Generating Animations from Screenplays

Yeyao Zhang\textsuperscript{1,3}, Eleftheria Tsipidi\textsuperscript{1}, Sasha Schriber\textsuperscript{1}, Mubbasir Kapadia\textsuperscript{1,2}, Markus Gross\textsuperscript{1,3}, Ashutosh Modi\textsuperscript{1}
\textsuperscript{1}Disney Research, \textsuperscript{2}Rutgers University, \textsuperscript{3}ETH Zurich
yezhang@inf.ethz.ch
\{etsipidi, sasha.schriber\}@disneyresearch.com
mubbasir.kapadia@rutgers.edu
\{gross, ashutosh.modi\}@disneyresearch.com

Abstract

Automatically generating animation from natural language text finds application in a number of areas e.g. movie script writing, instructional videos, and public safety. However, translating natural language text into animation is a challenging task. Existing text-to-animation systems can handle only very simple sentences, which limits their applications. In this paper, we develop a text-to-animation system which is capable of handling complex sentences. We achieve this by introducing a text simplification step into the process. Building on an existing animation generation system for screenwriting, we create a robust NLP pipeline to extract information from screenplays and map them to the system’s knowledge base. We develop a set of linguistic transformation rules that simplify complex sentences. Information extracted from the simplified sentences is used to generate a rough storyboard and video depicting the text. Our sentence simplification module outperforms existing systems in terms of BLEU and SARI metrics. We further evaluated our system via a user study: 68\% participants believe that our system generates reasonable animation from input screenplays.

1. Introduction

Generating animation from texts can be useful in many contexts e.g. movie script writing (Ma and Kevitt, 2006; Liu and Leung, 2006; Hanser et al., 2010), instructional videos (Lu and Zhang, 2002), and public safety (Johansson et al., 2004). Text-to-animation systems can be particularly valuable for screenwriting by enabling faster iteration, prototyping and proof of concept for content creators.

In this paper, we propose a text-to-animation generation system. Given an input text describing a certain activity, the system generates a rough animation of the text. We are addressing a practical setting, where we do not have any annotated data for training a supervised end-to-end system. The aim is not to generate a polished, final animation, but a pre-visualization of the input text. The purpose of the system is not to replace writers and artists, but to make their work more efficient and less tedious. We are aiming for a system which is robust and could be deployed in a production environment.

Existing text-to-animation systems for screenwriting (§2) visualize stories by using a pipeline of Natural Language Processing (NLP) techniques for extracting information from texts and mapping them to appropriate action units in the animation engine. The NLP modules in these systems translate the input text into predefined intermediate action representations and the animation generation engine produces simple animation from these representations.

Although these systems can generate animation from carefully handcrafted simple sentences, translating real screenplays into coherent animation still remains a challenge. This can be attributed to the limitations of the NLP modules used with regard to handling complex sentences. In this paper, we try to address the limitations of the current text-to-animation systems. Main contributions of this paper are:

- We propose a screenplay parsing architecture which generalizes well on different screenplay formats (§3.1).
- We develop a rich set of linguistic rules to reduce complex sentences into simpler ones to facilitate information extraction (§3.2).
- We develop a new NLP pipeline to generate animation from actual screenplays (§3).

The potential applications of our contributions are not restricted to just animating screenplays. The techniques we develop are fairly general and can be used in other applications as well e.g. in-
Figure 1: System Architecture: Screenplays are first segmented into different functional blocks. Then, the descriptive action sentences are simplified. Simplified sentences are used to generate animation.

2. Related Work

Translating texts into animation is not a trivial task, given that neither the input sentences nor the output animations have a fixed structure. Prior work addresses this problem from different perspectives (Hassani and Lee, 2016).

