Multiverse: An Interactive Storytelling System

Drew C. Taylor
drew.taylor@gtri.gatech.edu

Overview

Multiverse is an interactive storytelling system based on the Universe [1, 2] model of storytelling. Multiverse consists of a Multiverse programming language and the Multiverse player. The programming language allows an author to define goals, strategies to achieve those goals, and entities, such as characters, to support the story. The player compiles and presents the interactive story to a reader.

Related Work

In the Universe model, Lebowitz describes storytelling as an effort to satisfy author goals [2]. Author goals specify story states, such as the separation of a couple, that the author wishes to occur.

To satisfy an author goal, the system chooses from among strategies (here, plot fragments in Lebowitz) known to satisfy that goal. For example, to satisfy the goal of separating a couple, an author may provide a number of strategies to achieve that goal, such as a fight about the relationship, a disagreement over work, an affair, or an accident.

The strategies may be simple: an accident may simply kill one partner, thereby separating the couple. The strategies may also be complex: an affair may include a precondition that one partner has an intimate relationship with another character, which the system may convert into a romantic one; or it may include a subgoal to create such a relationship. Thus the strategies may themselves require satisfaction of additional author goals.

Wide Ruled [3, 4], an interactive storytelling system also based on the Universe model, inspires this system. Wide Ruled allows authors with little or no programming experience to write interactive stories based on the Universe Model. The system provides a user interface to walk the author through the creation of characters, environments, plot points, author goals, and plot fragments.

In contrast to Wide Ruled, Multiverse focuses on authors with some programming experience. Rather than provide wizards to define the story, the system provides a specialized programming language. The use of a programming language permits Multiverse to expand upon the capabilities of Wide Ruled, allowing the author to define more complex entities and strategies. The system also provides insight into its operation, allowing the author to inspect its memory and its effort to satisfy goals.

Design

In Multiverse, a story consists of an initial state and a set of goals. The system generates a story through the satisfaction of those goals, and the system supports interactivity by allowing the reader to select a goal to satisfy.

States. A state consists of a set of entities and a set of relationships between those entities. An entity consists of a type and a set of attributes, each of which consists of a name and a value. A relationship consists of a left entity, a right entity, and a set of attributes. An author may define an entity or a relationship in the initial state or in a strategy. The following example illustrates the definition of two characters entities, romeo and juliet, and a relationship between them:

```
character romeo {
  name   := "Romeo"
  house  := "Montague"
  gender := "male"
  age    := 15
  alive  := true
}

character juliet {
  name   := "Juliet"
  house  := "Capulet"
  gender := "female"
  age    := 13
  alive  := true
}

romeo<->juliet {
  marriage := true
}
```

Listing 1: Entities and Relationships

Goals and Strategies. Each goal consists of a set of strategies, any one of which may satisfy the goal. Each strategy consists of a sequence of statements, and each statement produces a successor state from the current state. The following example illustrates two simple goals, marriage and divorce, each of which consists of two strategies: in the case of marriage, "Love" and "Arrangement", and in the case of divorce, "By Death" and "By Divorce".

```
goal marriage(character c1, character c2) "Marriage" {
  strategy "Love" {
    ~ a sequence of steps ~
  }

  strategy "Arrangement" {
    ~ a sequence of steps ~
  }
}

goal separation(character c1, character c2) "Separation" {
  strategy "By Death" {
    ~ a sequence of steps ~
  }

  strategy "By Divorce" {
    ~ a sequence of steps ~
  }
}
```

Listing 2: Goals and Strategies

Statements. A statement is an assignment statement, which assigns a value to an identifier; a query statement, which finds an entity and assigns that entity to an identifier; a narration statement, which sets narrative text; or a subgoal statement, which directs the system to satisfy a goal. The following examples illustrate each of these statements:

```
juliet.alive := false

character c1, character c2 ?= (c1<->c2).marriage == true

"<p>For never was a story of more woe <br>
Than this of {{c1.name}} and {{c1.det}} {{c2.name}}.</p>"

separation(romeo, juliet)
```

Listing 3: Assignment, Query, Narration, and Subgoal Statements

Generation. To generate a story, the system first identifies the goals that are satisfiable from the initial state. To do so, the system attempts to satisfy each goal given the initial state.

To satisfy a goal, the system lists the set of strategies for the goal, shuffles the list, and attempts to satisfy each strategy. If the system can satisfy at least one strategy from the initial state, then

the system can satisfy the goal. If the system cannot satisfy any strategy from the initial state, then the system cannot satisfy the goal.

To satisfy a strategy, the system attempts to execute each of its statements. The execution of a statement may be a success - given the current state, it produces a successor state - or it may be a failure - given the current state, it cannot produce a successor state. If every statement is a success, then the system can satisfy the strategy; if any statement is a failure, then the system cannot satisfy the strategy.

Having identified those goals that it can satisfy, the system presents those satisfiable goals to the reader, who may select the goal they wish to the system to satisfy. The system then proceeds to satisfy the selected goal as described above.

Having satisfied the goal, the system returns the sequence of states from the initial state to the current state, which the system presents to the user as a sequence of narrative text.

The following is a screenshot of the system at this point in story generation. At left, the system displays the goals available to the user, with the selected goal underlined; at right, the system displays the text of the story.

