Shahid Beheshti University
Faculty of Computer Science and Engineering

TECHNICAL REPORT

Title:	RAVI: A Framework for Formal Narrative Authoring
Student Name:	Saeed Amiri-Chimeh
Student ID:	96543003
Supervisor:	Dr. Hassan Haghighi - Dr. Mojtaba Vahidi-Asl
Report ID:	CSE-GLB-0172-1153
Last Update:	October 19, 2022

1 Introduction

This document reports on a formal framework for authoring and modeling interactive narratives. Using this framework, the author can formally specify the interactivity of the narrative and automatically generate the narrative model as a multiple-entry automaton. Also, authors can run the narrative model and interact with it like a text-based adventure game.

Moreover, this framework allows authors to use built-in query blocks to declare assertions about the narrative. Consequently, the proposed framework is capable of automatic assertion checking which reduces the validation time and labour.

Finally, this document presents two different case-studies to demonstrate the capabilities of the proposed system and its Python implementation.

2 Ravi

In this section, we start by introducing formal definitions and theorems that constitute our approach to the representation, authorship, and validation of interactive narratives. Then, we present algorithms that can be used to generate formal narrative models from given user specifications. Afterward, we introduce a set of helper operations to facilitate the construction and validation of user-defined narrative assertions.

Throughout this section, we provide some examples to better explain every component of our approach. All of these examples are based on a simple interactive narrative titled "The Conflict". We refer to this narrative as the *example narrative* throughout the rest of this section. Figure 1 presents this narrative.

The Conflict

You start the narrative as a simple college student who has found a mysterious book. The book reveals an ancient conflict between two secret cults over the destiny of human race. One of them is called the 'Army of Virtu' which is led by the 'General'. The other faction is called the 'Clan of Chaos' which is led by the 'Preceptor'. You can choose to ally with any of these factions.

If you choose to ally with the Army, you have two options. First, you can fight the Preceptor and defeat the Clan. Second, you can betray the Army and institute the 'Order of Doubt'. If you chose to fight the Preceptor, you can still betray the Army and institute the Order later. Similarly, if you chose to institute the Order, you can still fight the Preceptor and defeat the Clan. As the founder of the Order, you can also fight the General and defeat the army.

If you choose to ally with the Clan, you have two options. First, you can fight the General and defeat the Army. Second, you can betray the Clan and institute the Order of Doubt. If you chose to fight the General, you can still betray the Clan and institute the Order later. Similarly, if you chose to institute the Order, you can still fight the General and defeat the Army. As the founder of the Order, you can also fight the Preceptor and defeat the Clan.

There are three ending scenarios for this interactive narrative. If you have defeated the General, it is possible to create absolute freedom with the expense of igniting the fire of anarchy all over the world. If you have defeated the Preceptor, it is possible to end all wars with the expense of establishing a tyranny. If you have defeated both the General and the Preceptor, then you can create a world where the people have the authority to choose their own destiny.

Figure 1: An interactive narrative that resembles a simple adventure game.

2.1 Overview

Before jumping to formal definitions and theorems, we briefly explain how authors are going to use our formal approach for developing an interactive narrative. In summary, they would

go through the following steps:

1. The authors first define the narrative context, which consists of sets of variables that are classified into author-defined narrative entity classes.

2. They define narrative choices as pairs of narrative transforms and pre-conditions. Narrative transforms are functions that take the state of the world context as their input, change the given state arbitrarily, and then return the altered state. Pre-conditions are narrative filters, functions that take a state as their input and return a boolean value that determines if the given state satisfies a certain property. Pre-conditions are guards that determine if their corresponding choice is permitted in a given narrative state.

3. Then, one or more initial states and any number of termination conditions are declared for the narrative. Similar to pre-conditions, termination conditions are narrative filters and take a state as their input and return a boolean value. Now, `Ravi` can automatically construct a state machine that presents every reachable narrative state from the initial ones, along every feasible path from any initial state to any state that satisfies at least one of the termination conditions.

4. The authors define some assertions by chaining and mixing a set of pre-defined helper operations that extract data from narrative models.

5. Finally, they can check the generated model against the assertions. For example, an author can check the impossibility of a player marrying a certain character if the player has chosen to kill that character before. Also, the author can run the modeled narrative like a text-based adventure game. This enables the author to experience the narrative, make choices, and test how the story would unfold.

Figure 2 visually summarizes the above-mentioned steps and their related concepts.

2.2 Formal Narrative Representation

Here, we introduce formal definitions and theorems that establish our approach to formal narrative modeling.

2.2.1 Entity Class

Definition 2.1. We define a *property* as an ordered pair consisting of a symbol and a set. Given that $p = (p_{symbol}, p_{set})$ is a property:

- We say p is named p_{symbol} with the domain of p_{set}.

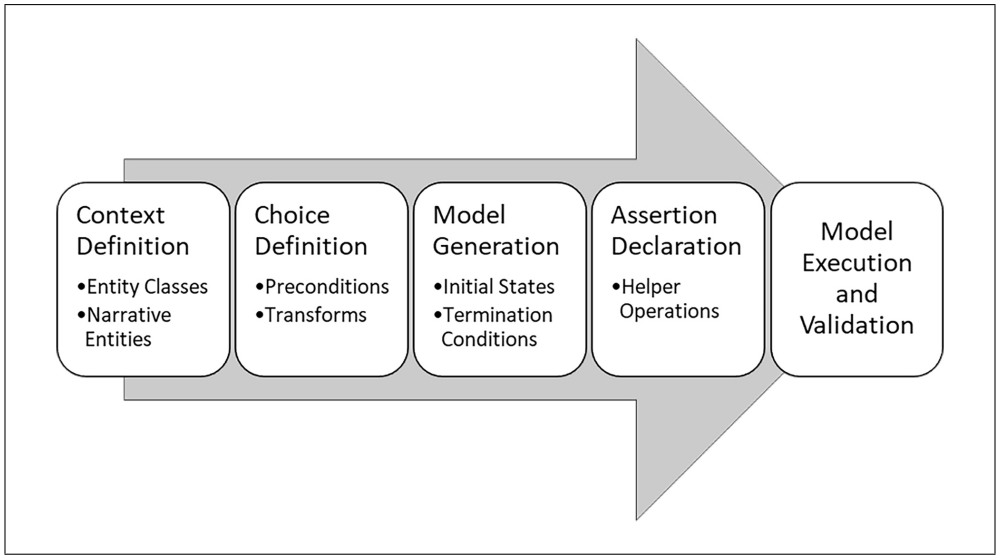

Figure 2: The process of narrative authoring in `Ravi`

- We write $p = p_{symbol} : p_{set}$ as it is a more convenient notation.
- We say two properties conflict if and only if their names are identical.

Definition 2.2. We define an *entity class* as an n-tuple of non-conflicting properties. Also, for any entity class

$$c = (N_1 : D_1, \ldots, N_n : D_n)$$

we define the domain of c as

$$DOM(c) = \prod_{i=1}^{n} D_i$$

Moreover, Given two entity classes

$$c1 = (N_1 : D_1, \ldots, N_n : D_n)$$
$$c2 = (M_1 : E_1, \ldots, M_m : E_m)$$

we say $c2$ inherits from $c1$ and write

$$c2 = c1 + (M_{n+1} : E_{n+1}, \ldots, M_m : E_m)$$

if and only if $(n < m)$ and $\forall 1 \leq i \leq n : (N_i = M_i) \wedge (D_i = E_i)$.

Example 2.3. Considering the example narrative, we define three entity classes of *character*, *player*, and *world* as follows.

$$characterClass = (isAlive : BOOL)$$
$$playerClass = characterClass + (alliance : FACTION)$$
$$worldClass = (status : STATUS)$$

where

$$\begin{aligned} BOOL &= \{T, F\} \\ FACTION &= \{army, clan, order, none\} \\ STATUS &= \{anarchy, tyranny, freedom, none\} \end{aligned}$$

2.2.2 Narrative Entity

Definition 2.4. A *narrative entity* is defined as the pair $(name, class)$, where *name* is a symbol and *class* is an entity class. Given that $e = (name_e, class_e)$ is a narrative entity:

- We say e is an instance of $class_e$ named $name_e$.
- For convenience, we write $e = name_e : class_e$.
- Similar to properties, we say two narrative entities conflict if and only if their names are identical.

A narrative entity can resemble any object or concept. For example, authors can model player inventory, game world, resources, and characters by defining related entities.

2.2.3 Narrative Context

The stage in which the narrative unfolds consists of many entities such as characters, locations, and resources. Combining these entities establishes the environment in which the player experiences the narrative. Here, we formalize this notion as the context of the narrative.

Definition 2.5. We define a *narrative context* as an n-tuple of non-conflicting narrative entities. Furthermore, for any narrative context

$$nc = (name_1 : class_1, \ldots, name_n : class_n)$$

we define the narrative space of nc as

$$SPACE(nc) = \prod_{i=1}^{n} DOM(class_i)$$

and say s is a narrative state of nc if and only if $s \in SPACE(nc)$.

Definition 2.6. Given the narrative context

$$nc = (name_1 : class_1, \ldots, name_n : class_n)$$

and any state of nc like $s = (v_1, \ldots, v_n)$, we reference v_i by $s[name_i]$ for any $1 \leq i \leq n$. Additionally, let's assume $class_i = (N_1 : D_1, \ldots, N_m : D_m)$ has m properties. Consequently, we can assume $v_i = (v_{i1}, \ldots, v_{im})$. Hereafter, we reference v_{ij} by $v_i[N_j] = s[name_i][N_j]$ for any $1 \leq j \leq m$.

