Proceedings of the Fourteenth Artificial Intelligence and
Interactive Digital Entertainment Conference (AIIDE 2018)

Mimisbrunnur: AI-Assisted Authoring for Interactive Storytelling

Ingibergur Sindri Stefnisson, David Thue
School of Computer Science
Reykjavik University
Reykjavik 101, Iceland

Abstract

Authoring in the context of Interactive Storytelling (IS) is inherently difficult, and there is a need for authoring tools that both enable and assist authors in the creation of new content. In this paper, we discuss our approach for creating an AI-assisted authoring tool via the concept of mixed-initiative systems. We introduce our tool, Mimisbrunnur, which uses this concept to assist authors in the creation of story content. We explain how the tool functions and introduce its fundamental components, including Natural Language Processing, a Suggestion Generator, and three authoring modules.

1 Introduction

Interactive stories (IS) are stories where the audience can interact and have an effect on what happens. Versions of these stories can be seen in such formats as *Choose Your Own Adventure* books, or story-based video games such as *The Wolf Among Us* or *Life is Strange* (Telltale Games 2013; Dontnod Entertainment 2015). Authoring interactive stories can be a long and arduous process. For the audience to have interesting choices and a reliable sense of agency, a large amount of content needs to be produced, some of which might never even be seen by most of the audience of the resulting interactive story. For example, during the creation of *Façade* (Mateas and Stern 2003), multiple person-years of authoring were needed to produce roughly 20 minutes of content that can be replayed with novelty only six or seven times. Despite extensive work in the context of authoring for IS (Louchart et al. 2008; Spierling and Szilas 2009), easing the authorial burden remains a challenging task.

Medler and Magerko (2006) gave requirements that a good authoring tool needs to have; they included generality, debugging capabilities, usability, environment representation, and the ability to specify pacing, timing, and scope. While meeting these requirements seem likely to make it *possible* to author using a given tool, the challenge of creating a large amount of content would still remain. The creation of content itself is one of the most challenging parts of authoring, and is an issue that needs to be addressed. In this paper, we present a system that we designed to tackle this challenge. The system can be used by an author to create an outline for a set of possible stories, one of which will be realized by a story generator at runtime (Thue et al. 2016; 2017). An outline includes information such as what the initial state looks like (e.g., entities and relations between them) and what conditions every generated story should satisfy, with respect to how the state of the world changes during the story. These outlines, along with a set of actions, can then be fed to a story generator (Guðmundsson 2017), which generates a partially ordered plan of actions that satisfies the conditions of the outline. We have implemented an authoring tool named Mimisbrunnur that incorporates Artificial Intelligence (AI) technologies in a variety of ways. Specifically, our tool uses natural language processing and mixed-initiative exploration to help the author create an outline, and stories from this outline can be previewed within the tool using the outputs of a story generator that uses AI planning.

Mimisbrunnur has three key modules: an Initial State Editor, an Action Editor, and a Goal Editor. The Initial State Editor is where an author can state facts about the outline's entities that should hold at the beginning of any generated story. The Action Editor is where they can create, view, and change the set of actions that can occur in any story. The Goal Editor is where an author can specify the conditions (*goals*) that they want to be met during every story.

In the following sections, we begin by explaining mixed-initiative systems and offer some background about the storytelling system that we are working with. We then review what others have done in this area. Next, we present our approach in more detail. We then explain our plan for evaluating our tool and discuss the benefits and limitations of our approach. Finally, we offer some conclusions as well as some ideas for the future of this work.

2 Background

Before we can present our authoring tool, some background concepts need to be explained. These include what we mean by "mixed-initiative system" and how our target story generation system operates.

2.1 Mixed-Initiative Systems

We say that a system is *mixed-initiative* when one or more agents work together iteratively (i.e., taking turns) to perform a task in the context of that system; any agent can take

the initiative to decide what should be done next. A mixed-initiative system is a useful way to model the process of multi-agent authoring, since iterative refinement is a common part of many authoring strategies. One powerful example of a mixed-initiative system is Google Search, in which the human agent can start typing in the search box and an AI agent starts filling in ways to complete the human's query. In prior work (Stefnisson and Thue 2017), we discuss this perspective in more detail.