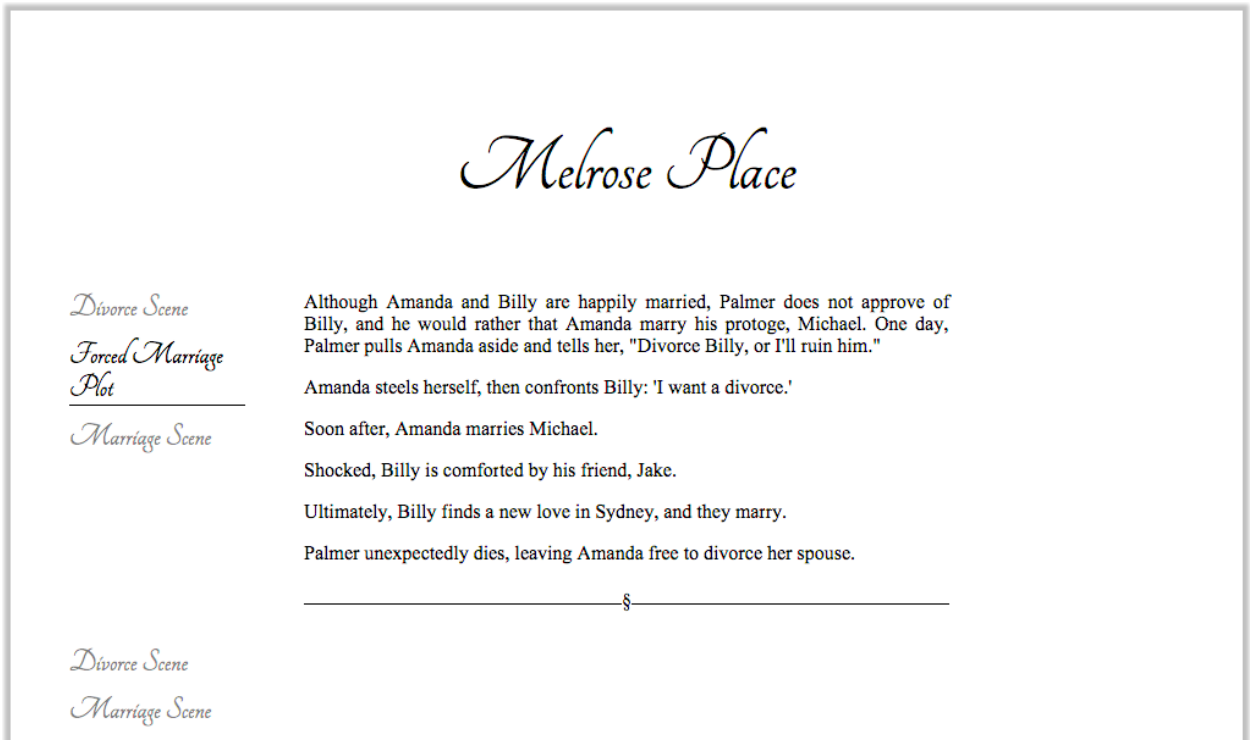

Figure 1: Story Generation

At this point, the system again attempts to satisfy each goal with the current state, so it may again present those goals to the reader, who continues to interact with the system.

Examples

Multiverse Story. The following example illustrates a simple, complete Multiverse story.

```
story "Once Upon a Time" 0

state {

  character aurora {
    name := "Aurora"
  }

  character phillip {
    name := "Phillip"
  }

}

goal tell() "Tell Story" {

  strategy "Marriage" {
    character c1, character c2 ?=
      c1 != c2 &&
      (c1<->c2).marriage == empty

    "<p>Once upon a time, {{c1.name}} and {{c2.name}}
     met and married."

    (c1<->c2).marriage := true
  }

```

```
28    strategy "Happily Ever After" {
29      character c1, character c2 ?
30        c1 == c2 ||
31        (c1<->c2).marriage == true
32        =
33        (c1<->c2).marriage == true
34
35      "<p>{{c1.name}} and {{c2.name}} lived happily
36       ever after.</p>"
37    }
38  }
```

Listing 4: An Example Multiverse Story

Line 1 specifies the title of the story, "Once Upon a Time", and the random seed, 0, for the system to use.

Lines 3-13 define the initial state of the story. Lines 5-7 define a character entity for which the name attribute is "Aurora". Lines 9-11 define a character entity for which the name attribute is "Phillip".

Lines 15-38 define a goal, "Tell Story", with two strategies. Lines 17-26 define the first strategy - "Marriage" - and Lines 28-37 define the second strategy - "Happily Ever After".

In the "Marriage" strategy, Lines 18-20 are a query statement. The query attempts to find two character entity values, c1 and c2, which are distinct ("`c1 != c2`") and for which no marriage relationship exists ("`(c1<->c2).marriage == empty`").

Lines 22-23 are a narration statement. Assuming the names of characters c1 and c2 are "Aurora" and "Phillip", respectively, this narration statement produces an HTML paragraph containing the text "Once upon a time, Aurora and Phillip met and married."

Line 25 is an assignment statement. An assignment statement changes the value of an attribute. In this assignment statement, the bidirectional relationship between c1 and c2 has an attribute, marriage, which the system sets to true.

In the "Happily Ever After" strategy, Lines 29-33 are a query statement. Unlike the first query, which finds any two characters for which the expression is true, this query establishes than an expression is true for every two characters, then finds a particular pair for which another expression is true.

The first expression (Lines 30-31) establishes that for every two characters c1 and c2, the two characters are either the same ("`c1 == c2`") or a marriage relationship exists between the two characters ("`(c1<->c2).marriage == true`").

The second expression (Line 33) attempts to find two character entity values c1 and c2 for which a marriage relationship exists between the two characters ("`(c1<->c2).marriage == true`").

Lines 33-34 are a narration statement. Again assuming the names of characters c1 and c2 are "Aurora" and "Phillip", respectively, this narration statement produces an HTML paragraph containing the text "Aurora and Phillip lived happily ever after."

For a detailed specification of the language, see Appendix A: Multiverse Language Reference.

Multiverse Player. To play a story in Multiverse, the reader uses the Multiverse player. The player is a Scala application that runs on the Java 8 Virtual Machine. To run the player, the reader runs the following command from the command line:

```
$ java -jar multiverse.jar "Once Upon a Time.multiverse"
```

The player attempts to compile the given story. If the story includes an error, the player informs the reader of the error and exits; if the story does not include an error, the player presents the reader with a window similar to the following:

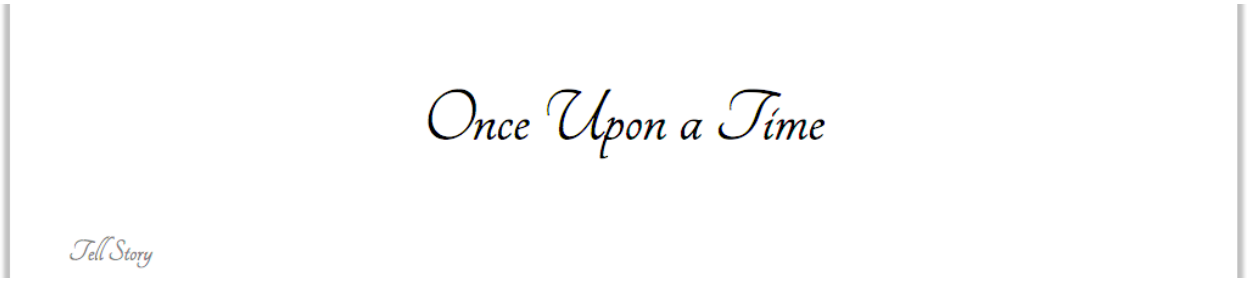

Figure 1: Player Window, Initial State

At top, the player displays the title of the story; at left, the player displays the satisfiable goals given the initial state. In "Once Upon A Time", the goal it may satisfy is the "Tell Story" goal.