Also, given $x \in SPACE(class_i)$, we define $s[name_i][N_j] \leftarrow x$ to be a clone of s like s' where $s'[name_i][N_j] = x$. We use this notation to create new states by substituting the value of an entity property in a given state.

Example 2.7. We can use the classes that we introduced in Example 2.3 and define the following narrative context for the example narrative:

$$\begin{aligned} exampleCntxt = (&player : playerClass,\\ &preceptor : characterClass,\\ &general : characterClass,\\ &world : worldClass) \end{aligned}$$

According to this definition, $s = ((T, clan), (T), (F), (tyranny))$ is a state of $exampleCntxt$. Consequently, we can say:

- $s[player][isAlive] = T$
- $s[player][alliance] = clan$
- $s[preceptor][isAlive] = T$
- $s[general][isAlive] = F$
- $s[world][status] = tyranny$

Moreover, we can say:

$$s[player][alliance] \leftarrow order = ((T, order), (T), (F), (tyranny))$$

2.2.4 Narrative Filters

In an interactive narrative, the player faces different choices under different circumstances. In other words, the narrative might offer a particular choice to the player only if a specific condition is satisfied. Therefore, we need a mechanism to check possible player choices for every narrative state. The following definitions formalize this concept.

Definition 2.8. Given the narrative context nc, we refer to any function from $SPACE(nc)$ to $\{T, F\}$ as a *narrative filter* for nc.

Definition 2.9. Assuming nc be a narrative context and f be a filter for nc, we define $PASS(f) = \{s \in SPACE(nc) | f(s) = T\}$.

In other words, $PASS(f)$ is the set of all narrative states that satisfy the condition implied by the filter f.

Example 2.10. Considering the example narrative, we define the following filters for the previously defined $exampleCntxt$:

$$noAlliance(s) = \begin{cases} T, & s[player][alliance] = none \\ F, & otherwise \end{cases}$$

$$memberOfArmy(s) = \begin{cases} T, & s[player][alliance] = army \\ F, & otherwise \end{cases}$$

$$memberOfClan(s) = \begin{cases} T, & s[player][alliance] = clan \\ F, & otherwise \end{cases}$$

$$canFightArmy(s) = \begin{cases} T, & s[player][alliance] \in \{clan, order\} \wedge \\ & s[general][isAlive] = T \\ F, & otherwise \end{cases}$$

$$canFightClan(s) = \begin{cases} T, & s[player][alliance] \in \{army, order\} \wedge \\ & s[preceptor][isAlive] = T \\ F, & otherwise \end{cases}$$

$$isAnarchyPossible(s) = \begin{cases} T, & s[player][alliance] \in \{clan, order\} \wedge \\ & s[general][isAlive] = F \\ F, & otherwise \end{cases}$$

$$isTyrannyPossible(s) = \begin{cases} T, & s[player][alliance] \in \{army, order\} \wedge \\ & s[preceptor][isAlive] = F \\ F, & otherwise \end{cases}$$

$$isFreedomPossible(s) = \begin{cases} T, & s[player][alliance] = order \wedge \\ & s[preceptor][isAlive] = F \wedge \\ & s[general][isAlive] = F \\ F, & otherwise \end{cases}$$

2.2.5 Narrative Transform

By making a choice, players move the narrative forward. Here, this means changing the variables of one or more narrative entities. The following definition formalizes this transformative aspect of narrative choices.

Definition 2.11. Given the narrative context nc, we refer to any function on $SPACE(nc)$ as a *narrative transform* for nc. Later, we show how narrative transforms can be used to change states of a given narrative context.

Example 2.12. Considering the example narrative, we define the following transforms for $exampleCntxt$:

$$\begin{aligned}
joinArmyTransform(s) &= s[player][alliance] \leftarrow army \\
joinClanTransform(s) &= s[player][alliance] \leftarrow clan \\
instituteOrderTransform(s) &= s[player][alliance] \leftarrow order \\
defeatGeneralTransform(s) &= s[general][isAlive] \leftarrow F \\
defeatPreceptorTransform(s) &= s[preceptor][isAlive] \leftarrow F \\
instateAnarchyTransform(s) &= s[world][status] \leftarrow anarchy \\
instateTyrannyTransform(s) &= s[world][status] \leftarrow tyranny \\
instateFreedomTransform(s) &= s[world][status] \leftarrow freedom
\end{aligned}$$

2.2.6 Narrative Choice

As mentioned before, every narrative choice is conditional and transformative. These two aspects of narrative choices correspond to the following two questions:

- When is a choice possible?
- What happens if the player makes a choice?

Filters and transforms can formally provide answers to these questions. Accordingly, we formally define narrative choices as follows.

Definition 2.13. Given the narrative context nc, a *narrative choice* in nc is the pair

$$(precondition, action)$$

where $precondition$ is a filter and $action$ is a transform for nc. Henceforth, we refer to $CHOICE(nc)$ as the set of all choices in nc.

Definition 2.14. Assuming nc be a narrative context and $(q, r) \in SPACE(nc)^2$ and $\chi = (precondition, action)$ be a choice in nc, we write $q \xrightarrow{\chi} r$ if and only if:

$$q \in PASS(precondition) \wedge r = action(q) \tag{1}$$

Informally, we say if $q \xrightarrow{\chi} r$, then the choice χ is capable of changing the state of narrative from q to r.

Example 2.15. Considering the example narrative, we use the filters and transforms that we presented in Example 2.10 and Example 2.12 to define the following narrative choices. Hereafter, we refer to the set of these choices as $exampleChoices$.

$$\begin{aligned}
joinArmy &= (noAlliance, joinArmyTransform)\\
joinClan &= (noAlliance, joinClanTransform)\\
defeatGeneral &= (canFightArmy, defeatGeneralTransform)\\
defeatPreceptor &= (canFightClan, defeatPreceptorTransform)\\
betrayClan &= (memberOfClan, instituteOrderTransform)\\
betrayArmy &= (memberOfArmy, instituteOrderTransform)\\
instateAnarchy &= (isAnarchyPossible, instateAnarchyTransform)\\
instateTyranny &= (isTyrannyPossible, instateTyrannyTransform)\\
instateFreedom &= (isFreedomPossible, instateFreedomTransform)
\end{aligned}$$

To better understand how narrative choices can connect states of a narrative context, assume $s1 = ((T, none), (T), (T), (none))$ and $s2 = ((T, army), (T), (T), (none))$. We can say that $noAlliance(s1) = T$ and $s2 = joinArmyTransform(s1)$. Therefore, we can write $s1 \xrightarrow{joinArmy} s2$.

2.2.7 Narrative Model

Here, we formally present our definition of a narrative model. Moreover, we describe the characteristics of a *good* model by defining consistency and terminability properties. Besides, we introduce some new notations which we later use when presenting helper operations that facilitate assertion checking.

Definition 2.16. Assuming nc be a narrative context, we say M is a *narrative model* for nc if and only if M is a multiple-entry finite automaton that can be represented by the 4-tuple (S, X, δ, T), where:

- $S \subseteq SPACE(nc)$ is a finite nonempty set of states.
- $X \subseteq CHOICE(nc)$ is a finite nonempty set of choices that act as the alphabet of the automaton.
- $\delta : S \times X \rightarrow S$ is a transition function that drives the automaton's state alternation.
- $T \subseteq S$ is a finite nonempty set of termination states.

Suppose M is a narrative model, and q and r are among its states. Given a choice like χ that connects q to r, we expect the choice to be capable of such a narrative transition. First,

we expect q to satisfy the pre-condition of χ. Also, we expect to obtain r by applying the action of χ on q. If this property holds across all of the model's transitions, we say the model is consistent. Definition 2.17 formally describes consistent narrative models.

Definition 2.17. Assuming $M = (S, X, \delta, T)$ be a narrative model for *nc*, we say M is *consistent* if and only if $\forall((q, \chi), r) \in \delta : q \xrightarrow{\chi} r$.

Definition 2.18. Assume $M = (S, X, \delta, T)$ be a narrative model for *nc* and q and r be members of S. We write $q \overset{M}{\Longrightarrow} r$ (read it as "r is reachable from q in M") if and only if $q = r$ or there is a sequence like $< s_1, \ldots, s_m >$ in S where

$$(s_1 = q) \wedge (s_m = r) \wedge (\forall i \in \{1, \ldots, m-1\} : (\exists \chi \in X : ((s_i, \chi), s_{i+1}) \in \delta))$$

Next, we prove that the reachability of two states in a model is a transitive relation. We will use this property later when we prove that any sub-model of a consistent model is consistent itself (See Section 2.4.1).