2.2 Storytelling System

Our authoring tool is designed to help create Thue et al. story outlines (2016; 2017). They defined an *outline* as a set of relations, either true or false, that restrict how a story should start (initial conditions), as well as what should happen in the middle of the story (intermediate goals), and then how it should end (final goals). They also define *abstract entities*, which are author-created entities that serve as "roles" that a concrete entity in the "story world" can fill when the story is running. Stories are then created by feeding an outline to an AI planner, which produces a partially-ordered plan of actions that satisfies the outline's constraints. The stories are then playable in the storytelling system; a player can interact with the characters, (e.g.take their items, give them other items) and the storytelling system can react by fixing the stories if the player disrupts a story that was already planned.

To give authors more control over what stories are possible, we give them the opportunity to create the actions that can occur. Actions are planning operators that have preconditions that need to be met and postconditions (effects) that change the world. An outline is authored using an open-world assumption, meaning that all unspecified relations are considered to be undetermined. Here is an example outline:

```
Initial State:
  Hero has heirloom
  Villain does not have heirloom
  treasure is lost
Intermediate Goal 1:
  Villain has heirloom
Final Goal:
  Hero has heirloom
  Hero has treasure
```

We define a few terms: an *entity* has a name and a type (which can be either Character or Item); a *relation* consists of a name, a list of entities, and a template text for an associated natural-language sentence; a *condition* consists of a relation and a Boolean value that specifies whether the relation is true or false; an *effect* consists of a relation and an operator that specifies whether the relation should be added or removed when its associated action is executed; a *goal* consists of a name and a list of conditions; and an *action* consists of a name, a list of conditions, and a list of effects.

3 Related Work

The challenge of creating content has often been approached using the techniques of Procedural Content Generation (PCG) (Yannakakis and Togelius 2011; Togelius et al. 2011; Hendrikx et al. 2013), where content is generated or recombined from a base set of elements. Any outputs of the generator that are undesirable are simply discarded, leaving the author to guess at which new inputs might yield a better output. One exception is *Sentient Sketchbook* (Liapis, Yannakakis, and Togelius 2013), a map editor in which the system gives map suggestions in real time based on genetic algorithms that maximize given fitness parameters.

Another way that generative systems have been used to ease the authoring burden is by directly assisting in the authoring process itself. Examples such as *Say Anything* (Swanson and Gordon 2008; 2012) and *Creative Help* (Roemmele and Gordon 2015) used an AI agent to pick responses to authored text from a corpus of natural language text. *Say Anything* requires users to use a suggestion after every sentence, while *Creative Help* allows authors more control over the suggestions. However, the artifacts of both systems become non-interactive as soon as the authoring stops.

PERSONAGE (Walker et al. 2011) uses an AI agent to assist with styling authored dialogue according to learned character models and author-selected parameters. This work allows relatively little cooperation between the mixed-initiative agents, since each agent (author and stylist) takes only one turn and they act in different spaces.

The authoring tool *ENIGMA* (Kriegel et al. 2007; Kriegel and Aylett 2010) has a mixed-initiative component in which AI characters can give suggestions for the next event and then the author can either accept that event or force a different event to occur. This system has similarities to our authoring tool, but its goal is different from ours; they aimed to simplify the authoring of autonomous character behaviours, while we aim to simplify the authoring of plot.

The authoring tool *Bowman* (Thomas and Young 2006; Thomas 2006) uses mixed-initiative planning in a plot-based system. The author gives a domain and describes goals for the story, and an AI planner produces possible stories. The author can then refine the domain and goals to try to get different stories from the planner. This process is similar to our description of common PCG systems, in which different results can only be obtained by acting in a space (e.g., possible domains) that is distinct from the generator's space (e.g., possible stories). *Bowman* also requires authors to be specifically trained in their STRIPS-like (Fikes and Nilsson 1971) authoring language, while our tool allows authors to write in a restricted natural language.

StoryFramer (Hayton et al. 2017) offers a way to use generate planning models from natural language text. Their natural language processing is stronger than what we present in this work, but their approach does not help in the generation of the stories themselves. Instead, their system takes a fully created plot and uses it to generate a planning model.

To the best of our knowledge, no system has been created that both (i) assists authors with creating a set of possible stories that will be chosen from at runtime and (ii) supports authoring in a restricted natural language.