On the command line, the player provides the reader with a description of the initial state and its effort to determine the satisfiable goals from that state:

```
Story Entity Sets
    Story Entity "character"
        ID 0
            name := "Aurora"
        ID 1
            name := "Phillip"
Bidirectional Relationships
Unidirectional Relationships

Goal "Tell Story" (Success)
    Strategy "Marriage" (Success)
        Event "Unlabeled Event 4" (Success)
        Event "Unlabeled Event 5" (Success)
        Event "Unlabeled Event 6" (Success)
        Event "Unlabeled Event 7" (Success)
```

Figure 2: Player Command Line Output, Initial State

The player displays the entity values, bidirectional relationships, and unidirectional relationships in the initial state. The player groups the entity values by their type.

The player then displays its effort to determine the satisfiable goals from the initial state. The player attempted the satisfy the "Tell Story" goal with the "Marriage" strategy; as every statement in the strategy is a success, the "Marriage" strategy is a success, and the "Tell Story" goal is a success.

To select the goal the reader wishes to satisfy, the reader clicks the name of the goal. The player satisfies the goal and displays its narrative text in the window:

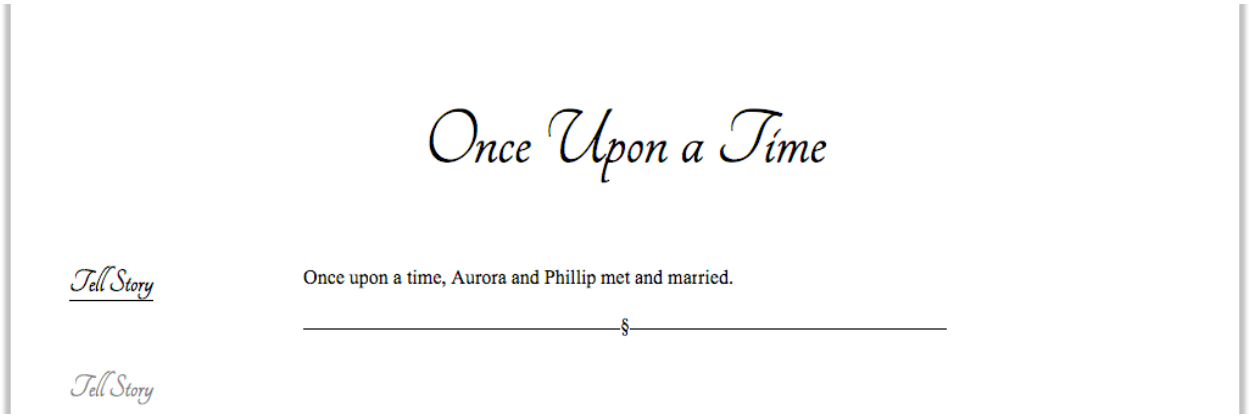

Figure 2: Player Window, Next State

At left, the player underlines the "Tell Story" goal, indicating that it is the goal that the reader selected. At right, the player displays the narrative text from the goal, "Once Upon a Time, Aurora and Phillip met and married."

On the command line, the player provides the reader with a description of its effort to satisfy the selected goal, a description of the state following the satisfaction of that goal, and and a description of its effort to determine the satisfiable goals from that state.

```
Goal "Tell Story" (Success)
    Strategy "Marriage" (Success)
        Event "Unlabeled Event 4" (Success)
        Event "Unlabeled Event 5" (Success)
        Event "Unlabeled Event 6" (Success)
        Event "Unlabeled Event 7" (Success)
    Strategy "Happily Ever After" (Failure)
        Event "Unlabeled Event 8" (Failure)

Story Entity Sets
    Story Entity "character"
        ID 0
            name := "Aurora"
        ID 1
            name := "Phillip"
Bidirectional Relationships
    (character, 0) <-> (character, 1)
        marriage := "true"
Unidirectional Relationships


Goal "Tell Story" (Success)
    Strategy "Happily Ever After" (Success)
        Event "Unlabeled Event 8" (Success)
        Event "Unlabeled Event 9" (Success)
    Strategy "Marriage" (Failure)
        Event "Unlabeled Event 4" (Success)
        Event "Unlabeled Event 5" (Failure)
```

Figure 3: Player Command Line Output, Next State

To satisfy the selected "Tell Story" goal, the player attempted to satisfy the "Happily Ever After" strategy; this failed. The player then attempted to satisfy the "Marriage" strategy; this succeeded.

Following the success of the selected "Tell Story" goal, the state includes a bidirectional relationship between the character with system id 0 (Aurora) and the character with system id 1 (Phillip).

Finally, the player displays its effort to determine the satisfiable goals from that state. In this case, the player attempted the satisfy the "Tell Story" goal with the "Marriage" strategy; this failed. The player then attempted to satisfy the "Happily Ever After" strategy; this is a success.

Again, the reader may click the name of the goal to satisfy. The player satisfies the goal and displays its narrative text in the window:

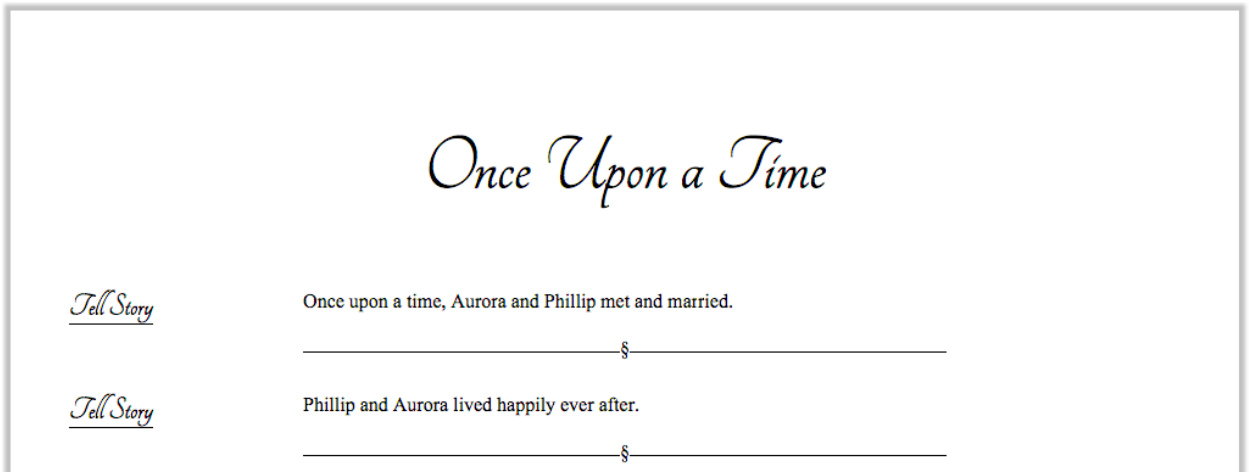

Figure 3: Player Window, Final State

Future Work

Types. Currently, the type of an entity serves only to group a set of entities under the same label; it does not require that the entities under that label have a specific set of attributes. Meanwhile, attributes and relationships have no type. Although this flexibility has an appeal, it also permits errors that types would reveal. A future version of Multiverse would allow the user to define an entity type as a specific set of attributes and to define the type of an attribute and a relationship.

Packages. The system does not provide any way for a story to incorporate entities, relationships, or goals from other stories. A packaging system would permit authors to share and reuse these components, allowing them to leverage the work of others to produce larger, more complex stories.

Random Attributes. When an author creates an entity, the author must specify the values of its attributes. This significantly limits the variety of entities that the author can create. For example, if an author cannot rely on the system to assign a unique name to a character at runtime, it is difficult to create distinguishable characters at runtime.

Gender-Dependent Nouns. The system provides pseudo-attributes like "sub" and "obj" to allow authors to include gender-appropriate subject and object pronouns for entities with gender attributes, but it does not provide a generic way to specify an attribute whose value depends on the gender of the entity. For example, it does not permit the author to specify a "child" attribute whose value is "girl" or "boy". Providing a way to do so would allow authors to easily write better stories.

References

[1] Lebowitz, M. (1984). Creating characters in a story-telling universe. *Poetics*, *13*(3), 171-194.

[2] Lebowitz, M. (1985). Story-telling as planning and learning. *Poetics*, *14*(6), 483-502.

[3] Skorupski, J., Jayapalan, L., Marquez, S., & Mateas, M. (2007). Wide ruled: A friendly interface to author-goal based story generation. In *Virtual Storytelling. Using Virtual Reality Technologies for Storytelling* (pp. 26-37). Springer Berlin Heidelberg.

[4] Skorupski, J., & Mateas, M. (2009). Interactive story generation for writers: Lessons learned from the Wide Ruled authoring tool. *Digital Arts and Culture 2009*.

[5] Turner, S. R. (1993). Minstrel: a computer model of creativity and storytelling.

[6] Wirth, N. (1996). Extended Backus-Naur Form (EBNF). *ISO/IEC*, *14977*, 2996.

Appendix A: Multiverse Language Reference

This appendix defines the Multiverse language. It begins with a definition of a value in the context of the language; the representation of those values, literals; references to those values, identifiers; and expressions of those values. It continues with a description of the statements available to the language: assignment statements, query statements, narration statements, and subgoal statements. Finally, it describes the use of those statements to construct strategies, goals, and stories.

Values

A value is a set of zero or more members; a member is a boolean, a number, a string, an entity, a relationship, or a set. A set may include members of different types: for example, a set may include both a boolean and a number.

A boolean is either the boolean true or the boolean false. A number is any number. A string is any string.

An entity consists of a type and a set of attributes, each of which consists of a name and a value. An attribute name is unique among the names of the attributes of its entity.

A relationship consists of a left entity, a right entity, and a set of attributes. A relationship is either a bidirectional relationship or a unidirectional relationship.

In the case of a bidirectional relationship, the order of the left entity and the right entity is not significant: the bidirectional relationship with the left entity A and the right entity B is the bidirectional relationship with the left entity B and the right entity A.

In the case of a unidirectional relationship, the order is significant: the unidirectional relationship with the left entity A and the right entity B is not the unidirectional relationship with the left entity B and the right entity A.

For simplicity, this appendix may use the phrase "boolean value" to refer to a value that is a set of one boolean; "number value" to refer to a value that is a set of one number; "string value" to refer to a value that is a set of one string; "entity value" to refer to a value that is a set of one entity; and "relationship value" to refer to a value that is a set of one relationship.

Literals

A literal is a representation of a value. A literal is a boolean literal, a number literal, a string literal, or the empty set literal.

A boolean literal is either `true` or `false`. The `true` literal represents a set with one member, the boolean true; similarly, the `false` literal represents a set with one member, the boolean false.

A number literal is an optional sign, optionally followed by zero or more digits followed by a decimal, followed by one or more digits. Examples include `1`, `-1`, `1.0`, `-1.0`, `0.1`, `-0.1`, `.1`, and `-.1`. The number literal represents a set with one member, the number. The following productions from the lexical grammar describe the number literal:

```
number = [ '-' ], [ { digit }, '.' ], digit, { digit }

digit  = ? any character for which
           scala.runtime.RichChar.isDigit returns true ?
```

Listing A-1: Number Literal Productions

A string literal is a double quote, followed by zero or more characters (except double quote) or escape sequences, followed by a double quote. Examples include `""`, `"string"`, and `"\"string\""`. The string literal represents a set with one member, the string. The following productions from the lexical grammar describe the string literal:

```
string     = '"', { escape | character - '"' }, '"'

escape     = '\', character

character  = ? any character ?
```

Listing A-2: String Literal Productions

The empty set literal is `empty`; it represents a set with no members.