Theorem 2.19. *Given that $M = (S, X, \delta, T)$ is a narrative model for nc and $(q, r, s) \in S^3$, if $q \overset{M}{\Longrightarrow} r$ and $r \overset{M}{\Longrightarrow} s$, then $q \overset{M}{\Longrightarrow} s$.*

Proof. We know that $q \overset{M}{\Longrightarrow} r$. Therefore, there is a sequence in S like

$$Q =< q_1, \ldots, q_m >$$

such that

$$(q_1 = q) \wedge (q_m = r) \wedge (\forall i \in \{1, \ldots, m-1\} : (\exists \chi \in X : ((q_i, \chi), q_{i+1}) \in \delta)) \tag{2}$$

Similarly, we know that $r \overset{M}{\Longrightarrow} s$. Therefore, there is a sequence in S like

$$R =< r_1, \ldots, r_n >$$

such that:

$$(r_1 = r) \wedge (r_n = s) \wedge (\forall i \in \{1, \ldots, n-1\} : (\exists \chi \in X : ((r_i, \chi), r_{i+1}) \in \delta)) \tag{3}$$

Considering (2) and (3), we can infer that $q_m = r = r_1$. Consequently, we join Q and R around r and construct a new sequence $T =< t_1, \ldots, t_{m+n-1} >$ where

$$< t_1, \ldots, t_m >=< q_1, \ldots, q_m >$$
$$< t_m, \ldots, t_{m+n-1} >=< r_1, \ldots, r_n >$$

Accordingly, we can say

$$(t_1 = q_1 = q) \wedge (t_{m+n-1} = r_n = s) \tag{4}$$

Now, assume t_i and t_{i+1} be two consecutive members of T. If $i \in 1, \ldots, m-1$ then $t_i = q_i$ and $t_{i+1} = q_{i+1}$. Accordingly, we can refer to (2) and say

$$\forall i \in \{1, \ldots, m-1\} : (\exists \chi \in X : ((t_i, \chi), t_{i+1}) \in \delta) \tag{5}$$

Likewise, if $i \in \{m, m+1, \ldots, m+n-2\}$ then $t_i = r_{i-m+1}$ and $t_{i+1} = r_{i-m+2}$. Accordingly, we can refer to (3) and say

$$\forall i \in \{m, \ldots, m+n-2\} : (\exists \chi \in X : ((t_i, \chi), t_{i+1}) \in \delta) \tag{6}$$

If we put (4), (5), and (6) together, we can infer that

$$\begin{aligned} &(t_1 = q) \wedge \\ &(t_{m+n-1} = s) \wedge \\ &(\forall i \in \{1, \ldots, m+n-2\} : (\exists \chi \in X : ((t_i, \chi), t_{i+1}) \in \delta)) \end{aligned} \tag{7}$$

By definition, (7) implies that $q \overset{M}{\Longrightarrow} s$. Therefore, the proof is complete. □

Definition 2.20. Assuming $M = (S, X, \delta, T)$ be a narrative model for *nc*, we say M is *terminable* if and only if $\forall s \in S : (\exists t \in T : s \overset{M}{\Longrightarrow} t)$.

Definition 2.21. Assuming $M = (S, X, \delta, T)$ be a narrative model for *nc*, we say the 3-tuple (q, χ, r) is a *narrative event* in M if and only if $((q, \chi), r) \in \delta$. Although this definition strongly resembles the definition of transition functions, we noted that this 3-tuple format better represents the concept of narrative events.

Additionally, given that $ev = (q, \chi, r)$ is an event in M, we say:

- $PRE(ev) = q$ is the *pre-state* of ev.
- $DRIVER(ev) = \chi$ is the *driver* of ev.
- $POST(ev) = r$ is the *post-state* of ev.

Definition 2.22. Assuming $M = (S, X, \delta, T)$ be a narrative model for *nc*, we say the ordered pair $(states, choices)$ is a *narrative path* in M if and only if:

- *states* is a sequence in S like $< s_1, \ldots, s_n >$ where $n > 1$.
- *choices* is a sequence in X like $< c_1, \ldots, c_{n-1} >$.
- $\forall i \in \{1, \ldots, n-1\} : (s_i, c_i, s_{i+1})$ is an event in M

In other words, a narrative path is a chain of narrative events.

Example 2.23. Considering the definition of *exampleCntxt* from Example 2.7, we present the following narrative states:

$$
\begin{aligned}
s_1 &= ((T, none), (T), (T), (none)) \\
s_2 &= ((T, army), (T), (T), (none)) \\
s_3 &= ((T, army), (F), (T), (none)) \\
s_4 &= ((T, clan), (T), (T), (none)) \\
s_5 &= ((T, clan), (T), (F), (none)) \\
s_6 &= ((T, order), (T), (T), (none)) \\
s_7 &= ((T, order), (F), (T), (none)) \\
s_8 &= ((T, order), (T), (F), (none)) \\
s_9 &= ((T, order), (F), (F), (none)) \\
s_{10} &= ((T, army), (F), (T), (tyranny)) \\
s_{11} &= ((T, clan), (T), (F), (anarchy)) \\
s_{12} &= ((T, order), (F), (T), (tyranny)) \\
s_{13} &= ((T, order), (T), (F), (anarchy)) \\
s_{14} &= ((T, order), (F), (F), (anarchy)) \\
s_{15} &= ((T, order), (F), (F), (tyranny)) \\
s_{16} &= ((T, order), (F), (F), (freedom))
\end{aligned}
$$

Subsequently, we define the following sets of states:

$$
\begin{aligned}
exampleStates &= \{s_1, \ldots, s_{16}\} \\
exampleTerminationStates &= \{s_{10}, \ldots, s_{16}\}
\end{aligned}
$$

Also, considering the *exampleChoices* from Example 2.15, we define an example transition function as follows:

$$exampleTransFunc : exampleStates \times exampleChoices \rightarrow exampleStates$$

Consequently, for any (s, c) in the domain of *exampleTransFunc*, we define the value of $exampleTransFunc(s, c)$ as it is presented in Table 1. In this table, rows represent members of *exampleChoices* and columns represent members of *exampleStates*.

Now, we can define the following narrative model for the example narrative.

$$
\begin{aligned}
exampleModel = (&exampleStates, \\
&exampleChoices, \\
&exampleTransFunc, \\
&exampleTerminationStates)
\end{aligned}
$$

Table 1: Definition of *exampleTransFunc* function

	s_1	s_2	s_3	s_4	s_5	s_6	s_7	s_8	s_9	s_{10}	s_{11}	s_{12}	s_{13}	s_{14}	s_{15}	s_{16}
joinArmy	s_2	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–
joinClan	s_4	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–
defeatGeneral	–	–	–	s_5	–	s_8	s_9	–	–	–	–	–	–	–	–	–
defeatPreceptor	–	s_3	–	–	–	s_7	–	s_9	–	–	–	–	–	–	–	–
betrayClan	–	–	–	s_6	s_8	–	–	–	–	–	–	–	–	–	–	–
betrayArmy	–	s_6	s_7	–	–	–	–	–	–	–	–	–	–	–	–	–
instateAnarchy	–	–	–	–	s_{11}	–	–	s_{13}	s_{14}	–	–	–	–	–	–	–
instateTyranny	–	–	s_{10}	–	–	–	s_{12}	–	s_{15}	–	–	–	–	–	–	–
instateFreedom	–	–	–	–	–	–	–	–	s_{16}	–	–	–	–	–	–	–

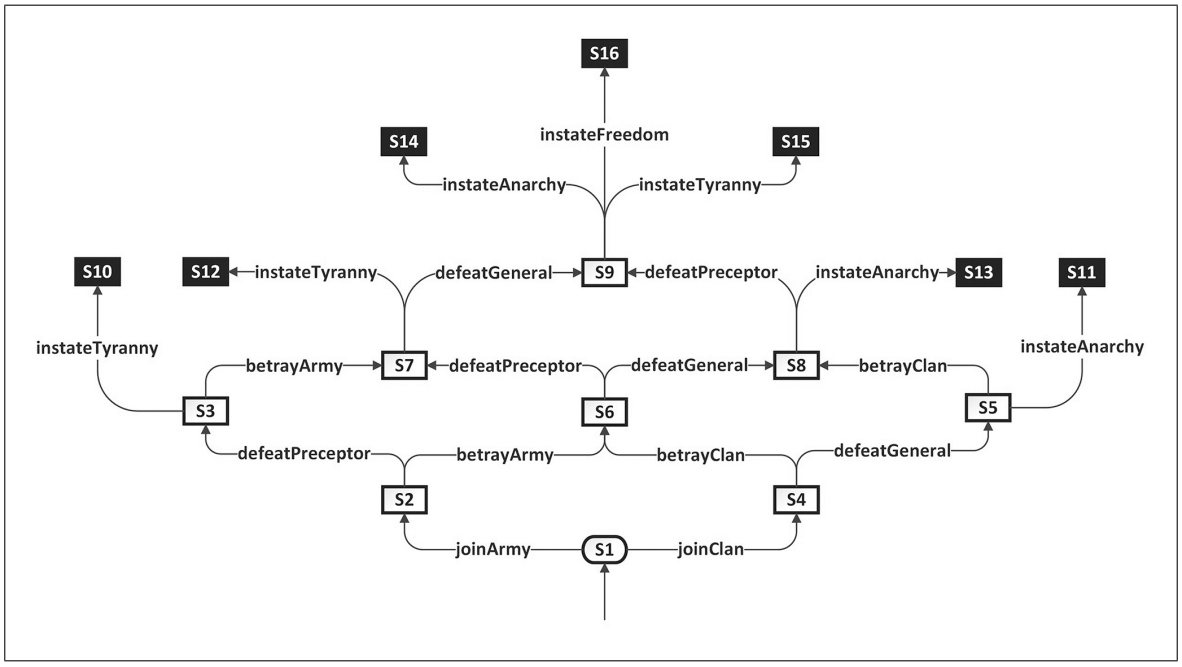

Figure 3: The diagram of the *exampleModel* as a state machine

Figure 3 depicts this model as a state machine.

