!
<<
```

Listing 2. Subsections and Jumps

3.2 Data and Function Binding

Plain text dialogue can be sufficient for some games, but in a lot of cases, audiovisual cues and game events play an important role. For this reason, Dia implements “actions”, which are basic function bindings that can notify game logic on what to do. Before an action can be used, it must be declared. The interpreter is expected to process all declarations in advance and take the appropriate steps to bind them. Declarations are not required to be contained within a text section, and it is encouraged to place all binding declarations at the top of the script file.
Actions can be called inline within a text message or in a separate statement. The former requires the action to be enclosed in parentheses, but no such enclosure is necessary for separate statements.

```
action ding :: Ding

# Example
Ding
> This message demonstrates an
  inline (Ding) action.
```

Listing 3. Simple Binding to Play a Sound

Actions are declared with an external name, and a script-related signature. The external name is used to identify the action within the game logic, and the signature is to be used within the Dia script itself. Signatures can be composed from both Unicode words and type names in arbitrary order, and the interpreter is responsible for deriving which words in the signature correspond to actual types.
The only predefined types in Dia are `String`, `Integer`, `Float`, and `Bool` with reserved `true` and `false` state keywords. Any other type is declared with the `data` keyword. The language is not concerned with the implementation of custom data types, and only expects to be able to pass references of their respective external objects.
Because an action signature can contain words in arbitrary order, strings are distinguished with double quotation marks. Actions can have a return type, which is identified with an arrow -> sign and a return type name in its definition. Such actions can be used to assign data references to variables.
Variables in Dia can be defined both implicitly and explicitly. Implicit definitions rely on the game logic to assign a value in advance. Explicit definitions are handled with the back arrow <- operator. Variable definitions are allowed to reside before text sections, in which case they are processed before any execution takes place. This allows the user to define source files with various common declarations and definitions, and use them as libraries in other source files. Additional script files can be referenced with the `include` keyword.

```
// library.dia
data Portrait
action switch_portrait :: Portrait
action load_portrait :: Load String -> Portrait
Alice: Portrait
Bob <- Load "/assets/faces/bob.png"

// example.dia
include library
# Example
(Alice)> Hi, I'm Alice.
(Bob)> And I am Bob.
```

Listing 4. Data Declarations and Definitions

Dia allows data declarations to contain references to additional typed data related to an instance. The structure of such types does not have to map directly to objects referenced in game logic. Data contained within such a type is accessed with a dot. All references contained within such a type are read-only and may only be resolved by game logic. References are evaluated at runtime.

```
data Actor
data ActorSlot
data VisualNovelScene
  front: ActorSlot
  back:  ActorSlot
  left:  ActorSlot
  right: ActorSlot

LivingRoom: VisualNovelScene
Alice: Actor
Bob: Actor

action place_actor :: Place Actor in ActorSlot
action set_scene   :: VisualNovelScene

# Compound data example
LivingRoom
Place Bob in LivingRoom.left
Place Alice in LivingRoom.right
```

Listing 5. Compound Data Declaration and Usage

3.3 Markup

Dia provides several shorthands for text markup. While actual style implementation is up to the developer, the built-in markup markers are called `*bold*`, `/italic/`, and `_underline_`. On rare occasions, literary fiction requires use of characters that are reserved in the language. While Dia does not provide escape sequences, it does provide a marker for unformatted text with three sequential grave accents "'.

```
# Simple markup
> *bold* /italic/ _underline_
> ```This >text< is displayed *verbatim* (as-is!)```
```

Listing 6. Simple Markup

A few reserved characters for markup may not always be sufficient, especially for games that need more control over how text is displayed, or deal with a variety of concepts that need to catch attention of the player. Some concepts, such as amounts of some in-game currency, may be complimented by an associated icon to help the player with quick concept recognition.
These requirements are covered with range markers and renderers. Markers accept a range of a message, and the interpreter notifies the game logic when a marked portion begins and ends. The markers can be chained together and accept parameters. The range itself is surrounded with equals = signs, and markers are prepended to it.