Identifiers

An identifier is a letter followed by zero or more letters or digits. The following productions from the lexical grammar describe an identifier:

```
identifier = letter, { letter | digit }

digit      = ? any character for which
               scala.runtime.RichChar.isDigit returns true ?

letter     = ? any character for which
               scala.runtime.RichChar.isLetter returns true ?
```

Listing A-3: Identifier Productions

A qualified identifier is a sequence of one or more identifiers; it is a reference to a value. The following productions from the syntactic grammar describe a qualified identifier:

```
qualified-identifier = identifier, { '.', identifier }
                     | '(', rel-identifier, ')', '.',
                       identifier, { '.', identifier }

rel-identifier       = qualified-identifier, '<->',
                       qualified-identifier
                     | qualified-identifier, '->',
                       qualified-identifier
```

Listing A-4: Qualified Identifier Productions

In the case of a sequence of identifiers separated by periods, the first identifier must reference an entity value. If followed by an identifier, that identifier must indicate an attribute of that entity: the identifier references the value of that attribute. For each identifier that follows, the preceding value must be an entity value and the identifier must indicate an attribute of that entity: the identifier references the value of that attribute.

For example, suppose there is a character entity value, Oedipus. That character has an attribute, mother, the value of which is another character entity value, Jocasta. Finally, that entity has an attribute, name, the value of which is a string value, "Jocasta".

Further suppose that the following qualified identifier references the character entity value Oedipus:

```
oedipus
```

Listing A-5: Qualified Identifier `oedipus`

The following qualified identifier therefore references the character entity value Jocasta.

```
oedipus.mother
```

Listing A-6: Qualified Identifier `oedipus.mother`

Finally, the following qualified identifier references the string value "Jocasta".

```
oedipus.mother.name
```

Listing A-7: Qualified Identifier `oedipus.mother.name`

In the case of a sequence of identifiers that begins with a relationship identifier, the qualified identifier must begin with a "(", followed by a qualified identifier, followed by either a "->" or a "<->" separator, followed by a qualified identifier, followed by a ")".

The first and second qualified identifiers must each reference an entity value. The first referenced entity value is the left entity of the relationship; the second referenced entity value is the right entity of the relationship.

If the qualified identifier includes the "<->" separator, the reference is to the bidirectional relationship between the left entity and the right entity. If the qualified identifier includes the "->" separator, the reference is to the unidirectional relationship between the left entity and the right entity.

Unlike an entity, the language cannot reference a relationship itself. Thus an identifier must follow the relationship and that identifier must indicate an attribute of that relationship: the identifier references the value of that attribute.

Similar to an entity, for each identifier that follows, the preceding value must be an entity value and the identifier must indicate an attribute of that entity: the identifier references the value of that attribute.

For example, suppose there is a bidirectional relationship in which the left entity is the character entity value Oedipus and the right entity is the character entity value Jocasta. That relationship has an attribute, marriage, the value of which is the boolean value true.

The following qualified identifier references that boolean value.

```
(oedipus<->jocasta).marriage
```

Listing A-8: Qualified Identifier `(oedipus<->jocasta).marriage`

The following qualified identifier also references that boolean value, as the qualified identifier oedipus.mother references the character entity value Jocasta.

```
(oedipus<->oedipus.mother).marriage
```

Listing A-9: Qualified Identifier `(oedipus<->oedipus.mother).marriage`

Operators

An operator is a function that returns a value. An operator is either a unary operator or a binary operator. In the case of a unary operator, the function accepts one value; in the case of a binary operator, the function accepts two values.

The language provides boolean, equality, relational, set, and arithmetic operators. The table below summarizes these operators. In each table, column 1 indicates the operator itself. Columns 2 and 3 indicate the expected cardinality and type of the left value, with the asterisk indicating that any cardinality or any type may appear. Similarly, columns 4 and 5 indicate the cardinality and type of the right value. Finally, columns 6, 7, and 8 indicate the cardinality, type, and content of the return value.

operator	**left value**		**right value**		**return value**		
	#	**type**	**#**	**type**	**#**	**type**	**a set containing . . .**
boolean operators							
`or`	1	boolean	1	boolean	1	boolean	the logical or of the member of the left set and the member of the right set
`and`	1	boolean	1	boolean	1	boolean	the logical and of the member of the left set and the member of the right set
`not`	1	boolean			1	boolean	the logical not of the member of the left set
equality operators							
`==`	*	*	*	*	1	boolean	true if every member of the left set appears in the right set and every member of the right set appears in the left set; false otherwise.
`!=`	*	*	*	*	1	boolean	true if any member of the left set does not appear in the right set or any member of the right set does not appear in the left set; false otherwise.
relational operators							
`<=`	1	number	1	number	1	boolean	true if the member of the left set is less than or equal to the member of the right set; false otherwise.

<	1	number	1	number	1	boolean	true if the member of the left set is less than the member of the right set; false otherwise.
>=	1	number	1	number	1	boolean	true if the member of the left set is greater than or equal to the member of the right set; false otherwise.
>	1	number	1	number	1	boolean	true if the member of the left set is greater than the member of the right set; false otherwise.
`subset`	*	*	*	*	1	boolean	true if every member of the left set appears in the right set
`superset`	*	*	*	*	1	boolean	true if every member of the right set appears in the left set
set operators							
`difference`	*	*	*	*	*	*	the members of the left set that do not appear in the right set
`intersection`	*	*	*	*	*	*	the members that appear in both the left set and the right set
`union`	*	*	*	*	*	*	the members that appear in the left set and the members that appear in the right set
#	n	any			1	number	the number that is the cardinality of the left set
arithmetic operators							
+	1	number	1	number	1	number	the number that is the sum of the member of the left set and the member of the right set

−	1	number	1	number	1	number	the number that is the member of the left set minus the member of the right set
*	1	number	1	number	1	number	the number that is the product of the member of the left set and the member of the right set
/	1	number	1	number	1	number	the number that is the member of the left set divided by the member of the right set
++			1	number	1	number	the member of the left set plus 1
−−			1	number	1	number	the member of the left set minus 1

Table A-1: Operators

An operation may be a success or a failure: if either the left value or the right value is not of the expected cardinality or the expected value, the operation is a failure; if both the left value and the right value are of the expected cardinality and the expected value, the operation is a success.

Expressions

An expression is a literal, a qualified identifier, a unary expression (a unary operator and an expression), a binary expression (a left expression, a right expression, and a binary operator), or a parenthesized expression.

An expression has a value, calculated as follows:

the value of a literal is the value represented by the literal;

the value of a qualified identifier is the value referenced by the qualified identifier;

the value of a unary expression is the value returned by its unary operator given the value of its unary expression;

the value of the binary expression is the value returned by its binary operator given the values of its left and right expressions; and

the value of a parenthesized expression is the value of its expression.