As can be seen in this figure, we chose s_1 to be the entry point of the state machine based on the implications of the example narrative. Moreover, termination states (members of *exampleTerminationStates*), are placed inside filled boxes, where non-termination states are placed in bordered boxes. By analyzing the diagram of the presented model, we can infer that this model is terminable because for every non-termination state there is a path to a termination state. Also, this model is consistent because for every edge in its corresponding diagram the starting state satisfies the pre-condition of the labeled choice and the destination

state is the result of applying the action of the labeled choice on the starting state.

Designing states and transitions might become exhausting for sophisticated narratives. Therefore, manually designing a narrative model can turn into a very complicated task. To avoid such a drawback, we present an algorithm that automatically handles the generation of narrative states and transitions. This algorithm takes a set of initial states, a set of choices, and a termination condition. The output of this algorithm is a complete narrative model. The following section provides more details about this algorithm.

2.3 Model Generation

As explained before, any narrative model is built upon the definition of a narrative context. Provided that *nc* is a narrative context, we present the GenerateNarrativeModel algorithm for creating consistent narrative models. This algorithm relies on a given narrative context along with a set of initial states and a set of choices. For every initial state, the algorithm examines every choice and checks if that state satisfies the pre-condition of that choice. If so, the action of that choice is applied to that state to create a new state. The new state is added to the output model along with a new transition. This process is recursively executed for the newly created states until either it reaches a state that satisfies the termination filter or the depth of recursion exceeds the *recursionDepth* or there is no applicable choice.

To simplify the recursive design of this algorithm, we present another algorithm called RecursivlyGenerateModel which will be used as a module inside the GenerateNarrativeModel. This algorithm takes over the recursive portion of the mentioned process. Algorithm 1 presents the pseudocode of RecursivlyGenerateModel.

In this algorithm, line 1 checks if the recursion has not reached its maximum depth. Then, for each choice, it is checked whether the related pre-condition is satisfied for the given state. If so, a new state is generated by applying the choice's action to the given state. Lines 5 and 6 update the model's set of choices and the model's transition function. Then, it is checked if the new state has not been met before. If so, the model's set of states is updated. Later, the same procedure is recursively run for the new state if it is not a termination one.

We used the *recursionDepth* parameter to put an upper limit on the number of recursive calls. For a carefully designed termination condition, the algorithm might return even if there was no limitation on the depth of recursion. However, it is possible to design poor termination conditions that lead to infinite recursive calls and infinite non-terminating states. Therefore, the *recursionDepth* parameter ensures the termination of the designed algorithms and guarantees the finiteness of the generated model.

Now that we have introduced RecursivlyGenerateModel, we can implement GenerateNarrativeModel for generating consistent narrative models. Algorithm 2 presents the pseudocode of GenerateNarrativeModel.

In this algorithm, lines 1 to 5 initialize the components of the output narrative model. In

Algorithm 1 RecursivlyGenerateModel

Inputs:

- $state \in SPACE(nc)$
- $choiceSet \subseteq CHOICE(nc)$ is a finite nonempty set of choices
- $termCondition$ is a narrative filter for nc
- $maxDepth$ is a positive integer
- $currentDepth$ is a positive integer
- S is a reference to the model's set of states
- X is a reference to the model's set of choices
- δ is a reference to the model's transition function

Result:

- S is updated to contain every reachable state from $state$.
- X is updated to contain every possible choice from $state$.
- δ is updated to contain every possible transition from $state$.

```
1:  if currentDepth < maxDepth then
2:      for choice ∈ choiceSet do
3:          if precondition_choice(state) = TRUE then
4:              newState ← action_choice(state)
5:              X ← X ∪ {choice}
6:              δ ← δ ∪ {((state, choice), newState)}
7:              if newState ∉ S then
8:                  S ← S ∪ {newState}
9:                  if termCondition(newState) = FALSE then
                        RecursivlyGenerateModel(newState,
                                                choiceSet,
                                                termCondition,
10:                                             maxDepth,
                                                currentDepth + 1,
                                                S, X, δ)
11: return
```

lines 6 and 7 RecursivlyGenerateModel is called for every given initial state. Then, lines 9 to 11 check all of the model's states to find the termination ones. Finally, line 14 assembles the components to return the model.

Also, it is worth mentioning that these algorithms do not necessarily generate terminable narrative models. Therefore, it is imperative to later check the generated models to make

Algorithm 2 GENERATENARRATIVEMODEL

Inputs:

- $Init \subseteq SPACE(nc)$ is a finite nonempty set of initial states.
- $choiceSet \subseteq CHOICE(nc)$ is a finite nonempty set of choices.
- $termCondition$ is a narrative filter for nc.
- $recursionDepth$ is a positive integer.

Result:

- The narrative model (S, X, δ, T)

1: $Init \leftarrow Init - PASS(termCondition)$
2: $S \leftarrow Init$
3: $X \leftarrow \varnothing$
4: $\delta \leftarrow \varnothing$
5: $T \leftarrow \varnothing$
6: **for** $i \in Init$ **do**
RECURSIVLYGENERATEMODEL(i,
$choiceSet$,
7: $termCondition$,
$recursionDepth$, 1,
S, X, δ)
8: **for** $state \in S$ **do**
9: **if** $termCondition(state)$ = TRUE **then**
10: $T \leftarrow T \cup \{state\}$
11: **return** (S, X, δ, T)

sure they are terminable. To this end, the Python package provides means to automatically check the terminability of any given model (See Section 3).

Example 2.24. In this example, we demonstrate how the GENERATENARRATIVEMODEL algorithm can be used to generate the model that we presented in Example 2.23.

First, we define a set of initial states. For this example, this set only contains s_1 which we previously defined in Example 2.23.

$$initial = \{((T, none), (T), (T), (none))\}$$

Also, we need a filter to act as the termination condition. We define this filter as

$$termination(s) = \begin{cases} T, & s[world][status] \neq none \\ F, & otherwise \end{cases}$$

Now, considering *exampleChoices* that we defined in Example 2.15, we can run the GENERATENARRATIVEMODEL algorithm with the following inputs to get the model that was depicted in Figure 3.

$$\text{GENERATENARRATIVEMODEL}(initial, exampleChoices, termination, \infty)$$

We set *recursionDepth* to infinity because we defined other inputs in such a way that the termination of the algorithm is assured. If one expects the algorithm to terminate with some given inputs, we suggest initially setting *recursionDepth* to very big values. If the algorithm took too long to respond, then it is worth trying smaller values and checking if the algorithm sticks in never-ending recursive calls.

2.4 Assertion Checking

To ease the design of narrative assertions, we present some helper operations. These operations can be chained together to construct different types of assertions. Given that *nc* is a narrative context, we define these operations throughout the rest of this section.

2.4.1 subModelFrom

We are going to start by introducing the *subModelFrom* operation. This operation takes two inputs. The first one is a narrative model. The second input is a subset of the states of the given model and is called the pruning set. The output of this operation is a new model that only includes states that are either members of the pruning set or reachable from at least one of its members. In other words, the resulting model will be a sub-model of the original model. This operation is particularly useful when the user wants to check assertions in certain segments of a narrative model. Formally, the inputs of *subModelFrom* are the following:

- $M = (S, X, \delta, T)$ is a narrative model for nc.
- $prunStates \subseteq S$ is a set of states in M.

The output of this operation is a narrative model $M' = (S', X, \delta', T')$ where:

- $S' = \{s \in S \mid \exists p \in prunStates : p \overset{M}{\Longrightarrow} s\}$
- $\delta' = \{((q, \chi), r) \in \delta \mid (q \in S') \wedge (r \in S')\}$
- $T' = T \cap S' = \{t \in T \mid \exists p \in prunStates : p \overset{M}{\Longrightarrow} t\}$

Previously, we mentioned the importance of consistency and terminability for a good narrative model. Therefore, it is worth proving that *subModelFrom* preserves these two properties if a consistent and terminable model is given. The following two theorems address this matter.

Theorem 2.25. *Given a consistent model $M = (S, X, \delta, T)$ and $prunStates \subseteq S$, the result of $subModelFrom(M, prunStates)$ is a consistent narrative model.*

Proof. Let's assume $(S', X, \delta', T') = subModelFrom(M, prunStates)$. We know M is consistent; Therefore, for any member of δ like $((q, \chi), r)$, we can say $q \xrightarrow{\chi} r$. Also, we know that any member of δ' is also a member of δ by definition. Consequently, we infer that

$$\forall((q, \chi), r) \in \delta' : q \xrightarrow{\chi} r$$

and complete the proof. □

Theorem 2.26. *Given a terminable model $M = (S, X, \delta, T)$ and $prunStates \subseteq S$, the result of $subModelFrom(M, prunStates)$ is a terminable narrative model.*

Proof. Let's assume $M' = (S', X, \delta', T') = subModelFrom(M, prunStates)$. Considering the definition of a terminable narrative model, we have to prove

$$\forall s \in S' : (\exists t \in T' : s \overset{M'}{\Longrightarrow} t)$$

To this end, we assume s be an arbitrary member of S' and show that there is a $t \in T'$ such that $s \overset{M'}{\Longrightarrow} t$. According to its definition, S' is a subset of S which implies that $s \in S$. Moreover, we know M is terminable. Thus, there is a $t \in T$ such that $s \overset{M}{\Longrightarrow} t$. Also, we assumed $s \in S'$; Therefore, there is a $p \in prunStates$ such that $p \overset{M}{\Longrightarrow} s$. Knowing that $p \overset{M}{\Longrightarrow} s$ and $s \overset{M}{\Longrightarrow} t$, we can infer that $p \overset{M}{\Longrightarrow} t$ which implies that $t \in T'$. Now that we showed $t \in T'$, demonstrating that $s \overset{M'}{\Longrightarrow} t$ would complete the proof.