4 Proposed Approach

With Mimisbrunnur, we aim to support creativity and ease the authorial burden for authors of interactive stories. We have created a tool that (i) allows authors to create outlines

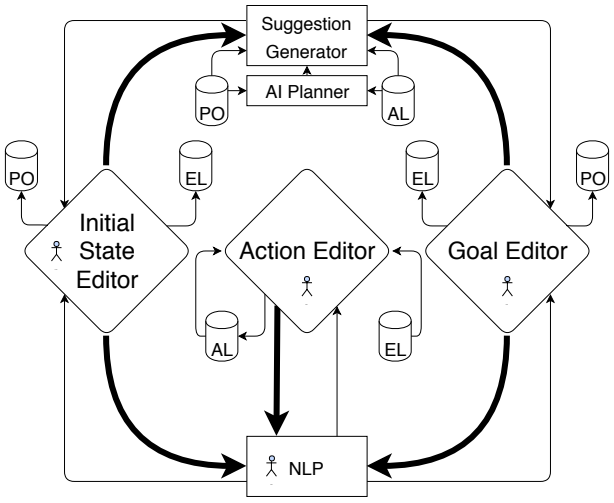

Figure 1: A schematic overview of the tool. Diamonds show modules where authoring occurs, rectangles show internal modules, and cylinders show libraries that are used by the modules (PO = Prototype Outline, EL = Entity Library, and AL = Action Library). The figure appears on modules that an author can interact with. Thick arrows mean that the receiving module observes everything in the source module.

in a restricted natural language, (ii) offers suggestions concerning what conditions they might want in their outlines, and (iii) allows authors to visualize and edit potential stories that their outlines produce.

As shown in Figure 1, our tool offers three modules for authoring: the Initial State Editor, the Action Editor, and the Goal Editor. These modules then interact with a Natural Language Processing system, a Suggestion Generator, and three libraries: an Action Library, an Entity Library, and a Prototype Outline. The Initial State Editor is where an author states the facts of the world in the beginning of the story, including: the context of the story, which entities the author cares about, how the entities relate to one another, and so on. The Action Editor is where an author can either see which actions are available or create new actions if desired. The Goal Editor is where the author can create goals for their outline; they can arrange the goals in the order they want them to happen, and add constraints to the outline and see the possible stories that can come out of the story generator.

We use the Suggestion Generator to generate suggestions for the author. We believe that doing so can help ease the authoring burden in two ways. First, it can produce sentences that an author might already have in mind. Second, it can find, produce, and highlight sentences whose addition to the outline would help generate more varied stories. Our Natural Language Processing module is what allows authors to write restricted natural language sentences, which the tool parses and attempts to understand using the Lexicalized Parser and Part-Of-Speech tagger of Stanford's CoreNLP (Toutanova et al. 2003; Manning et al. 2014). The libraries are databases that store and maintain all actions, entities, and the prototype of the outline that the author is writing. Before we introduce the authoring modules, we will first explain how suggestions are generated and how authored sentences are parsed.

Algorithm 1: Suggestion Generator

Inputs : *canvas*: a list of conditions (as a partially completed goal or initial state)
actions: a list of all actions in the Action Library
entities: a list of all permutations of entities in the Prototype Outline
Outputs: *suggestions*: a list of conditions as suggestions

1 **for** each *permutation* in *entities* **do**
2 **for** each *action* in *actions* **do**
3 rename(*action.entities*, *permutation*)
4 **if** *action.preconditions* are all in *canvas* **or**
5 *action.preconditions* contains a negation of any *canvas* condition **then**
6 continue
7 **else**
8 **if** *canvas* contains all but one of *action.preconditions* **then**
9 *suggestions* $\leftarrow$ *suggestions* $\oplus$ *action's missing precondition*
10 *PS* $\leftarrow \{\}$: an empty list of conditions (poss. suggestions)
11 **for** each *action* in *actions* **do**
12 *PS* $\leftarrow$ *PS* $\oplus$ *action.preconditions*
13 **if** *canvas* is not an initial state **then**
14 *PS* $\leftarrow$ *PS* $\oplus$ *action.effects*
15 **for** each *permutation* in *entities* **do**
16 **if** length(*suggestions*) > 10 **then**
17 break
18 **for** each *condition* in *PS* **do**
19 rename(*condition.entities*, *permutation*)
20 **if** *condition* and $\neg$*condition* are not in *canvas* **then**
21 *suggestions* $\leftarrow$ *suggestions* $\oplus$ *condition*

4.1 Suggestion Generator

Algorithm 1 shows how the tool generates suggestions based on a given *canvas* of conditions. This canvas might describe a partial initial state (in the Initial State Editor) or a partial goal (in the Goal Editor). Lines 1 to 9 search the Action Library for actions whose preconditions are *almost* met by the conditions in the current canvas, toward suggesting that the author add the conditions that are not yet met. Doing so requires generating and iterating over every possible permutation of each action, given the entities in the Prototype Outline (lines 1 to 3). Lines 4 to 5 check the preconditions of the current action permutation. If they are all already in the canvas (line 4), there are none left to suggest. If any of them negate a condition in the canvas (line 5), then the current action is impossible from the initial state or goal that the canvas represents. Line 6 skips every action permutation that meets these criteria. Line 8 checks the remaining action permutations for those which have all but one of their preconditions satisfied by the current canvas, and line 9 adds each such "missing" precondition to the list of suggestions to output. Lines 10 to 21 produce additional suggestions (if necessary) up to an arbitrary threshold of ten suggestions.