```
data Color
marker text_color :: Color
marker centered   :: centered
marker morse      :: morse
Red: Color

# Range marker example
> \Red=This text is in red=
> \Red\centered\morse= Oddly specific horizontally
    centered red text in morse code =
```

Listing 7. Custom Markup Ranges

To display data as text, or to generally let the game know that certain game data is being referenced in a specific position of a message, Dia features data renderers. Data renderers act similarly to actions, but with certain limitations. A renderer may only be used inline within a message block, and a renderer has no explicit return type, because a renderer is expected to insert itself as a portion of a text message. Unlike actions, renderers are evaluated immediately as part of a message, which can be useful when the total length of a message is required to be calculated before it is displayed. The example below additionally demonstrates how compound data may be used as well.

```
data Currency
data CurrencyAmount
  currency: Currency
  amount: Integer
data Item
data ShopItem
  item:  Item
  price: CurrencyAmount
data Color

renderer item_print :: Item
renderer item_override_name_print :: Item as String

Red: Color
Badge: Item

# Renderer example
> Hey, that's my {Badge}
> Hey, that's my \Red={Badge}=
> Hey, that's my \Badge.color={Badge as "BADGE"}=
```

Listing 8. Data Renderers

3.4 Control Flow with Options

While linear and predictable flow may be sufficient for some narrative-driven games, or for games that handle branching narrative by means outside the scope of dialogues, control flow is still necessary for a wide rage of narrative-driven games. For this reason, Dia provides the user with the ability to manage control flow by means of an "option" statement.

The built-in option block notifies the game logic and presents a set of text choices. Selecting any of them directs execution flow into the appropriate choice outcome. Choices themselves are indented and surrounded with square brackets. Choice outcomes are either inlined or indented.

```
action ding :: Ding
# Example
? [Option A] > Option A selected
  [Suboptions]
    > Here are some suboptions
    ? [Sub A] (Ding)
      [Sub B] > Suboption B selected
  [Start over] >> Example
```

Listing 9. Built-In Options

Selection from a list of text options may not always be sufficient, and some games provide the ability to interact by additional means, such as presenting items to actors. The language supports extending options via binding. The example below declares a new option type that lets the player select an item from an in-game inventory. The resulting choice can be optionally extracted into a variable with -> to be used within the block. Custom options can also support an empty fallback option.
To prevent repetition in cases when every choice has a common opening line or action call, it is possible to prepend a message block before the set of choices. The block will be executed once a choice is made, but before evaluating the choice result.
Adjacent options are evaluated together, meaning that is is possible to either select the text option or present an item out of order and continue script execution onward. This behavior can be prevented by placing non-option statements between option statements, or separating them with subsections

```
options present_item :: Present -> Item
action show_item :: Show Item
renderer name_to_string :: Item
AttorneysBadge: Item

# Custom options
? > I would like to ask you something
  [Day of the crime]
  > Where were you on the day of the murder?
  [Victim]
  > When was the last time you saw the victim?
? Present -> item
  > I would like to show you something
  [AttorneysBadge]
    > This is my {item} (Show item)
  []> Ah, sorry, not that...
```

Listing 10. Custom Options

Because developer-defined game logic has the authority over choice selection, options do not necessarily have to be explicitly shown to the player, and as such can be extended in creative ways to act as a switch statement in programming languages. Additionally, options can accept parameters similarly to actions. The example below binds an option to an external Pseudo-Random Number Generator (PRNG) and uses it to simulate a coin flip.

```
options dice_roll :: Roll Integer -> Integer
# Coin flip example
? Roll 2
  [1] > Heads!
  [2] > Tails!
```

Listing 11. Using Options as a Switch Statement

Lastly, in some cases, binding may appear excessive. For this, the language provides the ability to tag choices. The example below assumes that the game keeps track of the days of the week, and uses choice tags to selectively show choices depending on week days.