The system evaluates expressions in this order of precedence:

literals, qualified identifiers, and parenthesized expressions;

unary expressions with postfix operators ("++" and "--");

unary expressions with prefix operators ("#" and "not");

binary expressions with multiplication and division ("*" and "/");

binary expressions with addition and subtraction ("+" and "-")

binary expressions with set operators ("union", "intersection", "difference");

binary expressions with relational operators ("<", "<=", ">", ">=", "superset", "subset");

binary expressions with equality operators ("==", "!=");

binary expressions with or; and

binary expressions with and.

If two expressions have the same precedence, the left expression takes precedence over the right expression.

For example, the value of the following expression evaluates to 4, as the addition operator takes precedence over the union operator.

```
3 + 1 union 1 + 3

4 union 1 + 3

4 union 4

4
```

Listing A-10: Order of Operations, Example 1

In contrast, the value of the following expression evaluates to 7, as the expression in parentheses takes precedence over the addition operator.

```
3 + (1 union 1) + 3

3 + 1 + 3

4 + 3

7
```

Listing A-11: Order of Operations, Example 2

An expression may be a success or a failure: if any operation in the expression is a failure, the expression is a failure; if every operation in the expression is a success, the expression is a success.

Statements

A statement is an assignment statement, a query statement, a narration statement, or a subgoal statement. Execution of a statement may be a success or a failure.

Assignment Statements

An assignment statement sets the value of an attribute, and it may declare the entity or the relationship of that attribute. The following productions describe the syntax of the assignment statement:

```
assignment          = qualified-identifier, ':=', or-expression
                    | entity
                    | relationship

entity              = identifier, identifier, assignment-block

relationship        = rel-identifier, assignment-block

assignment-block    = '{', { assignment }, '}'
```

Listing A-12: Assignment Statement Productions

In the first form of the assignment statement, the qualified identifier indicates an attribute and the assignment sets the value of the attribute to the value of the expression.

For example, the following assignment statement sets the value of the attribute marriage to the boolean value true.

```
(oedipus<->oedipus.mother).marriage := true
```

Listing A-13: Assign the value of an attribute.

In the second form of the assignment statement, the statement creates an entity value of the specified type and binds that value to the specified qualified identifier. In the assignment block that follows, each identifier indicates an attribute of the entity, and each assignment sets the value of that attribute.

For example, the following assignment statement creates an character entity value and binds that value to the qualified identifier jocasta. It then sets the value of the attribute name to a string value ("Jocasta") and the value of the attribute gender to a string value ("female").

```
character jocasta {
  name   := "Jocasta"
  gender := "female"
}
```

Listing A-14: Assign Entity, Example 1

The following assignment statement creates an character entity value and binds that value to the qualified identifier oedipus. It then sets the value of the attribute name to a string value ("Oedipus"), the value of the attribute gender to a string value ("male"), and the value of the attribute mother to a character entity value (jocasta).

```
character oedipus {
  name   := "Oedipus"
  gender := "male"
  mother := jocasta
}
```

Listing A-15: Assign Entity, Example 2

In the third form of the assignment statement, the statement retrieves a specified relationship. In the assignment block that follows, each identifier indicates an attribute of the relationship, and each assignment sets the value of that attribute.

For example, the following assignment statement retrieves the bidirectional relationship between the character entity value oedipus and the character entity value jocasta. In the assignment block that follows, it sets the value of the attribute marriage to the boolean value true and the attribute marriageYear to the the number value 500.

```
oedipus <-> jocasta {
  marriage     := true
  marriageYear := 500
}
```

Listing A-16: Assign Relationship

In the first form of the assignment statement, the statement is a success if the qualified identifier references an attribute and the expression is a success; otherwise, it is a failure.

In the second form of the assignment statement, the statement is a success if the qualified identifier to which the system attempts to bind the entity value is not bound to another entity value; if every qualified identifier in the assignment block is an attribute; and if every expression in the assignment block is a success.

Query Statements

A query statement evaluates an expression in the context of a state, and it binds one or more qualified identifiers to the same number of entities. The following productions describe the syntax of the query statement:

```
query          = [ parameter-list ], '?=', or-expression
               | [ parameter-list ], '?',  or-expression,
                 [ '=', or-expression ]

parameter-list = identifier, identifier,
                 { ',', identifier, identifier }
```

Listing A-17: Query Statement Productions

If the parameter list is absent, the query statement behaves as an if statement: if both expressions in the query statement evaluate to true, then the query statement is a success; if at least one expression is a failure or evaluates to false, then the query statement is a failure.

If the parameter list is present, it describes a set: each member of the set contains an entity of the given type bound to the given qualified identifier, and the set contains every permutation of entities of the given types.

For example, the following parameter list describes a set, each member of which contains a character entity value bound to the identifier "hero" and a character entity value bound to the identifier "pal"; the set contains every permutation of two characters.

```
character hero, character pal
```

Listing A-18: Query Parameter List

If the state included four characters entity values - say, Hercules, Iolaus, Xena, and Gabrielle - the parameter list specifies the following set:

hero Hercules pal Hercules	hero Hercules pal Iolaus	hero Hercules pal Xena	hero Hercules pal Gabrielle
hero Iolaus pal Hercules	hero Iolaus pal Iolaus	hero Iolaus pal Xena	hero Iolaus pal Gabrielle
hero Xena pal Hercules	hero Xena pal Iolaus	hero Xena pal Xena	hero Xena pal Gabrielle
hero Gabrielle pal Hercules	hero Gabrielle pal Iolaus	hero Gabrielle pal Xena	hero Gabrielle pal Gabrielle

Table A-2: Query Parameters

The system uses this set to evaluate the query.

For each member of the set, the system evaluates the query expression. If the query employs the `?=` operator, the query statement is a success if the query expression is true for at least one member of the set; the query is a failure is the query expression is a failure or is false for every member of the set.

If the query employs the `?` operator, the query is a success if the first expression is true for every member of the set and if the second expression is true for at least one member of the set; the query is a failure if the first expression is a failure or is false for at least one member of the set or if the second expression is a failure or is false for every member of the set.