Algorithm 2: Natural Language Processing

Inputs : *sentence*: a sentence in restricted natural language
Outputs: *condition* : a condition that represents *sentence*

```
1  removeContractions(sentence)
2  for each word in sentence do
3  |   taggedWord ← tagPenTreebank(word)
4  |   if taggedWord is a proper noun then
5  |   |   entity ← newEntity(word, Character)
6  |   |   condition.entities ← condition.entities ⊕ entity
7  |   else if taggedWord is a singular or plural noun then
8  |   |   entity ← newEntity(word, Item)
9  |   |   condition.entities ← condition.entities ⊕ entity
10 parseTree ←
     getGrammaticalStructure(lexicallyParse(sentence))
11 if parseTree contains negation then
12 |   condition.value ← false
13 else
14 |   condition.value ← true
15 if parseTree.root is part of a nominal subject clause and a
     direct object clause or parseTree.root is part of a passive
     auxillary or a copula then
16 |   condition.name ← parseTree.root
17 else
18 |   condition.name ← unknown
```

First, lines 10 to 14 gather all of the conditions that appear in the actions of the Action Library into a list of possible suggestions. If the canvas represents an initial state, the effects of the actions are excluded, because an effect added to the initial state does not allow for any more possible actions. Next, lines 15 to 21 iterate through all potential permutations of each possible suggestion, given the entities in the Prototype Outline. Until the threshold is met, lines 20 and 21 will add possible suggestion permutations to the suggestions list, provided that neither the candidate condition nor its negation are already in the canvas.

4.2 Natural Language Processing

Algorithm 2 shows how our tool parses an authored sentence into a condition; conditions are the main building blocks of goals, actions, effects, and the initial state. To explain the algorithm, we will use the input sentence "Sam didn't steal cookies from John" as a running example. To start, line 1 removes all detected contractions (e.g., "didn't" becomes "did not"). Lines 2 to 9 examine each word in the sentence to infer what entities should be in the output condition. To do so, line 3 tags each word using the Penn Treebank notation (Marcus, Marcinkiewicz, and Santorini 1993):

```
(Sam NNP) (did VBD) (not RB) (steal VB)
(cookies NNS) (from IN) (John NNP).
```

Based on those tags, proper nouns (NNP) become Characters in the output condition (lines 5 and 6), while singular and plural nouns (NNS) become Items (lines 8 and 9). *Sam* and *John* become characters, and *cookies* becomes an item.

Lines 10 to 18 use the structure of the sentence to infer what the name of the output condition should be. Line 10 parses the sentence using a lexicalized parser and produces a *parse tree* which is then used to process the grammatical structure of the sentence:

```
[nsubj(steal-4, Sam-1), aux(steal-4, did-2),
neg(steal-4, not-3),root(ROOT-0, steal-4),
dobj(steal-4, cookies-5), case(John-7, from-6),
nmod:from(steal-4, John-7)].
```

If the parse tree contains a negation (neg), then the condition's value gets set as false; otherwise, it gets sets as true (lines 11 to 14). If the root word of the parse tree is part of either (i) both a nominal subject clause (nsubj) and a direct object clause (dobj), or (ii) a passive auxilliary (auxpass) or copula (cop), then the name of the output condition is set as the root word (lines 15 to 16). Otherwise, the name is set as unknown (line 18). Our example would then be a condition that the AI planner could use: $steal(sam, cookies, john) = false$, with the template text: "Sam didn't steal cookies from John".

If the algorithm sets the condition's name as unknown, we allow the author to choose what the condition's name should be. Once a condition name has been set, the tool then saves that information for future use. Therefore, if an author writes a new sentence that involves the same condition, lines 10 to 18 of Algorithm 2 can be skipped.