We know that $s \overset{M}{\Longrightarrow} t$. This implies that there is a sequence like $Q =< s_1, \ldots, s_m >$ in S where

$$(s_1 = s) \wedge (s_m = t) \wedge (\forall i \in \{1, \ldots, m-1\} : (\exists \chi \in X : ((s_i, \chi), s_{i+1}) \in \delta)) \tag{8}$$

Let us assume s_n be any member of Q. Considering the sequence $< s_1, \ldots, s_n >$ and the fact that it is a subsequence of Q, we can say $s \overset{M}{\Longrightarrow} s_n$ by referring to (8) and implying that

$$(s_1 = s) \wedge (s_n = s_n) \wedge (\forall i \in \{1, \ldots, n-1\} : (\exists \chi \in X : ((s_i, \chi), s_{i+1}) \in \delta))$$

Knowing that $p \overset{M}{\Longrightarrow} s$ and $s \overset{M}{\Longrightarrow} s_n$, we can infer that $p \overset{M}{\Longrightarrow} s_n$. Therefore, we can say $s_n \in S'$ and infer that Q is a sequence in S'.

Considering (8), we know that for any two consecutive members of Q like s_i and s_{i+1}, there is a choice like $\chi \in X$ such that $((s_i, \chi), s_{i+1}) \in \delta$. Also, we showed that Q is a sequence in S' which implies that both s_i and s_{i+1} are members of S'. Accordingly, we can refer to the definition of δ' and say $((s_i, \chi), s_{i+1}) \in \delta'$. Consequently, $Q =< s_1, \ldots, s_m >$ is a sequence in S' where:

$$(s_1 = s) \wedge (s_m = t) \wedge (\forall i \in \{1, \ldots, m-1\} : (\exists \chi \in X : ((s_i, \chi), s_{i+1}) \in \delta')) \tag{9}$$

By definition, (9) implies that $s \overset{M'}{\Longrightarrow} t$. Hence, the proof is complete. □

2.4.2 eventsIn

This operation takes a narrative model as input and extracts all of its narrative events. Accordingly, assuming $M = (S, X, \delta, T)$ be a narrative model, this operation is defined as follows:

$$eventsIn(M) = \{(q, \chi, r) \mid ((q, \chi), r) \in \delta\}$$

2.4.3 preStatesOf

This operation takes a set of narrative events as input and returns a set of narrative states that consists of pre-states of the given events. Formally, this operation is defined as follows:

$$preStatesOf(\{ev_1, \ldots, ev_n\}) = \bigcup_{i=1}^{n} \{PRE(ev_i)\}$$

2.4.4 postStatesOf

This operation takes a set of narrative events as input and returns a set of narrative states that consists of post-states of the given events. Formally, this operation is defined as follows:

$$postStatesOf(\{ev_1, \ldots, ev_n\}) = \bigcup_{i=1}^{n} \{POST(ev_i)\}$$

2.4.5 statesOf

This operation takes two different types of inputs. However, it always returns a set of narrative states. If the input is the narrative model $M = (S, X, \delta, T)$, then this operation simply returns S. Alternatively, if the input is a set of narrative events like $\{ev_1, \ldots, ev_n\}$, then the output of this operation is defined as follows:

$$statesOf(\{ev_1, \ldots, ev_n\}) = \bigcup_{i=1}^{n} \{PRE(ev_1), POST(ev_i)\}$$

2.4.6 choicesOf

While this operation takes two different types of inputs, it always returns a set of narrative choices. If the input is a set of narrative events like $\{ev_1, \ldots, ev_n\}$, then the output is defined as follows:

$$choicesOf(\{ev_1, \ldots, ev_n\}) = \bigcup_{i=1}^{n} \{DRIVER(ev_i)\}$$

Alternatively, if the input is a narrative model like M, the output of this operation is defined as:

$$choicesOf(M) = choicesOf(eventsIn(M))$$

2.4.7 pathsFromTo

This operator takes three inputs. The first input is a narrative model. The second input is a set of states which we call $fromStates$. Similarly, the last input is a set of states which we call $toStates$. This operation outputs a set consisting of every narrative path in the given model that starts from any member of $fromStates$ and ends in any member of $toStates$.

2.4.8 filterEventsByChoices

This operation takes two inputs. The first input is a set of narrative choices. The other one is a set of narrative events. This operation filters the given events by checking if their driver is among the given choices. Formally, this operation is defined as follows:

$$filterEventsByChoices(X, E) = \{ev \in E \mid DRIVER(ev) \in X\}$$

2.4.9 filterEventsByPreStates

This operation takes two inputs. The first input is a narrative filter. The other one is a set of narrative events. This operation filters the given events by checking if their pre-states pass the given filter. Formally, this operation is defined as follows:

$$filterEventsByPreStates(f, E) = \{ev \in E \mid f(PRE(ev)) = T\}$$

2.4.10 filterEventsByPostStates

This operation takes two inputs. The first input is a narrative filter. The other one is a set of narrative events. This operation filters the given events by checking if their post-states pass the given filter. Formally, this operation is defined as follows:

$$filterEventsByPostStates(f, E) = \{ev \in E \mid f(POST(ev)) = T\}$$

2.4.11 filterStates

This operation takes two inputs. The first one is a set of narrative states, and the second one is a filter function. This operation filters the given states by checking if they pass the provided filter. Formally, this operation is defined as follows:

$$filterStates(states, f) = \{s \in states \mid f(s) = T\}$$

Example 2.27. In this example, we demonstrate how to use some of the above-mentioned helper operations to check the following assertion about the model from Example 2.23.

"*It should be possible to defeat general even if the player chose to join the army at the beginning of his/her journey*".

This assertion wants to check if the player can make a particular choice ($defeatGeneral$) if a specific choice ($joinArmy$) has been made before. To make such an assertion, we start by extracting every event from $exampleModel$ using $eventsIn$ operation.

$$\begin{gathered}
allEvents \\
= \\
eventsIn(exampleModel) \\
= \\
\{(s_1, joinArmy, s_2), (s_1, joinClan, s_4), (s_2, defeatPreceptor, s_3), \\
(s_2, betrayArmy, s_6), (s_3, betrayArmy, s_7), (s_3, instateTyranny, s_{10}), \\
(s_4, defeatGeneral, s_5), (s_4, betrayClan, s_6), (s_5, betrayClan, s_8), \\
(s_5, instateAnarchy, s_{11}), (s_6, defeatGeneral, s_8), (s_6, defeatPreceptor, s_7), \\
(s_7, defeatGeneral, s_9), (s_7, instateTyranny, s_{12}), (s_8, defeatPreceptor, s_9), \\
(s_8, instateAnarchy, s_{13}), (s_9, instateAnarchy, s_{14}), (s_9, instateTyranny, s_{15}), \\
(s_9, instateFreedom, s_{16})\}
\end{gathered}$$

Next, we only keep events that are driven by $joinArmy$ using $filterEventsByChoices$ operation.

$$\begin{gathered}
joinArmyEvents \\
= \\
filterEventsByChoices(\{joinArmy\}, allEvents) \\
= \\
\{(s_1, joinArmy, s_2)\}
\end{gathered}$$

Then, we can get every state that is an immediate result of joining the army. We do this by applying $postStatesOf$ operation on $joinArmyEvents$.

$$resultsOfJoiningArmy = postStatesOf(joinArmyEvents) = s_2$$

Now, we need to analyze parts of $exampleModel$ that are reachable from these states to check if it is still possible to defeat the general. To this end, we use the $subModelFrom$ operation as follows:

$$subModel = subModelFrom(exampleModel, resultsOfJoiningArmy)$$

Figure 4 depicts $subModel$ as a state machine. As can be seen in this figure, $subModel$ only contains parts of the $exampleModel$ that rise from s_2.