CONFUCIUS (Ma and Kevitt, 2006) is a system that converts natural language to animation using the FDG parser (Tapanainen and Järvinen, 1997) and WordNet (Miller, 1995). ScriptViz (Liu and Leung, 2006) is another similar system, created for screenwriting. It uses the Apple Pie parser (Sekine, 1998) to parse input text and then recognizes objects via an object-specific reasoner. It is limited to sentences having conjunction between two verbs. SceneMaker (Hanser et al., 2010) adopts the same NLP techniques as proposed in CONFUCIUS (Ma and Kevitt, 2006) followed by a context reasoning module. Similar to previously proposed systems, we also use dependency parsing followed by linguistic reduction (§3.2).

Recent advances in deep learning have pushed the state of the art results on different NLP tasks (Honnibal and Johnson, 2015; Wolf et al., 2018; He et al., 2017). We use pre-trained models for dependency parsing, coreference resolution and SRL to build a complete NLP pipeline to create intermediate action representations. For the action representation (§3.4), we use a key-value pair structure inspired by the PAR architecture (Badler et al., 2000), which is a knowledge base of representations for actions performed by virtual agents.

Our work comes close to the work done in the area of Open Information Extraction (IE) (Niklaus et al., 2018). In particular, to extract information, Clause-Based Open IE systems (Del Corro and Gemulla, 2013; Angeli et al., 2015; Schmidek and Barbosa, 2014) reduce a complex sentence into simpler sentences using linguistic patterns. However, the techniques developed for these systems do not generalize well to screenplay texts, as these systems have been developed using well-formed and factual texts like Wikipedia, Penn TreeBank, etc. An initial investigation with the popular Open IE system OLLIE (Open Language Learning for Information Extraction) (Mausam et al., 2012) did not yield good results on our corpus.

Previous work related to information extraction for narrative technologies includes the CARDINAL system (Marti et al., 2018; Sanghrajka et al., 2018), as well as the conversational agent PICA (Falk et al., 2018). They focus on querying knowledge from stories. The CARDINAL system also generates animations from input texts. However, neither of the tools can handle complex sentences. We build on the CARDINAL system. We develop a new NLP module to support complex sentences and leverage the animation engine of CARDINAL.

Recently, a number of end-to-end image generation systems have been proposed (Mansimov et al., 2015; Reed et al., 2016). But these systems do not synthesize satisfactory images yet, and are not suitable for our application. It is hoped that the techniques proposed in this paper could be used for automatically generating labelled da-
ta (e.g. (text,video) pairs) for training end-to-end text-to-animation systems.

3. Text-to-Animation System

We adopt a modular approach for generating animations from screenplays. The general overview of our approach is presented in Figure 1. The system is divided into three modules:

- **Script Parsing Module**: Given an input screenplay text, this module automatically extracts the relevant text for generating the animation (§3.1).
- **NLP Module**: It processes the extracted text to get relevant information. This has two sub-modules:
  - **Text Simplification Module**: It simplifies complex sentences using a set of linguistic rules (§3.2).
  - **Information Extraction Module**: It extracts information from the simplified sentences into pre-defined action representations (§3.4).
- **Animation Generation Module**: It generates animation based on action representations (§3.5).

3.1. Script Parsing Module

Typically, screenplays or movie scripts or scripts (we use the terms interchangeably), are made of several scenes, each of which corresponds to a series of consecutive motion shots. Each scene contains several functional components: Headings (time and location), Descriptions (scene description, character actions), Character Cues (character name before dialog), Dialogs (conversation content), Slug Lines (actions inserted into continuous dialog) and Transitions (camera movement). In many scripts, these components are easily identifiable by indentation, capitalization and keywords. We call these scripts well-formatted, and the remaining ones ill-formatted. We want to segment the screenplays into components and are mainly interested in the Descriptions component for animation generation.

**Well-formatted Scripts**: We initially tried ScreenPy (Winer and Young, 2017) to annotate the well-formatted scripts with the component labels. However, ScreenPy did not perform well on our data. We developed our own model, based on Finite State Machines (FSM), for parsing scripts (for details refer to Appendix A). Due to space limitations, we do not describe the FSM model; the key idea is that the model uses hand-crafted transition rules to segment the input screenplay and generates (paragraph, component name) pairs.