```
? @mon [I hate Mondays]
    > Me too, captain. Let's power through
      this week together
  @tue@wed [What a week, huh]
    > Captain, the week is not even
      halfway over yet!
  [How are the engines?]
    > I'm still working on them
```

Listing 12. Option Labels

4 READABILITY AND COMPLEXITY METRICS

Because the language is interpreted like a program, it can be treated as such to measure script complexity. For this purpose, the complexity analysis will be conducted with Halstead metrics [6, 8]. This particular method of analysis was chosen because the metrics appear to correlate with cognitive load, according to a study using fMRI scanning by Peitek et al. [16]. As designing character interactions for cutscenes and dialogues is taxing enough cognitively as a creative task, the cognitive load imposed by the language is best to be kept to a minimum.
At the same time, Dia scripts mostly consist of commands that display large amounts of text that may easily outnumber operators, the total count of which is used for Halstead metrics. Additionally, Dia is designed to help make external function calls resemble natural language more than a general purpose programming language. This, to a certain extent, also allows to treat script files written in Dia as data files. Thus, to measure readability, a simple data file readability formula proposed by Gryk [5] will be used. It is preferable to let the script writer focus on the text of the story itself, instead of language syntax.
Ink, which was chosen for readability and complexity comparison, can be treated in the same way. While Ink does provide features to write game logic within its scripts, those features are optional, and will not be used in the analysis. Ink also features tags, which have to be manually parsed, but are reported by the interpreter engine, allowing arbitrary-looking external commands. This, similarly to Dia, allows both metrics to be applied.

The following metrics comparison utilizes two scripts. One is a simple dialogue, based on the example script from the Ink manual [10]. This script was chosen in particular because it is a generic dialogue featuring a conversation between two actors with branches and sub-branches, all the while being short. Additionally, this script is intended to present the strongest points of Ink, which is the ability to create and modify branching narrative with simple syntax. One final reason is that the script is a snippet from a published commercial game "80 Days", developed by the designers of Ink.
The other script is the same conversation, but modified to feature on-screen actor interaction and text markup. The lines have also been modified to fit a first-person Visual Novel style narrative. Both scripts are presented in Dia and Ink, with their respective readability and complexity metrics. It should be noted that, for the Dia script with metadata in Listing 15, only the conversation part is taken for the metrics calculations, leaving out the declaration section. This is because the goal is to compare similar executed sections. To avoid inventing additional manually parsed syntax for Ink, all metadata is kept only in tags. This results in markup being applied to whole messages in Dia for the sake of parity of outcomes of scripts in both languages, but nonetheless used.
A dialogue had to be selected in particular, as opposed to a monologue or a polylogue, because dialogues appear to possess a balance between having too little and too much interaction between characters, which requires different levels of metadata representation. A monologue with too little additional information about the scene or its only character would put Ink at an advantage too great to be representative, as a monologue without interaction may be expressed as plain text in this language. Such a monologue would have to be inflated artificially with calls to game logic data, making the comparison difficult to reason about. On the other hand, a polylogue with too much interaction may significantly disadvantage Ink, because large amounts of metadata references would make compiler-time type checking too obvious to declare a necessary feature, which is nonexistent for Ink with manually parsed tags.

```
# Prologue
> I looked at Monsieur Fogg
? [...and I could contain myself no longer.]
  > 'What is the purpose of our journey, Monsieur?'
  > 'A wager', he replied.
  ? [A wager!]
    > 'A wager!' I returned.
    > He nodded.
    ? [But surely that is foolishness!]
      [A most serious matter then!]
    > He nodded again
    ? [But can we win?]
      > 'That is what we will endeavour to find out',
    he answered.
      [A modest wager, I trust?]
      > 'Twenty thousand pounds', he replied, quite
    flatly.
      [I asked nothing of him then]
      > And after a final, polite cough, he offered
    nothing more to me.
    [Ah.]
    > 'Ah', I replied, uncertain what I thought.
  > After that, +
  [...]
  > ...but I said nothing, and +
> we passed the day in silence.
```

Listing 13. Basic dialogue in Dia