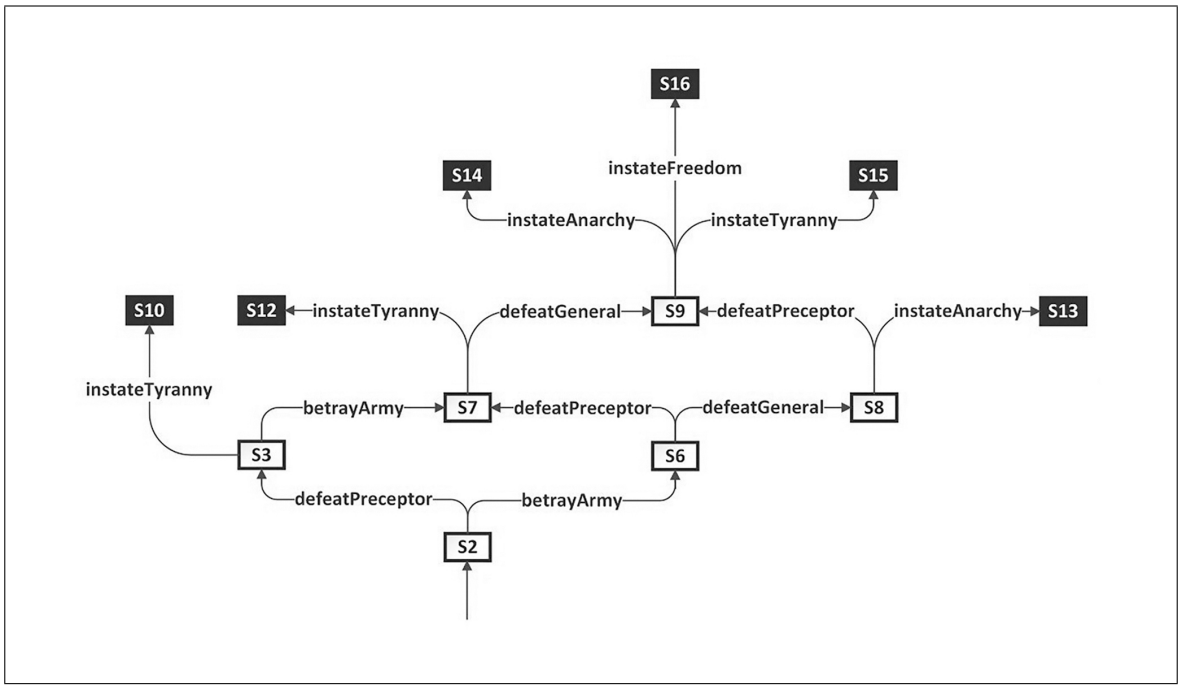

Figure 4: The diagram of the subModel as a state machine

Subsequently, we need to check if $subModel$ includes $defeatGeneral$. To this end, we can use $choicesOf$ operation and check if $defeatGeneral \in choicesOf(subModel)$. Since $defeatGeneral$ is a possible choice in the $subModel$ (see Figure 4), we can conclude that $exampleModel$ satisfies the assertion.

3 Python Package and Case Studies

In this section, we showcase the Python package which implements our framework. We chose Python because of its popularity and easy-to-use syntax. This package allows users to define

narrative contexts, design narrative choices, and generate narrative models. Moreover, it facilitates assertion checking by implementing every helper operation that we mentioned in Section 2.4. Additionally, this package allows users to run the generated narrative models and experience their interactivity.

This section consists of two case studies. The first one implements the example narrative that we modeled in Section 2. The second case study resembles a resource management game. We demonstrate the capabilities of the mentioned Python package throughout these case studies.

For both of these case studies, we start by showing how users can employ our framework to model an interactive narrative. Then, we show how to interactively run the modeled narrative and test its evolution by different user choices. Moreover, we show how to declare assertions and automatically check if they are valid in the specified narrative.

3.1 First Case Study

The interactive narrative which we are going to model in this case study was mentioned at the beginning of Section 2. This narrative has multiple endings and includes several narrative branches that eventually merge together. Therefore, we believe it is a good example to showcase the capabilities of `Ravi`. Previously, we used this narrative to better explain different components of our framework through several examples. Here, we follow the same route and show how those formal examples can be implemented using the provided Python package. We start by importing necessary packages and defining two enumerations of `organization` and `world_status` as shown in Figure 5.

```
from enum import *
from Ravi import *

class organization(Enum):
    none = 1
    order = 2
    army = 3
    clan = 4

class world_status(Enum):
    anarchy = 1
    tyranny = 2
    freedom = 3
    none = 4
```

Figure 5: Defining some enumerations for the first case study

Next, we define some entity classes and add some properties to them. Declaring a variable of type `EntityClass` creates a fresh new class with no properties. Also, we can use the `inherit_from` function to create an entity class by inheriting from a pre-existing one. One can add properties to entity classes using the `addProperty` method. This method takes three

parameters. The first one is the name of the property and should be a string. The second parameter determines the domain of the property. The user can set this parameter to a built-in Python type (bool, int, float, etc.) or a custom type such as a previously-defined enumeration. The third parameter is the default value of this new property. Figure 6 shows a code snippet that defines three classes and adds different properties to each of them.

```
class_character = EntityClass()
class_character.addProperty("IS_ALIVE", bool, True)

class_player = inherits_from(class_character)
class_player.addProperty("ALLIANCE", organization, organization.none)

class_world = EntityClass()
class_world.addProperty("STATUS",world_status,world_status.none)
```

Figure 6: Defining entity classes for the first case study

Now, we use the `NarrativeContext` class to create a new narrative context. Additionally, we use the `addEntity` method to add different entities to the newly created context. The `addEntity` method takes two parameters. The first parameter is the name of the new entity, and the second one determines its class. Figure 7 shows a code snippet that creates a new narrative context and adds four entities to it.

```
context = NarrativeContext()
context.addEntity("PLAYER", class_player)
context.addEntity("PRECEPTOR", class_character)
context.addEntity("GENERAL", class_character)
context.addEntity("WORLD", class_world)
```

Figure 7: Defining the narrative context for the first case study

Next, we define some pre-conditions. These pre-conditions are going to be used when we define narrative choices. As mentioned in Section 2, pre-conditions are filters. In other words, they are functions that take a narrative state and return a boolean value. To define a filter in Python, one should simply declare a function that takes one parameter of type `NarrativeState` and returns a value of type bool. To this end, the `getValue` method of any variable of type `NarrativeState` can be used to read the value of a certain property. This method takes two strings as its inputs. The first parameter should be the name of the desired entity and the second one should be the name of the desired property. Figure 8 shows a code snippet that defines eight pre-conditions.

To define narrative choices, we need both pre-conditions and transforms. Transforms are functions that take one input of type `NarrativeState` and return a value of the same type. To this end, the `setValue` method of any variable of type `NarrativeState` can be used to modify the value of a certain property. This method takes two strings as its inputs. The first parameter should be the name of the desired entity and the second one should be the

```
def no_alliance(s: NarrativeState) -> bool:
    return s.getValue("PLAYER", "ALLIANCE") == organization.none

def member_of_army(s: NarrativeState) -> bool:
    return s.getValue("PLAYER", "ALLIANCE") == organization.army

def member_of_clan(s: NarrativeState) -> bool:
    return s.getValue("PLAYER", "ALLIANCE") == organization.clan

def can_fight_army(s: NarrativeState) -> bool:
    return s.getValue("PLAYER", "ALLIANCE") in {organization.order, organization.clan} and \
           s.getValue("GENERAL", "IS_ALIVE")

def can_fight_clan(s: NarrativeState) -> bool:
    return s.getValue("PLAYER", "ALLIANCE") in {organization.order, organization.army} and \
           s.getValue("PRECEPTOR", "IS_ALIVE")

def is_anarchy_possible(s: NarrativeState) -> bool:
    return s.getValue("PLAYER", "ALLIANCE") in {organization.order, organization.clan} and \
           not s.getValue("GENERAL", "IS_ALIVE")

def is_tyranny_possible(s: NarrativeState) -> bool:
    return s.getValue("PLAYER", "ALLIANCE") in {organization.order, organization.army} and \
           not s.getValue("PRECEPTOR", "IS_ALIVE")

def is_freedom_possible(s: NarrativeState) -> bool:
    return s.getValue("PLAYER", "ALLIANCE") == organization.order and \
           not s.getValue("PRECEPTOR", "IS_ALIVE") and \
           not s.getValue("GENERAL", "IS_ALIVE")
```

Figure 8: Defining pre-conditions for the first case study

name of the desired property. Figure 9 shows a code snippet that defines eight narrative transforms.

Now that we have defined some transforms, we are ready to define narrative choices. To do this, we use the `NarrativeChoice` class. The constructor of this class takes the following three inputs:

1. A pre-condition
2. A narrative transform
3. A user-friendly description

Figure 10 presents a code snippet that defines eight narrative choices. In this figure, the first choice is named `choice_join_army` which consists of `no_alliance` as its pre-condition and `join_army` as its transform. In other words, players can choose to join the army only if they have no alliance. If they make the mentioned choice, the state of the narrative will change according to the definition of `join_army`.

Next, we have to define some filters to determine when the narrative terminates. We referred to these filters as termination conditions in Section 2. Similar to pre-conditions, termination conditions are functions that take a narrative state as their input and return a

```
def join_army(s: NarrativeState) -> NarrativeState:
    s.setValue("PLAYER", "ALLIANCE", organization.army)
    return s

def join_clan(s: NarrativeState) -> NarrativeState:
    s.setValue("PLAYER", "ALLIANCE", organization.clan)
    return s

def establish_order(s: NarrativeState) -> NarrativeState:
    s.setValue("PLAYER", "ALLIANCE", organization.order)
    return s

def defeat_general(s: NarrativeState) -> NarrativeState:
    s.setValue("GENERAL", "IS_ALIVE", False)
    return s

def defeat_preceptor(s: NarrativeState) -> NarrativeState:
    s.setValue("PRECEPTOR", "IS_ALIVE", False)
    return s

def make_anarchy(s: NarrativeState) -> NarrativeState:
    s.setValue("WORLD", "STATUS", world_status.anarchy)
    return s

def make_tyranny(s: NarrativeState) -> NarrativeState:
    s.setValue("WORLD", "STATUS", world_status.tyranny)
    return s

def make_democracy(s: NarrativeState) -> NarrativeState:
    s.setValue("WORLD", "STATUS", world_status.freedom)
    return s
```

Figure 9: Defining transforms for the first case study

boolean value. In this case study, we only define a single termination condition as shown in Figure 11. This termination condition simply checks if the status of the world has a value other than none.

The next step involves defining narrative assertions. We use the `NarrativeAssertion` class. The constructor of this class takes two inputs. The first input should be a user-friendly description for the assertion. The second input should be a function that takes a value of type `NarrativeModel` and returns a boolean value. In general, the body of this function should be implemented using the helper operations that we presented in Section 2.4. Also, we encourage the use of lambda functions in assertion implementation.