To manage the scope of this project, the natural language that authors are able to write with is restricted in the following ways. (i) Conditions that refer to the same concept must always be written using the same word. For example, if an author has stated that "John has gold", and then wants to state that someone else does not have gold, they must use "not has" instead of "not have" (e.g., "Robin does not has gold"). (ii) Sentences can not contain more than one condition at a time, nor use pronouns; sentences such as "John stole gold from Robin and buried it", would need to be split into two different sentences to be parsed properly. A future system could integrate more complex NLP so that more complicated sentences could be parsed.

4.3 Initial State Editor

Figure 2 shows a screenshot of the Initial State Editor, which has two different areas. The area on the left is a text field where an author can write sentences about the initial state of any abstract entities that are important in their outline. These sentences describe conditions that must be true at the beginning of every story that is generated from the current outline. The area on the right is the suggestion box, which displays sentences generated by the suggestion generator;

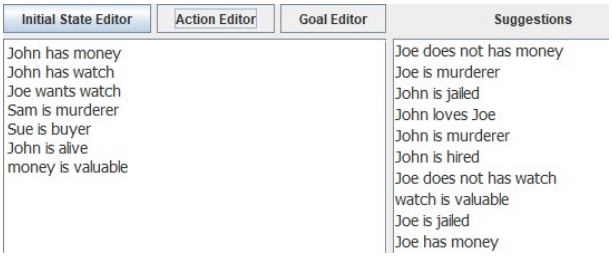

Figure 2: A screenshot of the Initial State Editor.

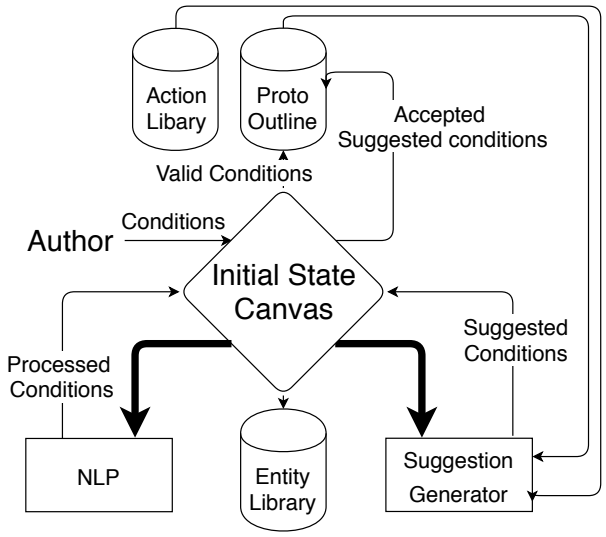

Figure 3: A high level overview of the Initial State Editor.

the author can accept any suggestion by double-clicking on it. Both areas together comprise the *initial state canvas*.

Figure 3 shows how the Initial State Editor connects to the other parts of our tool. An author writes sentences, and each sentence is processed by our Natural Language Processing system to produce a new condition (Section 4.2). For each condition produced, the abstract entities that it describes are stored in an *entity library* (a database of all the entities in the outline), and the condition itself is sent to the initial state canvas. The suggestion generator sees the conditions in the canvas and generates suggested sentences that the author can choose from (Section 4.1). The author can also continue writing new sentences. When the author finishes setting up the initial state, both the authored conditions and the author-accepted, suggested conditions are sent into the Prototype Outline as its initial state. The tool then automatically generates inequality relations for all entities of the same type, under the simplifying assumption that two abstract entities (roles) with different names can never be filled by the same entity in the story world (Thue et al. 2016).

4.4 Action Editor

Figure 4 shows the Action Editor, where authors can see possible actions and create their own. Actions are based on planning operators; they have preconditions that need to be met so that an action can be performed, and effects that then change the world. When an author wants to create a new action, they first provide a name for the action (e.g., "steal" in Figure 4). They then write the preconditions and effects as sentences (using any abstract entities that they wish), and these sentences are then parsed into conditions by our Natural Language Processor. We convert conditions to effects by using the condition's Boolean value to determine whether the effect should be added (value: true), or removed (false).

In general, story actions should be more general than what Figure 4 seems to show; it should be generally possible for

Figure 4: The steal action shown in the Action Editor.

one character to steal an item from another, so long as the specified preconditions hold for whichever entities are involved. To recover the general version of each action, our tool automatically replaces each named entity in an authored action with typed placeholder entities. For example, "John" becomes "Character 1", "Robin" becomes "Character 2", and "gold" becomes "Item 1". It also adds inequality preconditions among placeholders of the same type (e.g., between "Character 1" and "Character 2"), under the assumption that different placeholders should be filled by different entities when the action is instantiated. It also adds an effect that states that the action happened with these entities, which is used for hard constraints as we explain in Section 4.5.