**Ill-formatted Scripts**: Taking all the (paragraph, component name) pairs generated by the FSM as ground truth, an SVM model is trained to segment ill-formatted screenplays with inconsistent indentations. For extracting features, each paragraph is encoded into a fix-sized vector using a pre-trained Universal Sentence Encoder. The SVM is trained and tested on a 9:1 train-test data split. The result shows an accuracy of 92.72% on the test set, which is good for our purposes, as we are interested mainly in the Descriptions component.

**Coreference Resolution**: Screenplays contain a number of ambiguous entity mentions (e.g. pronouns). In order to link mentions of an entity, an accurate coreference resolution system is required. The extracted Descriptions components are processed with the NeuralCoref system. Given the text, it resolves mentions (typically pronouns) to the entity they refer to in the text. To facilitate entity resolution, we prepend each Description component with the Character Cues component which appears before it in the screenplay (e.g. [character]MARTHA: [dialog]“I knew it!” [description]She then jumps triumphantly → MARTHA. She then jumps triumphantly).

3.2. Text Simplification Module

In a typical text-to-animation system, one of the main tasks is to process the input text to extract...
There has been work on text simplification (typically verbs) and participants (typically subject/object of the verb), which is subsequently used for generating animation. This works well for simple sentences having a single verb with one subject and one (optional) object. However, the sentences in a screenplay are complicated and sometimes informal. In this work, a sentence is said to be complicated if it deviates from easily extractable and simple subject-verb-object (and its permutations) syntactic structures and possibly has multiple actions mentioned within the same sentence with syntactic interactions between them. By syntactic structure we refer to the dependency graph of the sentence.

In the case of screenplays, the challenge is to process such complicated texts. We take the text simplification approach, i.e. the system first simplifies a complicated sentence and then extracts the relevant information. Simplification reduces a complicated sentence into multiple simpler sentences, each having a single action along with its participants, making it straightforward to extract necessary information.

Recently, end-to-end Neural Text Simplification (NTS) systems (Nisioi et al., 2017; Saggio, 2017) have shown reasonable accuracy. However, these systems have been trained on factual data such as Wikipedia and do not generalize well to screenplay texts. Our experiments with such a pre-trained neural text simplification system did not yield good results (§5.1). Moreover, in the context of text-to-animation, there is no standard labeled corpus to train an end-to-end system.

There has been work on text simplification using linguistic rules-based approaches. For example, (Siddharthan, 2011) propose a set of rules to manipulate sentence structure to output simplified sentences using syntactic dependency parsing. Similarly, the YATS system (Ferrés et al., 2016) implements a set of rules in the JAPE language (Cunningham et al., 2000) to address six syntactic structures: Passive Constructions, Appositive Phrases, Relative Clauses, Coordinated Clauses, Correlated Clauses and Adverbial Clauses.

Most of the rules focus on case and tense correction, with only 1-2 rules for sentence splitting. We take inspiration from the YATS system, and our system incorporates modules to identify and transform sentence structures into simpler ones using a broader set of rules.

In our system, each syntactic structure is handled by an Analyzer, which contains two processes: Identify and Transform. The Identify process takes in a sentence and determines if it contains a particular syntactic structure. Subsequently, the Transform process focuses on the first occurrence of the identified syntactic structure and then splits and assembles the sentence into simpler sentences. Both Identify and Transform use Part-of-Speech (POS) tagging and dependency parsing (Honnibal and Montani, 2017) modules implemented in spaCy 2.0.3

The simplification algorithm (Algorithm 1) starts with an input sentence and recursively processes it until no further simplification is possible. It uses a queue to manage intermediate simplified sentences, and runs in a loop until the queue is empty. For each sentence, the system applies each syntactic analyzer to Identify the correspondent