```
- I looked at Monsieur Fogg
* ... and I could contain myself no longer.
  'What is the purpose of our journey, Monsieur?'
  'A wager,' he replied.
  ** 'A wager!'[] I returned.
      He nodded.
      *** 'But surely that is foolishness!'
      *** 'A most serious matter then!'
      --- He nodded again.
      *** 'But can we win?'
            'That is what we will endeavour to find
    out,' he answered.
      *** 'A modest wager, I trust?'
            'Twenty thousand pounds,' he replied,
    quite flatly.
      *** I asked nothing further of him then[.], and
    after a final, polite cough, he offered nothing
    more to me. <>
  ** 'Ah[.'],' I replied, uncertain what I thought.
  -- After that, <>
* ... but I said nothing[] and <>
- we passed the day in silence.
- -> END
```

Listing 14. Basic dialogue in Ink

	Dia, Basic	Ink, Basic
Unique operators *n1*	6	6
Total operators *N1*	38	38
Unique operands *n2*	23	25
Unique operands *N2*	23	25
Volume	296.34	312.11
Difficulty	3.00	3.00
Level	0.33	0.33
Effort	899.01	936.34
Characters for identifiers	517	518
Total characters	555	563
Readability	93%	92%

Table 1. Metrics for Basic Scripts

```
data ActorSlot
data VNScene
  left: ActorSlot
  right: ActorSlot
  front: ActorSlot
data Actor
data ScrollSpeed

action scene_set    :: VNScene
action actor_say    :: Actor
action portrait_set :: Actor String
action put_actor    :: Place Actor in ActorSlot
action thinking     :: Thinking
action text_scroll_speed :: ScrollSpeed

Slow: ScrollSpeed
Fast: ScrollSpeed
Office: VNScene
London_Cityscape: VNScene
Passepartout: Actor
Fogg: Actor


# Prologue
Office
(Thinking)> I looked at Monsieur Fogg
Place Fogg in Office.left
? [...and I could contain myself no longer.]
  (Passepartout)> What is the purpose of our journey,
    Monsieur?
  (Fogg)> A wager
  ? [A wager!]
    (Passepartout)(Fast)> *A wager!*
    Fogg nodding
    ? -> answer
      (Passepartout)> {answer}
      [But surely that is foolishness!]
      [A most serious matter then!]
    Fogg nodding
    ? [But can we win?]
      (Fogg)> That is what we will endeavour to find
    out.
      [A modest wager, I trust?]
      (Fogg serious)(Slow)> /Twenty thousand pounds./
      [I asked nothing of him then]
      (Thinking)> And after a final, polite cough, he
    offered nothing more to me.
    [Ah.]
      (Passepartout)> Ah.
      (Thinking)> I was uncertain what I thought that
    moment.
  (Thinking)> After that, +
  [...]
  > ...but I said nothing, and +

London_Cityscape
> we passed the day in silence.
```

Listing 15. Dialogue with metadata in Dia

```
#scene: Office
- I looked at Monsieur Fogg #thinking
- #left: Fogg
* ... and I could contain myself no longer.
  What is the purpose of our journey, Monsieur? #say:
    Passepartout
  A wager #say: Fogg
  ** A wager![] #say: Passepartout #quickly #bold
    --- #actor: Fogg nod
    *** But surely that is foolishness! #say:
    Passepartout
    *** A most serious matter then! #say: Passepartout
    --- #actor: Fogg nod
    *** But can we win? #say: Passepartout
          That is what we will endeavour to find out #
    say: Fogg
    *** A modest wager, I trust? #say: Passepartout
          Twenty thousand pounds, #say: Fogg #italics
    #slowly
    *** I asked nothing further of him then[.], and
    after a final, polite cough, he offered nothing
    more to me. <> #thinking
  ** Ah[.]#say: Passepartout
    --- I was uncertain what I thought that moment. #
    thinking
  -- After that, <>
* ... but I said nothing[] and <>