4.5 Goal Editor

Figure 5 shows the Goal Editor, where an author can define what they want to have happen in the story in two ways: by partially ordering a sequence of goals (rectangles in the figure), and by placing hard constraints directly on story actions (ellipses in the figure).

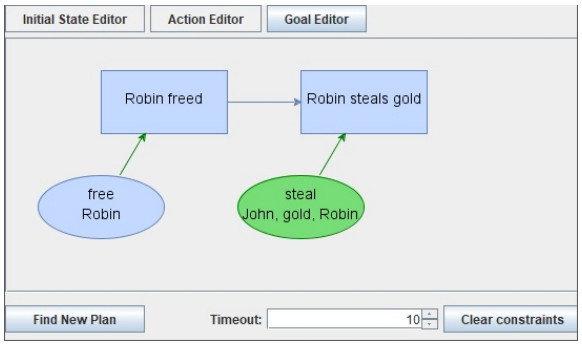

Figure 5: A potential plan shown by the Goal Editor. Ellipses are actions and rectangles are goals. The steal action being green shows that the action is required (a hard constraint).

The author can set up as many goals as they want to have met during the story, and they can connect the goals with arrows to indicate the order in which they should become true in every story that is generated from the outline. Goals are authored similarly to initial state conditions: by writing sentences. The primary difference is that each goal sentence describes a condition that must become true in every generated story at some point *after* it begins. The Suggestion Genera-

tor produces suggestions while goals are authored, using a screen similar to the Initial State Editor.

Hard constraints allow an author to state that a certain instance of an action (i.e., with particular entities filling its placeholders) should always happen in every possible story, or should never happen in any story. To place constraints on story actions, the author must first ask the Suggestion Generator to suggest a plan of actions that meets the goals that are currently on the canvas (via the "Find New Plan" button in Figure 5). To generate this plan, the Suggestion Generator uses an AI Planner (Guðmundsson 2017; Thue et al. 2017). It analyzes the resulting actions to determine which actions satisfy which goals. It then places an ellipse on the editor's canvas for each action, and connects each action to each goal that it satisfies with an arrow from the action's ellipse to the goal's rectangle.

Technically, we implemented hard constraints by automatically generating an extra effect for every action called *internalActionHappened*. This effect includes all the entities in the action. It gets added to the initial state as a false condition for all possible permutations of entities from the Prototype Outline. When an author sets a hard constraint, a corresponding internalActionHappened condition is created inserted into a final goal for the AI planner, with its value being true/false for the action always/never happening.

Figure 6 shows a schematic diagram of the Goal Editor. As an example of how the Goal Editor might be used, consider an outline with an initial state such that *John has gold* and *Robin is jailed*. Suppose that the author wants Robin to somehow get free from jail and then steal the gold from John. One way to do so would be to have two goals set up in order. The first goal could contain the condition *Robin is not jailed*, and the second goal could contain the condition *Robin has gold*. For convenience, authors can name each goal; they might choose *Robin freed* and *Robin steals gold*, respectively (the rectangles in Figure 5). Suppose that there are two actions that could lead to each of these goals. For the first goal, John could set Robin free or Robin could break out of jail. For the second goal, John could give Robin the gold, or Robin could steal the gold from John.

Now the author can run the planner for their goals. Suppose that the output plan is: *John sets Robin free* and then *John gives Robin the gold*. This is a perfectly acceptable story, but not close enough to what the author wanted (Robin did not steal the gold). There are two ways that an author could mend this. One is to change the goals, adding *Robin stole gold from John* as a goal to ensure that the *give* action is not possible. Another way is to add a hard constraint by selecting the action *John gives Robin the gold* in the canvas and stating that it should never occur. In both cases, clicking "Find New Plan" would yield a desired result: either *John sets Robin free* and then *Robin steals the gold from John*, or *Robin breaks free* and then *Robin steals the gold from John*.

5 Discussion and Future Work

We have created a system that supports authoring in a restricted natural language and assists authors with creating a set of possible stories. We believe this will help to ease the authorial burden for creating interactive stories, and we have shown how the modules of the authoring tool interact with one another to assist with the authoring process.