### Table 1: Linguistic rules for text simplification module

<table>
<thead>
<tr>
<th>Syntactic Structure</th>
<th>Identify procedure</th>
<th>Transform procedure</th>
</tr>
</thead>
<tbody>
<tr>
<td>Coordination</td>
<td>search all cc and con conj in dependency tree</td>
<td>cut cc and conj link. If conj is verb, mark it as new root, else replace it with its sibling node.</td>
</tr>
<tr>
<td>Pre-Correlative Conjunction</td>
<td>locates positions of keywords: &quot;either&quot;, &quot;both&quot;, &quot;neither&quot;</td>
<td>removed the located word from dependency tree</td>
</tr>
<tr>
<td>Appositive Clause</td>
<td>find appos token and its head (none)</td>
<td>glue appositive noun phrase with ‘to be’</td>
</tr>
<tr>
<td>Relative Clause</td>
<td>find rel node and its head</td>
<td>cut appos link, then traverse from root. Then, if no “wh” word present, put head part after anchor part, else, we divide them into 5 subcases (Table 2)</td>
</tr>
<tr>
<td>Adverbial Clause Modifier</td>
<td>find advcl token and its head. Also conjuncts of head token</td>
<td>cut advcl edge. If advcl token does not have subject, add subject of root as advcl's most-left child and remove prep and mark token. Then traverse from both root and advcl token</td>
</tr>
<tr>
<td>Inverted Clause Subject</td>
<td>attr token has to be the child of head or advcl token</td>
<td>change position of actual verb and subject</td>
</tr>
<tr>
<td>Clausal Complement</td>
<td>find ccomp token in dependency tree</td>
<td>cut ccomp link, add subject to subordinate clause if necessary</td>
</tr>
<tr>
<td>Passive Voice</td>
<td>check presence of auxpass or ccomp optinally for auxpass and agent</td>
<td>cut auxpass link if any. Cut ccomp or auxpass link. Prepend subject token to verb token’s right children. Finally append suitable subject.</td>
</tr>
<tr>
<td>Open Clause Complement</td>
<td>find xcomp verb token and actual verb token</td>
<td>cut aux pass, then replace ccomp-verb in subject’s children with actual-verb, traverse from actual-verb, else, cut ccomp link, traverse from xcomp-verb</td>
</tr>
<tr>
<td>Adjective Clause</td>
<td>find acl verb token and its head</td>
<td>cut acl link. Link subject node with it. Traverse from acl node</td>
</tr>
</tbody>
</table>

3https://spacy.io
<table>
<thead>
<tr>
<th>Type</th>
<th>Example Input Sentence</th>
<th>System Output Sentence 1</th>
<th>System Output Sentence 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Coordination</td>
<td>She LAUGHS, and [re] give [conj] Kevin a kiss.</td>
<td>She laughs.</td>
<td>She gives Kevin a kiss.</td>
</tr>
<tr>
<td>Pre-Correlative</td>
<td>It’s followed by another squad car, both [preconj] with sirens blaring.</td>
<td>It’s followed by another squad car, with sirens blaring.</td>
<td>–</td>
</tr>
<tr>
<td>Relative-dobj</td>
<td>She pulls out a letter, which [dobj] she hand [rel] to Kevin.</td>
<td>She pulls out a letter.</td>
<td>She hands a letter to Kevin.</td>
</tr>
<tr>
<td>Relative-pobj</td>
<td>A reef encloses the cove where [pobj] he came [rel] from.</td>
<td>A reef encloses the cove.</td>
<td>he comes from the cove.</td>
</tr>
<tr>
<td>Relative-advmod</td>
<td>Frank gestures to the SALESMAN, who [nsubj]’s waiting [rel] on a woman.</td>
<td>Frank gestures to the SALESMAN.</td>
<td></td>
</tr>
<tr>
<td>Relative-poss</td>
<td>The girl, whose [poss] name is [rel] Helga, answers.</td>
<td>The girl answers.</td>
<td>The girl’s name is Helga.</td>
</tr>
<tr>
<td>Inverted Cl. Subject</td>
<td>Running [conj] towards Oz is Steve Stifler.</td>
<td>Steve Stifler runs towards Oz.</td>
<td>–</td>
</tr>
<tr>
<td>Clausal Component</td>
<td>The thing is, it actually sounds [ecomp] really good.</td>
<td>The thing is, it actually sounds really good.</td>
<td>It actually sounds really good.</td>
</tr>
<tr>
<td>Passive Voice</td>
<td>They [nsubj] pass are suddenly illuminated by the glare of headlights.</td>
<td>Suddenly the glare of headlights illuminated them.</td>
<td>–</td>
</tr>
<tr>
<td>Open Clausal</td>
<td>The sophomore comes running [con] through the kitchen.</td>
<td>The sophomore runs through the kitchen.</td>
<td>The sophomore comes.</td>
</tr>
<tr>
<td>Adjective</td>
<td>Stifler has a toothbrush hanging [act] from his mouth.</td>
<td>A toothbrush hangs from Stifler’s mouth.</td>
<td>Stifler has a toothbrush.</td>
</tr>
</tbody>
</table>