For example, the following query (using the `?=` operator) is a success, as there exists at least one member of the set for which the hero and the pal are different entities:

```
character hero, character pal ?= hero != pal
```

Listing A-19: Query Example 1

However, the following query (using the `?` operator) is a failure, as for all members of the set, this is not true:

```
character hero, character pal ? hero != pal
```

Listing A-20: Query Example 2

Now suppose that every character has an attribute, alive, for which the value is true. In this case, the following query (using the ? operator) is a success, as for all members of the set, the hero is alive and the pal is alive:

```
character hero, character pal ? hero.alive && pal.alive
```

Listing A-21: Query Example 3

Finally, suppose that a relationship exists between the characters Hercules and Iolaus and between the characters Xena and Gabrielle, and that each relationship has an attribute, partner, for which the value is true. Further suppose that Hercules and Xena include an attribute, leader, for which the value is true, and that Iolaus and Gabrielle include an attribute, follower, for which the value is true. In this case, the following query (using the ? operator and the = operator) is a success:

```
character hero, character pal
  ? hero.alive && pal.alive
  = (hero<->pal).partner && hero.leader && pal.follower
```

Listing A-22: Query Example 4

Above, for all members of the set, the hero is alive and the pal is alive, and for at least one member of the set, there exists a partner relationship between the hero and the pal, the hero is a leader, and the pal is a follower.

If the query is a success, the system randomly selects one member of the set for which the query expression is true, binds the entity to its identifier, and makes the identifier available to the strategy.

In the example above, the query expression is true for the following two members of the set:

hero Hercules pal Iolaus	hero Xena pal Gabrielle

Table A-3: Query Parameters for which Query Expression is true

The system randomly selects one, and in the strategy in which the query appears, the identifiers hero and pal reference Hercules and Iolaus (if the system selects the member on the left) or Xena and Gabrielle (if the system selects the member on the right).

Narration Statements

A narration statement sets the narration text for the state. It consists of a literal string, and it may include HTML formatting and Handlebars templates.

The following example sets the narration text for the state to an HTML paragraph containing the text "Once upon a time . . ."

```
"<p>Once upon a time . . .</p>"
```

Listing A-23: Narration Statement with HTML

The following example sets the narration text for the state to an HTML paragraph containing the text "Xena and Gabrielle wandered an Ancient Greece that looked very much like a modern New Zealand."

```
"<p>{{hero.name}} and {{pal.name}} wandered an Ancient
 Greece that looked very much like a modern New Zealand.</p>"
```

Listing A-24: Narration Statement with Handlebars Templates

If an entity includes an attribute with the name "gender", the value of which is one of "male", "female", or "neuter", the system supplies pseudo-attributes to support pronouns of the appropriate gender. The following example sets the narration text for the state to "Xena hurled her chakram at her opponent, killing him."

```
"<p>{{hero.name}} hurled {{hero.det}} chakram at {{hero.det}}
 opponent, killing {{opponent.obj}}."
```

Listing A-25: Narration Statement with Pronouns

The following table summarizes the pseudo-attributes the system supplies for each gender.

pseudo-attribute	**gender**		
	male	**female**	**neuter**
Sub	He	She	It
sub	he	she	it

Obj	Him	Her	It
obj	him	her	it
Pos	his	hers	its
pos	his	hers	its
Det	His	Her	Its
det	his	her	its
Ref	Himself	Herself	Itself
ref	himself	herself	itself

Table 3: Pronoun Pseudo-Attributes

A narration statement is always a success.

Subgoal Statements

A subgoal statement directs the system to satisfy a goal. The statement consists of a goal identifier and a sequence of zero or more arguments, each of which is a qualified identifier that references an entity value. The following productions describe the syntax of the subgoal statement:

```
subgoal              = identifier, '(', [ argument-list ], ')'

argument-list        = qualified-identifier,
                       { ',', qualified-identifier }
```

Listing A-26: Subgoal Statement Productions

A subgoal statement is a success only if the types of the entity values in the argument list match the types of the parameters in the parameter list and if the attempt to satisfy the goal is a success. Otherwise, the subgoal statement is a failure.

Strategies

A strategy is a sequence of statements and an optional label. The following productions describe the syntax of a strategy:

```
strategy-declaration = 'strategy', [ string ], strategy-block

strategy-block       = '{', { statement }, '}'
```

```
statement              = subgoal | query | assignment | narration
```

Listing A-27: Strategy Productions

A strategy may be a success or a failure: it is a success only if every statement is a success; otherwise, it is a failure.

For example, consider the following strategy:

```
strategy "Establishing Scene" {
  character hero, character pal ?=
        hero.alive &&
        pal.alive &&
        (hero<->pal).partner &&
        hero.leader &&
        pal.follower

  "<p>{{hero.name}} and {{pal.name}} wandered an Ancient
  Greece that looked very much like a modern New
  Zealand.</p>"

  fight(hero)
}
```

Listing A-28: Strategy Example

The strategy above is a success only if the query statement is a success, the narration statement is a success, and the subgoal statement is a success; otherwise, the strategy is a failure.

Goals

A goal consists of an identifier, a sequence of zero or more parameters, an optional label, and a set of strategies. The following productions describe the syntax of a goal:

```
goal-declaration     = 'goal', identifier,
                       '(', [ parameter-list ],  ')',
                       [ string ], goal-block

goal-block           = '{', { strategy-declaration }, '}'
```

Listing A-29: Goal Productions

For each parameter, the system binds the entity value to the qualified identifier, and it makes these identifiers available to the strategies for goal.

A goal may be a success or a failure: it is a success if at least one strategy is a success; otherwise, it is a failure.

For example, consider the following goal:

```
goal separation(character c1, character c2) "Separation" {
  strategy "By Death" {
    ~ a sequence of steps ~
  }

  strategy "By Divorce" {
    ~ a sequence of steps ~
  }
}
```

Listing A-30: Goal Example

The goal above is a success only if either the strategy "By Death" is a success or the strategy "By Divorce" is a success; otherwise, the goal is a failure.

Stories

A story consists of an optional header, an optional state, and zero or more goals. The following productions describe the syntax of a story:

```
unit                = [ story ],
                      [ state-declaration ],
                      { goal-declaration }

story               = 'story', string, [ number ]

state-declaration   = 'state', state-block

state-block         = '{', { assignment }, '}'
```

Listing A-31: Story Productions

A story header consists of a title and an optional random seed. For example, the following story header specifies a title of "Romeo and Juliet" and a random seed of zero:

```
story "Romeo and Juliet" 0
```

Listing A-32: Story Header

A state consists of a sequence of zero or more assignment statements. For example, the following state specifies two entity values, both of type character:

```
character romeo {
  name   := "Romeo"
  house  := "Montague"
  gender := "male"
  age    := 15
  alive  := true
}

character juliet {
  name   := "Juliet"
  house  := "Capulet"
  gender := "female"
  age    := 13
  alive  := true
}
```

Listing A-33: State

Appendix B: Syntactic Grammar

The EBNF grammar [6] below describes the Multiverse syntactic grammar, which specifies the transformation of a sequence of tokens into a syntactically correct program. The sections below divide productions of the grammar into stories, states, goals, and strategies; argument and parameter lists; statements; expressions; identifiers; and literals.