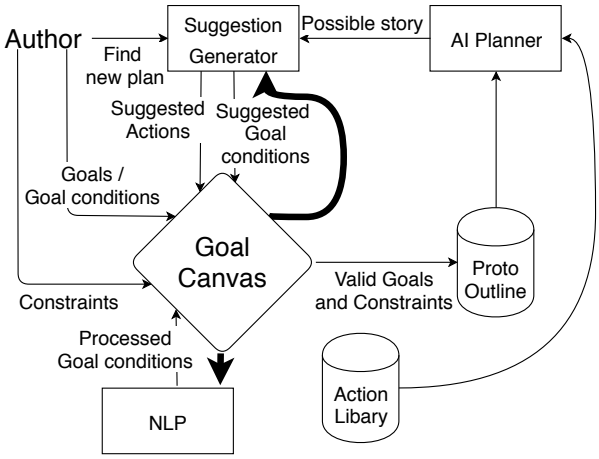

Figure 6: A high level overview of the Goal Editor.

There are some limitations to our approach. Although the natural language processing seems reasonably capable, it can only understand very simple relations. While it is possible for an author to resolve many misunderstandings using a temporary, Wizard-of-Oz interface, there are some words like locations that our parser simply does not handle well. It would be helpful to determine what relations the parser should understand easily (e.g., by working with potential authors). We discussed other limitations related to NLP in Section 4.2. Although our tool aims to ease the authoring burden, an author must still set up the necessary elements of an outline as well as a library of actions.

Other avenues of future work include extending the Suggestion Generator to allow an author to add a score such as *interestingness* to the actions that come out of the planner. The Suggestion Generator could then use plans with higher interestingness to generate its suggestions. Mimisbrunnur is currently "raw"; it is a research tool for creating stories for a specific storytelling system. To make it ready for public use, it would need more generality and further improvements. An evaluation of our tool is currently underway using the Creativity Support Index (CSI) (Cherry and Latulipe 2014), a psychometric survey designed for evaluating a tool's ability to assist a user's creative work.

6 Conclusion

We have presented our work toward easing the authorial burden of generative, interactive storytelling. Our solution has extended prior work by allowing authors to create story outlines using a restricted natural language and with assistance from an AI agent. Our authoring tool puts the author and the AI agent into a mixed-initiative context, where the agent suggests ideas for initial constraints and goal constraints and visualizes possible stories that could result from the authored outline. By using our tool, authors gain actionable insight into how the content they author will manifest as a set of generated stories.

References

Cherry, E., and Latulipe, C. 2014. Quantifying the creativity support of digital tools through the creativity support index. *ACM Transactions on Computer-Human Interaction (TOCHI)* 21(4):21.

Fikes, R. E., and Nilsson, N. J. 1971. STRIPS: A new approach to the application of theorem proving to problem solving. *Artificial intelligence* 2(3-4):189–208.

Guðmundsson, T. T. 2017. Flexible authoring using GOLOG planning in interactive storytelling. Master's dissertation, School of Computer Science, Reykjavik University.

Hayton, T.; Porteous, J.; Ferreira, J.; Lindsay, A.; and Read, J. 2017. Storyframer: From input stories to output planning models. In *ICAPS Workshop on Knowledge Engineering for Planning and Scheduling*, 1–9. ICAPS.

Hendrikx, M.; Meijer, S.; Van Der Velden, J.; and Iosup, A. 2013. Procedural content generation for games: A survey. *ACM Transactions on Multimedia Computing, Communications, and Applications (TOMM)* 9(1):1.

Kriegel, M., and Aylett, R. 2010. Crowd-sourced AI authoring with ENIGMA. In Aylett, R.; Lim, M. Y.; Louchart, S.; Petta, P.; and Riedl, M., eds., *Interactive Storytelling*, 275–278. Springer Berlin Heidelberg.

Kriegel, M.; Aylett, R.; Dias, J.; and Paiva, A. 2007. An authoring tool for an emergent narrative storytelling system. In *AAAI Fall Symposium on Intelligent Narrative Technologies*, 55–62. AAAI Press.

Liapis, A.; Yannakakis, G. N.; and Togelius, J. 2013. Sentient sketchbook: Computer-aided game level authoring. In *Proceedings of the 8th International Conference on Foundations of Digital Games*, 213–220. Society for the Advancement of the Science of Digital Games.

Louchart, S.; Swartjes, I.; Kriegel, M.; and Aylett, R. 2008. Purposeful authoring for emergent narrative. In Spierling, U., and Szilas, N., eds., *Interactive Storytelling*, 273–284. Springer Berlin Heidelberg.