- #scene: London_Cityscape
- we passed the day in silence.
- -> END
```

Listing 16. Dialogue with metadata in Ink

	Dia, Metadata	Ink, Metadata
Unique operators *n1*	12	7
Total operators *N1*	70	71
Unique operands *n2*	33	39
Unique operands *N2*	47	66
Volume	642.55	756.73
Difficulty	8.55	5.92
Level	0.12	0.17
Effort	5490.85	4482.16
Characters for identifiers	517	662
Total characters	691	751
Readability	89.4%	88.1%

Table 2. Metrics for Scripts with Metadata

5 DISCUSSION

With regards to the first research question concerning the grammar constructs needed to express a scripted event, it was established that using the BWW ontology has helped identify entities related to the domain of research, which in turn has helped translate them into the necessary grammar constructs. This also helped establish the boundaries of the system and how it must interface with its environment. As an answer, it is proposed that the most important constructs must at least describe events that are either information-related (messages), activity-related (actions) and control flow-related (options). These events must exist in a vector that form the script itself, and the script must be aware of the context in which it is being executed. The particular implementation was described in section 3.

As for the second research question on readability impact by data and function binding, the readability analysis has shown that both languages appear very similar in a basic uncomplicated setting. In a complicated environment with metadata, however, the results are beginning to diverge, but not enough to definitively say that one language is more complicated than another. While the readability metric is near identical for both scripts, the Halstead metrics do not show any definite metric that one language bears significantly more cognitive load than another.

While it could be argued that both scripts are similar at a glance, the tag structure that was chosen for Ink script camouflages the fact that Ink tags are actually arbitrary strings that are not checked by the language compiler and require manual parsing to introduce any functionality to them. In a setting where a developer prefers to ensure no errors at runtime from tags, an additional checking system would have to be implemented on top of the language compiler to allow it. This problem is not present in Dia, where all metadata identifiers must be declared and type-checked during compile-time. A similar problem occurs with text markup. While it is possible to introduce and manually parse special syntax in messages for Ink akin to built-in markup and range markers in Dia, doing so would add another error-prone layer of complexity for the developers and maintainers of the game. Such notations would also have to be communicated with and ensured among developers and script writers and as a result lead to further complexity.

Thus, for the second research question, it is concluded that adding metadata constructs to a language bears little impact on its readability, because otherwise, in order to fulfill design requirements, the developer must invent a separate notation system that still affects the readability. While an overall longer program is produced due to necessary type declarations, the executed portion itself is similar, with the added benefit of compiler type and identifier checking. While this introduces more concepts to the language, their benefits outweigh possible increase in steepness of the learning curve.

This paper was produced within significant time limitations, which severely narrowed its scope. For example, due to the lack of time, it was not feasible to design and develop an interpreter for the language that would work within an existing game engine. The lack of an interpreter, combined with the lack of time unfortunately also meant it was impossible to conduct user testing with non-programmer participants. Such a study could provide useful information on how easy Dia actually is to learn, as well as how compares against a language such as Ink. The research resorted to formal metrics as a result.

This research bears several implications on DSL design and tool development. It shows that ontology based domain analysis can be applied to design game development tools in a quick manner, covering all necessary requirements in the process. The research also indicates that cognitive difficulty is not a deeply researched topic, especially for DSLs, which required additional reasoning to classify Dia and Ink as programming languages fit for Halstead software metrics, for example. Lastly, the research shows that there is a potential niche for an interactive narrative language that is both very flexible and simple to understand quickly.