Stories, States, Goals, and Strategies

```
unit                = [ story ],
                      [ state-declaration ],
                      { goal-declaration }

story               = 'story', string, [ number ]

state-declaration   = 'state', state-block

state-block         = '{', { assignment }, '}'

goal-declaration    = 'goal', identifier,
                      '(', [ parameter-list ],  ')',
                      [ string ], goal-block

goal-block          = '{', { strategy-declaration }, '}'

strategy-declaration = 'strategy', [ string ], strategy-block

strategy-block      = '{', { statement }, '}'
```

Argument and Parameter Lists

```
argument-list       = qualified-identifier,
                      { ',', qualified-identifier }

parameter-list      = identifier, identifier,
                      { ',', identifier, identifier }
```

Statements

```
statement           = subgoal | query | assignment | narration

subgoal             = identifier, '(', [ argument-list ], ')'

narration           = string
```

```
query               = [ parameter-list ], '?=', or-expression
                    | [ parameter-list ], '?', or-expression,
                      [ '=', or-expression ]

assignment          = entity
                    | relationship
                    | qualified-identifier, ':=', or-expression

entity              = identifier, identifier, assignment-block

relationship        = rel-identifier, assignment-block

assignment-block    = '{', { assignment }, '}'
```

Expressions

```
or-expression       = and-expression,
                      { or-operator, and-expression }

or-operator         = '||' | 'or'

and-expression      = equal-expression,
                      { and-operator, equal-expression }

and-operator        = '&&' | 'and'

equal-expression    = rel-expression,
                      { equal-expression, rel-expression }

equal-expression    = '==' | '!='

rel-expression      = set-expression,
                      { rel-operator, set-expression }

rel-operator        = '>=' | '<=' | '>' | '>='
                    | 'subset' | 'superset'

set-expression      = add-expression,
                      { set-operator, add-expression }

set-operator        = 'union' | 'difference' | 'intersection'

add-expression      = mul-expression,
                      { add-operator, mul-expression }
```

```
add-operator          = '+' | '-'

mul-expression        = unary-expression,
                        { mul-operator, unary-expression }

mul-operator          = '*' | '/'

unary-expression      = [ unary-operator ], postfix-expression

unary-operator        = '!' | 'not' | '#'

postfix-expression    = atom-expression, [postfix-operator]

postfix-operator      = '--' | '---' | '++' | '+++'

atom-expression       = literal
                      | qualified-identifier
                      | '(', expression-or, ')'
```

Identifiers

```
qualified-identifier = '(', rel-identifier, ')',
                       '.', identifier, { '.', identifier }
                     | identifier, { '.', identifier }

rel-identifier       = qualified-identifier, '<->',
                       qualified-identifier
                     | qualified-identifier, '->',
                       qualified-identifier
```

Literals

```
literal              = "true" | "false"
                     | "empty"
                     | number | string
```

Listing B-1: Syntactic Grammar

Appendix C: Lexical Grammar

The EBNF grammar [6] below describes the Multiverse lexical grammar, which specifies the transformation of Multiverse source into a sequence of tokens. Briefly, a token may be one of the following:

> an identifier, one letter followed by zero or more letters or digits;
>
> a number, an optional sign, optionally followed by one digit followed by a decimal, followed by one or more digits;
>
> a string, zero or more characters (except double-quote) or escape sequences, enclosed in double-quotes; or
>
> an operator, any reserved operator.

Whitespace and comments separate tokens. Whitespace is zero or more characters that serve to represent horizontal or vertical space, such as spaces, tabs, newlines, carriage returns. Comments are zero or more characters (except tilde) or escape sequences, enclosed in tildes.

Note that in Multiverse, both strings and comments continue until they are closed by a double-quote or a tilde, respectively. Unlike some other languages, which require that a string close before a newline character, Multiverse strings may include newline characters. Similarly, some other languages permit a newline character to close a comment; in multiverse, only a tilde may close a comment.

```
token      = identifier | number | string | operator

identifier = letter, { letter | digit }

number     = [ '-' ], [ { digit }, '.' ], digit, { digit }

string     = '"', { escape | character - '"' }, '"'

ignorable  = comment | whitespace

comment    = '~', { escape | character - '~' }, '~'

escape     = '\', character

operator   = ? any reserved symbol token ?

whitespace = ? any character for which
               scala.runtime.RichChar.isWhitespace return true ?
```

```
digit      = ? any character for which
               scala.runtime.RichChar.isDigit returns true ?

letter     = ? any character for which
               scala.runtime.RichChar.isLetter returns true ?

character  = ? any character ?
```

Listing C-1: Lexical Grammar

Appendix D: Reserved Tokens

The tables below list reserved tokens.

The first includes alphanumeric tokens. These reserved tokens include operators (such as "`and`", "`or`", and "`not`"), literals (such as "`true`", "`false`", and "`empty`"), and keywords (such as "`story`", "`state`", and "`goal`"). Multiverse source may not employ these tokens as entity or attribute identifiers.

`and`	`false`	`not`	`story`	`superset`
`difference`	`goal`	`or`	`strategy`	`true`
`empty`	`intersection`	`state`	`subset`	`union`

Table D-1: Reserved Alphanumeric Tokens

The second includes symbol tokens. These reserved tokens include operators (such as "`&&`", "`||`", and "`!`") and separators (such as "`{`", "`(`", and "`->`").

`&&`	`||`	`!`	`+`	`-`	`*`	`/`	`++`
`+++`	`--`	`--`	`<`	`<=`	`>`	`>=`	`#`
`==`	`!=`	`:=`	`+=`	`-=`	`?=`	`?`	`=`
`{`	`}`	`(`	`)`	`<->`	`->`	`.`	`,`

Table D-2: Reserved Symbol Tokens