Manning, C. D.; Surdeanu, M.; Bauer, J.; Finkel, J.; Bethard, S. J.; and McClosky, D. 2014. The Stanford CoreNLP natural language processing toolkit. In *Association for Computational Linguistics: System Demonstrations*, 55–60. ACL.

Marcus, M. P.; Marcinkiewicz, M. A.; and Santorini, B. 1993. Building a large annotated corpus of english: The penn treebank. *Computational linguistics* 19(2):313–330.

Mateas, M., and Stern, A. 2003. Façade: An experiment in building a fully-realized interactive drama. *Game Developers Conference, Game Design track*.

Dontnod Entertainment. 2015. Life Is Strange. https://www.lifeisstrange.com/.

Telltale Games. 2013. The Wolf Among Us. https://telltale.com/series/the-wolf-among-us/.

Medler, B., and Magerko, B. 2006. Scribe: A tool for authoring event driven interactive drama. In Göbel, S.; Malkewitz, R.; and Iurgel, I., eds., *Technologies for Interactive Digital Storytelling and Entertainment*, 139–150. Springer Berlin Heidelberg.

Roemmele, M., and Gordon, A. S. 2015. Creative help: A story writing assistant. In Schoenau-Fog, H.; Bruni, L. E.; Louchart, S.; and Baceviciute, S., eds., *Interactive Storytelling*, 81–92. Springer International Publishing.

Spierling, U., and Szilas, N. 2009. Authoring issues beyond tools. In Iurgel, I. A.; Zagalo, N.; and Petta, P., eds., *Interactive Storytelling*, 50–61. Springer Berlin Heidelberg.

Stefnisson, I., and Thue, D. 2017. Authoring tools should be mixed-initiative systems. In *Proceedings of the ICIDS 2017 Workshop on Authoring for Interactive Storytelling*. Online: https://narrativeandplay.org/ais/proceedings.html.

Swanson, R., and Gordon, A. S. 2008. Say anything: A massively collaborative open domain story writing companion. In Spierling, U., and Szilas, N., eds., *Interactive Storytelling*, 32–40. Springer Berlin Heidelberg.

Swanson, R., and Gordon, A. S. 2012. Say anything: Using textual case-based reasoning to enable open-domain interactive storytelling. *ACM Transactions on Interactive Intelligent Systems (TiiS)* 2(3):16.

Thomas, J. M., and Young, R. M. 2006. Author in the loop: Using mixed-initiative planning to improve interactive narrative. In *Proceedings of the ICAPS-06 Workshop on Preferences And Soft Constraints for Planning*, 21–30. ICAPS.

Thomas, J. M. 2006. Collaborative authoring of plan-based interactive narrative. In *Proceedings of the ICAPS-06 Doctorial Consortium*, 127–130. ICAPS.

Thue, D.; Schiffel, S.; Árnason, R. A.; Stefnisson, I. S.; and Steinarsson, B. 2016. Delayed roles with authorable continuity in plan-based interactive storytelling. In Nack, F., and Gordon, A. S., eds., *Interactive Storytelling*, 258–269. Springer International Publishing.

Thue, D.; Schiffel, S.; Guðmundsson, T. Þ.; Kristjánsson, G. F.; Eiríksson, K.; and Björnsson, M. V. 2017. Open world story generation for increased expressive range. In Nunes, N.; Oakley, I.; and Nisi, V., eds., *Interactive Storytelling*, 313–316. Springer International Publishing.

Togelius, J.; Yannakakis, G. N.; Stanley, K. O.; and Browne, C. 2011. Search-based procedural content generation: A taxonomy and survey. *IEEE Transactions on Computational Intelligence and AI in Games* 3(3):172–186.

Toutanova, K.; Klein, D.; Manning, C. D.; and Singer, Y. 2003. Feature-rich part-of-speech tagging with a cyclic dependency network. In *Proceedings of the Conference of the North American Chapter of the Association for Computational Linguistics on Human Language Technology*, 252–259. ACL.

Walker, M. A.; Grant, R.; Sawyer, J.; Lin, G. I.; Wardrip-Fruin, N.; and Buell, M. 2011. Perceived or not perceived: Film character models for expressive NLG. In Si, M.; Thue, D.; André, E.; Lester, J. C.; Tanenbaum, J.; and Zammitto, V., eds., *Interactive Storytelling*, 109–121. Springer Berlin Heidelberg.

Yannakakis, G. N., and Togelius, J. 2011. Experience-driven procedural content generation. *IEEE Transactions on Affective Computing* 2(3):147–161.