For further research, the most immediate recommendation is development of a language plugin for a game engine. Such a plugin could allow Dia to function within a game logic environment, and allow assessing actual language flexibility in a practical setting. Additionally, while there may have been sufficient reasoning for selecting a specific snippet to conduct an analysis, its subjectivity may still introduce selection bias. As such, an experiment with non-programmer participants is recommended for further research with the use of the developed language plugin. Such an experiment could not only greatly reduce selection bias, as the scripts would be produced by participants instead, but also provide useful information on how non-programmers perceive Dia, as well as digital storytelling tools in general.

REFERENCES

[1] Lucas Beyak and Jacques Carette. 2011. SAGA: A DSL for Story Management. *Domain-Specific Languages*, 66, 48–67. DOI: 10.4204/eptcs.66.3.

[2] Henrik Engström, Jenny Brusk, and Patrik Erlandsson. 2018. Prototyping Tools for Game Writers. *THE COMPUTER GAMES JOURNAL*, 7, 3, (Sept. 1, 2018), 153–172. DOI: 10.1007/s40869-018-0062-y.

[3] Boryana Goncharenko and Vadim Zaytsev. 2016. Language design and implementation for the domain of coding conventions. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Software Language Engineering* (SLE 2016). Association for Computing Machinery, New York, NY, USA, (Oct. 20, 2016), 90–104. ISBN: 978-1-4503-4447-0. DOI: 10.1145/2997364.2997386.

[4] Jason Gregory. 2019. *Game Engine Architecture.* (3rd ed.). CRC Press. ISBN: 978-1-138-03545-4.

[5] Michael R. Gryk. 2022. Human Readability of Data Files. *Balisage Series on Markup Technologies*, (July 30, 2022). DOI: 10.4242/balisagevol27.gryk01.

[6] Maurice H. Halstead. 1977. *Elements of Software Science (Operating and Programming Systems Series).* Elsevier Science Inc., USA, (Apr. 1977). 128 pp. ISBN: 978-0-444-00205-1.

[7] Hugh Hancock. 2002. Better Game Design Through Cutscenes. Game Developer. (Apr. 2, 2002). Retrieved Apr. 30, 2023 from https://www.gamedeveloper.com/design/better-game-design-through-cutscenes.

[8] T Hariprasad, G Vidhyagaran, K Seenu, and Chandrasegar Thirumalai. 2017. Software complexity analysis using halstead metrics. In *2017 International Conference on Trends in Electronics and Informatics (ICEI).* (May 2017), 1109–1113. DOI: 10.1109/ICOEI.2017.8300883.

[9] [SW] Joseph Humfrey, Ink Library June 1, 2023. inkle. URL: https://github.com/inkle/ink-libraryRetrieved June 10, 2023 from.

[10] Joseph Humfrey. 2023. Writing with Ink. GitHub. (Jan. 21, 2023). Retrieved June 18, 2023 from https://github.com/inkle/ink.

[11] Mika Letonsaari. 2019. Nonlinear Storytelling Method and Tools for Low-Threshold Game Development. *Seminar.net*, 15, 1, (June 14, 2019), 1–17. DOI: 10.7577/seminar.3074.

[12] Marjan Mernik, Jan Heering, and Anthony M. Sloane. 2005. When and how to develop domain-specific languages. *ACM Computing Surveys*, 37, 4, (Dec. 1, 2005), 316–344. DOI: 10.1145/1118890.1118892.

[13] Daniel Moody. 2009. The "Physics" of Notations: Toward a Scientific Basis for Constructing Visual Notations in Software Engineering. *IEEE Transactions on Software Engineering*, 35, 6, (Nov. 2009), 756–779. DOI: 10.1109/TSE.2009.67.

[14] Brad A. Myers. 1990. Taxonomies of visual programming and program visualization. *Journal of Visual Languages & Computing*, 1, 1, (Mar. 1, 1990), 97–123. DOI: 10.1016/S1045-926X(05)80036-9.

[15] Jeffrey Parsons and Yair Wand. 1997. Using objects for systems analysis. *Communications of the ACM*, 40, 12, (Dec. 1, 1997), 104–110. DOI: 10.1145/265563.265578.

[16] Norman Peitek, Sven Apel, Chris Parnin, André Brechmann, and Janet Siegmund. 2021. Program Comprehension and Code Complexity Metrics: An fMRI Study. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. (May 2021), 524–536. DOI: 10.1109/ICSE43902.2021.00056.

[17] Fábio Petrillo, Marcelo Pimenta, Francisco Trindade, and Carlos Dietrich. 2009. What went wrong? A survey of problems in game development. *Computers in Entertainment*, 7, 1, (Feb. 27, 2009), 13:1–13:22. DOI: 10.1145/1486508.1486521.

[18] 2023. Ren'Py Games List. Retrieved June 10, 2023 from https://games.renpy.org/.

[19] Simon Renger. 2022. *Investigation into the Criteria of Embeddability of Visual Scripting Languages within the Domain of Game Development.* (June 23, 2022). DOI: 10.13140/RG.2.2.11976.39686.

[20] Jonathan Schaeffer et al. 2007. ScriptEase: A Generative/Adaptive Programming Paradigm for Game Scripting. *ERA*. DOI: 10.7939/R33F4M223.

[21] 2023. Text — Ren'Py Documentation. (June 22, 2023). Retrieved July 1, 2023 from https://www.renpy.org/doc/html/text.html.

[22] Yair Wand and Ron Weber. 1995. On the deep structure of information systems. *Information Systems Journal*, 5, 3, 203–223. DOI: 10.1111/j.1365-2575.1995.tb00108.x.

[23] Michael Washburn, Pavithra Sathiyanarayanan, Meiyappan Nagappan, Thomas Zimmermann, and Christian Bird. 2016. What went right and what went wrong: an analysis of 155 postmortems from game development. In *Proceedings of the 38th International Conference on Software Engineering Companion* (ICSE '16). Association for Computing Machinery, New York, NY, USA, (May 14, 2016), 280–289. ISBN: 978-1-4503-4205-6. DOI: 10.1145/2889160.2889253.

A GRAMMAR