Table 2: Syntactic simplification rules applied on example sentences.

ding syntactic structure in the sentence (line 14). If the result is positive, the sentence is processed by the Transform function to convert it to simple sentences (line 16). Each of the output sentences is pushed by the controller into the queue (line 19). The process is repeated with each of the Identiﬁers analyzers (line 13). If none of the analyzers can be applied, the sentence is assumed to be simple and it is pushed into the result list (line 21). We summarize linguistic rules in Table 1 and examples are given in Table 2. Next, we describe the coordination linguistic rules. For details regarding other rules, please refer to Appendix B.

**Coordination**: Coordination is used for entities having the same syntactic relation with the head and serving the same functional role (e.g. subj, obj, etc.). It is the most important component in our simpliﬁcation system. The parser tags word units such as “and” and “as well as” with the dependency label cc, and the conjugated words as conj. Our system deals with coordination based on the dependency tag of the conjugated word.

In the case of coordination, the Identiﬁer function simply returns whether cc or conj is in the dependency graph of the input sentence. The Transform function manipulates the graph structure based on the dependency tags of the conjugated words as shown in Figure 2. If the conjugated word is a verb, then we mark it as another root of the sentence. Cutting cc and conj edges in the graph and traversing from this new root results in a new sentence parallel to the original one. In other cases, such as the conjugation between nouns, we simply replace the noun phrases with their siblings and traverse from root again.

### 3.3. Lexical Simpliﬁcation

In order to generate animation, actions and participants extracted from simpliﬁed sentences are mapped to existing actions and objects in the animation engine. Due to practical reasons, it is not possible to create a library of animations for all possible actions in the world. We limit our library to a predefined list of 52 actions/animations, expanded to 92 by a dictionary of synonyms (§3.5). We also have a small library of pre-uploaded objects (such as “campﬁre”, “truck” and others).