Figure 12 shows the definition of three assertions for this case study. The first assertion checks if it is possible to defeat the general even if the player chose to join the army at first. The second assertion checks if it is possible to end with anarchy even if the player chose to betray the clan. Finally, the third assertion checks if it is impossible to defeat the general if the player is still allied with the army.

Now, we are ready to generate a narrative model based on the above-mentioned definitions. First, we have to declare a variable of type `NarrativeSetting`. To declare such a variable, we need four ingredients as follows:

```
choice_join_army = NarrativeChoice(no_alliance, join_army,
                                   "JOIN THE ARMY OF VIRTUE")

choice_join_clan = NarrativeChoice(no_alliance, join_clan,
                                   "JOIN THE CLAN OF CHAOS")

choice_defeat_general = NarrativeChoice(can_fight_army, defeat_general,
                                        "FIGHT ARMY AND DEFEAT THE GENERAL")

choice_defeat_preceptor = NarrativeChoice(can_fight_clan, defeat_preceptor,
                                          "FIGHT CLAN AND DEFEAT THE PRECEPTOR")

choice_betray_clan = NarrativeChoice(member_of_clan, establish_order,
                                     "BETRAY CLAN AND ESTABLISH THE ORDER OF DOUBT")

choice_betray_army = NarrativeChoice(member_of_army, establish_order,
                                     "BETRAY ARMY AND ESTABLISH THE ORDER OF DOUBT")

choice_make_anarchy = NarrativeChoice(is_anarchy_possible, make_anarchy,
                                      "LIGHT UP THE FIRE OF ANARCHY ALL AROUND THE WORLD")

choice_make_tyranny = NarrativeChoice(is_tyranny_possible, make_tyranny,
                                      "CREATE A WORLD-WIDE TYRANNY OF VIRTUE")

choice_make_democracy = NarrativeChoice(is_freedom_possible, make_democracy,
                                        "SAVE HUMANITY FROM BOTH ANARCHY AND TYRANNY")
```

Figure 10: Defining narrative choices for the first case study

```
def terminationCondition(s: NarrativeState) -> bool:
    return s.getValue("WORLD", "STATUS") != world_status.none
```

Figure 11: Defining a termination condition for the first case study

1. A set of initial states
2. A set of termination conditions
3. A list of narrative choices
4. A list of assertions

In this case study, we use the default state of the previously defined context as the sole initial state. Given a narrative setting and an integer as the maximum depth of recursions, we can use the `GenerateNarrativeModel` function to generate a narrative model. The output of this function is of type `NarrativeModel`. Figure 13 demonstrates the process of declaring a `NarrativeSetting` and using it for narrative model generation in Python.

Now, we can use the `validateAssertions` method to see if the model passes the given assertions. Moreover, we can use the `hasAbsoluteTermination` method to check if the generated model is terminable. Figure 14 shows how to use these methods alongside their results. As can be seen in this figure, the model has passed all three assertions and it is reported to be terminable.

```
assertion_1 = NarrativeAssertion(
    "Possible to defeat the general even if the player chose to join the army at first.",
    lambda model:
    choice_defeat_general in choicesOf(
                subModelFrom(
                    postStatesOf(
                        filterEventsByChoice(
                            {choice_join_army},
                            eventsIn(model)
                        )
                    ),
                    model
                )
    )
)

assertion_2 = NarrativeAssertion(
    "Possible to end with anarchy even if the player chose to betray the clan.",
    lambda model:
    len(
        filterStates(
            lambda s: s.getValue("WORLD", "STATUS") == world_status.anarchy,
            statesOf(
                subModelFrom(
                    postStatesOf(
                        filterEventsByChoice(
                            {choice_betray_clan},
                            eventsIn(model)
                        )
                    ),
                    model
                )
            )
        )
    ) != 0
)

assertion_3 = NarrativeAssertion(
    "Impossible to defeat the general if the player is still allied with the army.",
    lambda model:
    len(
        filterStates(
            lambda s: s.getValue("PLAYER", "ALLIANCE") == organization.army,
            statesOf(
                preStatesOf(
                    filterEventsByChoice(
                        {choice_defeat_general},
                        eventsIn(model)
                    )
                )
            )
        )
    ) == 0
)
```

Figure 12: Defining assertions for the first case study

Moreover, we can use the `runNarration` method to run the model and experience its interactivity. This method takes two inputs. The first input should be a boolean value that determines if we want to see the current state in each narrative step. The second input is of type `NarrativeState` and will be used as the entry point to the narrative.

The execution halts when it reaches this method and asks the user to choose from a list

```
initial_states = {NarrativeState(context)}
term_conditions = {terminationCondition}
choices = \
    [
        choice_join_army, choice_join_clan, choice_defeat_general,
        choice_defeat_preceptor, choice_make_anarchy, choice_make_tyranny,
        choice_betray_clan, choice_betray_army, choice_make_democracy
    ]
assertions = [assertion_1, assertion_2, assertion_3]

settings: NarrationSetting = NarrationSetting(initial_states=initial_states,
                                              termination_conditions=term_conditions,
                                              choices=choices,
                                              assertions=assertions)

model: NarrativeModel = GenerateNarrativeModel(setting=settings, max_depth=math.inf)
```

Figure 13: Generating the narrative model for the first case study

```
model.validateAssertions()
print("TERMINABLE:", model.hasAbsoluteTermination())
```

▼

```
=== STARTING ASSERTION CHECKING ===
[1] Possible to defeat the general even if the player chose to join the army at first: PASS
[2] Possible to end with anarchy even if the player chose to betray the clan: PASS
[3] Impossible to defeat the general if the player is still allied with the army: PASS
=== ASSERTION CHECKING COMPLETED ===
TERMINABLE: True
```

Figure 14: Checking assertions for the first case study

of choices. Then, the execution proceeds and changes the current narrative state based on the given input. Subsequently, a new list of choices is shown to the user. Accordingly, the user can start from an arbitrary narrative state and interactively experience the narration. Figure 15 depicts this process and demonstrates how the narration unfolds as the user makes choices.

3.2 Second Case Study

In this case study, we implement a simple resource management game. In resource management games, the user choices are centered around collecting different resources and spending them to obtain other resources in a limited time frame. Designing such a game involves carefully balancing the exchange rate between different resources to obtain the desired level of difficulty and challenge for the player. Consequently, authoring a narrative that embodies this style of interactivity involves special concerns that are meaningless in narratives like the one we presented in the previous case study. Accordingly, we chose this case study to demonstrate the flexibility of `Ravi` and its assertion checking mechanism. Figure 16 presents the

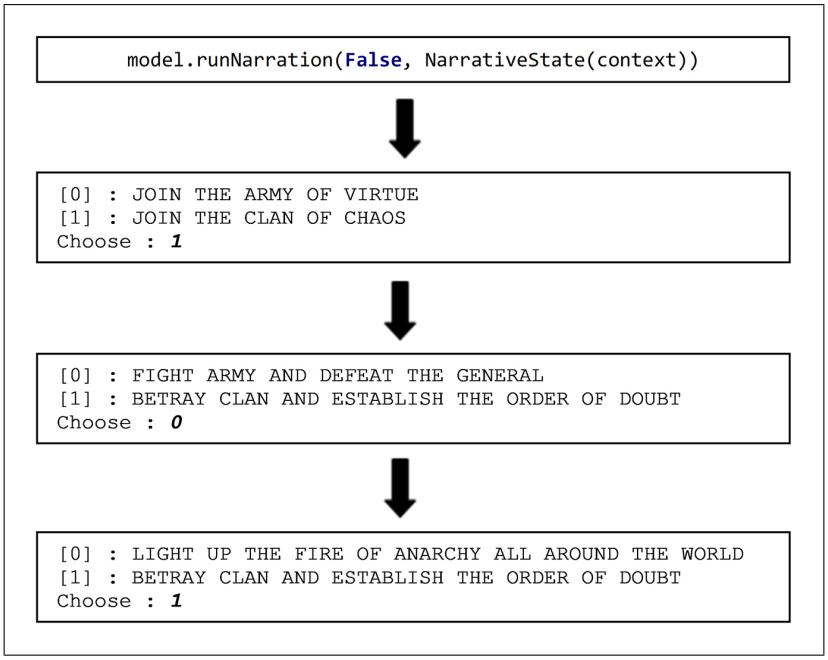

Figure 15: Running the narration for the first case study

narrative design of this case study.