```
script = bind_section? text_section*

bind_section = (bind | var_declaraction)+

text_section = heading statement+
statement = msg | bind | var_declaraction | action | option_block | subheading | goto | return


heading = "#" TEXT NEWLINE
subheading = "#"{2+} TEXT NEWLINE

msg_block = msg NEWLINE (INDENT ((msg_content | msg) NEWLINE)+ OUTDENT)?
msg = ">" msg_content
msg_content = (TEXT | action | renderer | range | goto | return)+

range = ("\" WORD+ )+ "=" msg_content "="
renderer = "{" WORD+ "}"
action = "(" WORD+ ")"

option_block = "?" (WORD+)? ("->" WORD)? NEWLINE
               INDENT (msg_block)* (option_choice (msg_block|NOTHING) NEWLINE)+ OUTDENT
option_choice = "[" (TEXT) "]"

goto = ">>" TEXT
return = "<<"

bind = bind_data | bind_action | bind_option
bind_data = "data" WORD (NEWLINE INDENT var_declaration+ OUTDENT)? NEWLINE
bind_option = "options" WORD "::" WORD+ -> WORD NEWLINE
bind_action = "action" WORD "::" WORD+ (-> WORD)? NEWLINE
bind_renderer = 'renderer' WORD '::' WORD+ NEWLINE
bind_rangemarker = 'marker' WORD '::' WORD+ NEWLINE

var_declaration = var_name : type NEWLINE
var_assignment = var_name (: type)? "<-" action NEWLINE
var_name = WORD
type = WORD

import = "import" filename NEWLINE

filename = TEXT

WORD = //a single alphanumeric word identifier
TEXT = //valid unicode text excl reserved characters
STRING = //valid unicode

INDENT // indentation level increase
OUTDENT // indentation level returned to previous state
NEWLINE // new line
```