Resource Management

- *The player starts the game in a world where there are 6 trees and 7 mines.*
- *The player can cut trees to increase its wood stockpile. Each tree provides 7 units of wood.*
- *The player can dig mines to collect metal resource. Each mine provides 5 units of metal.*
- *The player can use the collected resources to make bows and swords.*
    1. *Every bow requires 2 units of wood and 1 unit of metal.*
    2. *Every sword requires 1 unit of wood and 2 units of metal.*
- *All of the above-mentioned tasks require a whole day to complete and the player is given only once choice per day.*
- *The goal of this game is to make at least one bow and at least one sword in 5 days or less.*
- *The validity of the following assertions must be checked:*
    1. *The number of wood units should never get negative*
    2. *The number of metal units should never get negative*
    3. *A successful playthrough should be hard but possible. More specifically, the chance of a successful playthrough should be more than 5% and less that 8%.*

Figure 16: An interactive narrative that resembles a resource management game

We start by importing `Ravi` and declaring a set of variables as shown in Figure 17. These variables capture some of the values that were mentioned in the narrative.

```
from Ravi import *

wood_from_every_tree = 7
metal_from_every_mine = 5

wood_for_bow = 2
metal_for_bow = 1
wood_for_sword = 1
metal_for_sword = 2

tree_cutting_duration = 1
mine_digging_duration = 1
bow_making_duration = 1
sword_making_duration = 1
```

Figure 17: Importing `Ravi` and declaring some variables for the second case study

Then, we define two entity classes. One of them encapsulates the player state and the other one encapsulates the world state. We use these classes to define a narrative context for this case study. These steps are shown in Figure 18.

```
class_player = EntityClass()
class_player.addProperty("WOOD", int, 0)
class_player.addProperty("METAL", int, 0)
class_player.addProperty("BOW", int, 0)
class_player.addProperty("SWORD", int, 0)

class_world = EntityClass()
class_world.addProperty("TREE", int, 6)
class_world.addProperty("MINE", int, 4)
class_world.addProperty("REMAINING_DAYS", int, 5)

context = NarrativeContext()
context.addEntity("PLAYER", class_player)
context.addEntity("WORLD", class_world)
```

Figure 18: Defining entity classes and the narrative context for the second case study

Next, we define some pre-conditions to control the possible choices of the player at any given state. We need four pre-conditions for cutting trees, digging mines, making bows, and making swords. Figure 19 presents two of these pre-conditions.

```
def can_cut_tree(s: NarrativeState) -> bool:
    return s.getValue("WORLD", "TREE") > 0 and \
           s.getValue("WORLD", "REMAINING_DAYS") >= tree_cutting_duration

def can_dig_mine(s: NarrativeState) -> bool:
    return s.getValue("WORLD", "MINE") > 0 and \
           s.getValue("WORLD", "REMAINING_DAYS") >= mine_digging_duration
```

Figure 19: Some pre-conditions of the second case study

Now, we define narrative transforms for cutting trees, digging mines, making bows, and

making swords. All of these transforms decrement the remaining days and update the resource quantities. Figure 20 presents two of these transforms.

```
def cut_tree(s: NarrativeState) -> NarrativeState:
    newValue = s.getValue("PLAYER", "WOOD") + wood_from_every_tree
    s.setValue("PLAYER", "WOOD", newValue)

    newValue = s.getValue("WORLD", "REMAINING_DAYS") - tree_cutting_duration
    s.setValue("WORLD", "REMAINING_DAYS", newValue)
    return s

def dig_mine(s: NarrativeState) -> NarrativeState:
    newValue = s.getValue("PLAYER", "METAL") + metal_from_every_mine
    s.setValue("PLAYER", "METAL", newValue)

    newValue = s.getValue("WORLD", "REMAINING_DAYS") - mine_digging_duration
    s.setValue("WORLD", "REMAINING_DAYS", newValue)
    return s
```

Figure 20: Some narrative transforms of the second case study

Subsequently, we can define four narrative choices for cutting trees, digging mines, making bows, and making swords. Figure 21 shows two of these choices.

```
ch_cut_tree = NarrativeChoice(can_cut_tree, cut_tree,
                              "CUT A TREE TO GET +%d WOOD (%d DAYS)" \
                               %(wood_from_every_tree,tree_cutting_duration))

ch_dig_mine = NarrativeChoice(can_dig_mine, dig_mine,
                              "DIG A MINE TO GET +%d METAL (%d DAYS)" \
                              %(metal_from_every_mine,mine_digging_duration))
```

Figure 21: Some narrative choices of the second case study

The next step involves defining two termination conditions. The first one implements the success condition and the other one tells us if the player has lost the game. Figure 22 shows these two termination conditions.

```
def successCondition(s: NarrativeState) -> bool:
    return s.getValue("PLAYER", "BOW") >= 1 and s.getValue("PLAYER", "SWORD") >= 1

def failCondition(s: NarrativeState) -> bool:
    return s.getValue("WORLD", "REMAINING_DAYS") == 0
```

Figure 22: Defining termination conditions for the second case study

Now, we have to implement the required three assertions. Figure 23 shows the implementation of the third assertion which validates the chance of successful playthroughs. To implement this assertion, we used the `pathsFromTo` operator. More precisely, we used this operator to count all of the narrative paths that end in a state satisfying the success condition. Similarly, we counted all of the narrative paths that end in a losing state. Having these

```
assertion_1 = ...

assertion_2 = ...

def calculate_success_Ratio(model: NarrativeModel) -> float:
    successPathCount = len(pathsFromTo(model,
                                       model.initialStates,
                                       filterStates(successCondition, statesOf(model))))

    failPathCount = len(pathsFromTo(model,
                                    model.initialStates,
                                    filterStates(failCondition, statesOf(model))))

    return successPathCount / (successPathCount + failPathCount)


assertion_3 = NarrativeAssertion(
    "5 percent < success ratio < 8 percent",
    lambda m: 0.05 < calculate_success_Ratio(m) < 0.08
)
```

Figure 23: Defining assertions for the second case study

values, we calculated the success ratio of the given model and checked if it was in the proper range.

Finally, we can generate the narrative model as shown in Figure 24.

```
initial_states = {NarrativeState(context)}
term_conditions = {failCondition, successCondition}
choices = [ch_cut_tree, ch_dig_mine,ch_make_bow, ch_make_sword]
assertions = [assertion_1, assertion_2, assertion_3]
settings: NarrationSetting = NarrationSetting(initial_states=initial_states,
                                              termination_conditions=term_conditions,
                                              choices=choices,
                                              assertions=assertions)

model: NarrativeModel = GenerateNarrativeModel(setting=settings, max_depth=math.inf)
```

Figure 24: Generating a narrative model for the second case study

At this point, we can write more code to check the implemented assertions or print other useful data about the modeled narrative. Figure 25 presents the results of such code.

As can be seen in Figure 25, the modeled does not pass the third assertion. Accordingly, the author has to redesign the narrative so this assertion could be passed. Considering the values that were mentioned in Figure 17, if we change the value of `wood_from_every_tree` from 7 to 2 and change the value of `metal_from_every_mine` from 5 to 3, then the resulting model would pass all three assertions. Figure 26 shows how these new values change the success ratio and put it in the desired range for the third assertion.

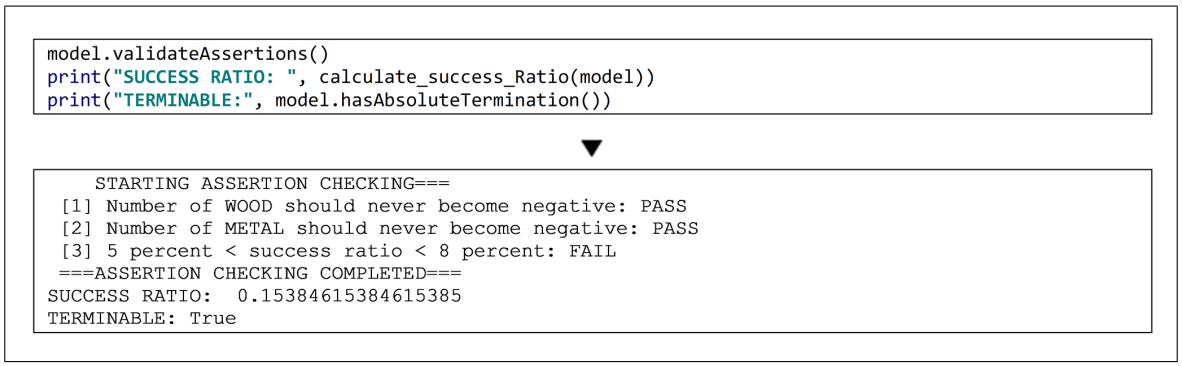

Figure 25: Checking assertions for the second case study

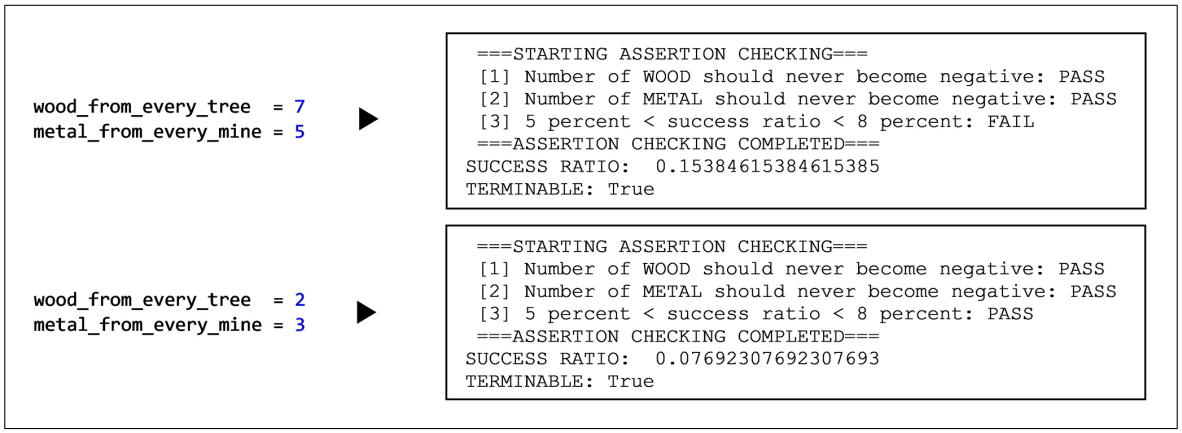

Figure 26: Different assertion checking results for different design values