To animate unseen actions not in our list, we use a word2vec-based similarity function to ﬁnd the nearest action in the list. Moreover, we use WordNet (Miller, 1995) to exclude antonyms. This helps to map non-list actions (such as “squint at”) to the similar action in the list (e.g. “look”). If we fail to ﬁnd a match, we check for a mapping while including the verb’s preposition or syntactic object. We also use WordNet to obtain hypernyms for further checks, when the similarity function fails to ﬁnd a close-enough animation. Correspondingly, for objects, we use the same similarity function and WordNet’s holonyms.

```
<table>
<thead>
<tr>
<th>Type</th>
<th>Example Input Sentence</th>
<th>System Output Sentence 1</th>
<th>System Output Sentence 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Coordination</td>
<td>She LAUGHS, and [re] give [conj] Kevin a kiss.</td>
<td>She laughs.</td>
<td>She gives Kevin a kiss.</td>
</tr>
<tr>
<td>Pre-Correlative</td>
<td>It’s followed by another squad car, both [preconj] with sirens blaring.</td>
<td>It’s followed by another squad car, with sirens blaring.</td>
<td>–</td>
</tr>
<tr>
<td>Relative-dobj</td>
<td>She pulls out a letter, which [dobj] she hand [rel] to Kevin.</td>
<td>She pulls out a letter.</td>
<td>She hands a letter to Kevin.</td>
</tr>
<tr>
<td>Relative-pobj</td>
<td>A reef encloses the cove where [pobj] he came [rel] from.</td>
<td>A reef encloses the cove.</td>
<td>he comes from the cove.</td>
</tr>
<tr>
<td>Relative-advmod</td>
<td>Frank gestures to the SALESMAN, who [nsubj]’s waiting [rel] on a woman.</td>
<td>Frank gestures to the SALESMAN.</td>
<td></td>
</tr>
<tr>
<td>Relative-poss</td>
<td>The girl, whose [poss] name is [rel] Helga, answers.</td>
<td>The girl answers.</td>
<td>The girl’s name is Helga.</td>
</tr>
<tr>
<td>Inverted Cl. Subject</td>
<td>Running [conj] towards Oz is Steve Stifler.</td>
<td>Steve Stifler runs towards Oz.</td>
<td>–</td>
</tr>
<tr>
<td>Clausal Component</td>
<td>The thing is, it actually sounds [ecomp] really good.</td>
<td>The thing is, it actually sounds really good.</td>
<td>It actually sounds really good.</td>
</tr>
<tr>
<td>Passive Voice</td>
<td>They [nsubj] pass are suddenly illuminated by the glare of headlights.</td>
<td>Suddenly the glare of headlights illuminated them.</td>
<td>–</td>
</tr>
<tr>
<td>Open Clausal</td>
<td>The sophomore comes running [con] through the kitchen.</td>
<td>The sophomore runs through the kitchen.</td>
<td>The sophomore comes.</td>
</tr>
<tr>
<td>Adjective</td>
<td>Stifler has a toothbrush hanging [act] from his mouth.</td>
<td>A toothbrush hangs from Stifler’s mouth.</td>
<td>Stifler has a toothbrush.</td>
</tr>
</tbody>
</table>
```
Our list of actions and objects is not exhaustive. Currently, we do not cover actions which may not be visual. For out of list actions, we give the user a warning that the action cannot be animated. Nevertheless, this is a work in progress and we are working on including more animations for actions and objects in our knowledge base.

3.4. Action Representation Field (ARF): Information Extraction

For each of the simplified sentences, information is extracted and populated into a predefined key-value pair structure. We will refer to the keys of this structure as Action Representation Fields (ARFs). These are similar to entities and relations in Knowledge Bases. ARFs include: owner, target, prop, action, origin action, manner, modifier location, modifier direction, start-time, duration, speed, translation, rotation, emotion, partial start time (for more details see Appendix C). This structure is inspired by the PAR (Badler et al., 2000) architecture, but adapted to our needs.

To extract the ARFs from the simplified sentences, we use a Semantic Role Labelling (SRL) model in combination with some heuristics, for example creating word lists for duration, speed, translation, rotation, emotion. We use a pre-trained Semantic Role Labelling model based on a Bi-directional LSTM network (He et al., 2017) with pre-trained ELMo embeddings (Peters et al., 2018). We map information from each sentence to the knowledge base of animations and objects.

3.5. Animation Generation

We use the animation pipeline of the CARDINAL system. We plug in our NLP module in CARDINAL to generate animation. CARDINAL creates pre-visualizations of the text, both in storyboard form and animation. A storyboard is a series of pictures that demonstrates the sequence of scenes from a script. The animation is a 3-D animated video that approximately depicts the script. CARDINAL uses the Unreal game engine (Games, 2007) for generating pre-visualizations. It has a knowledge base of pre-baked animations (52 animations, plus a dictionary of synonyms, resulting in 92) and pre-uploaded objects (e.g. “campfire”, “tent”). It also has 3-D models which can be used to create characters.

4. Text-to-Animation Corpus

We initially used a corpus of Descriptions components from ScreenPy (Winer and Young, 2017), in order to study linguistic patterns in the movie script domain. Specifically, we used the “heading” and “transition” fields from ScreenPy’s published JSON output on 1068 movie scripts scraped from IMSDb. We also scraped screenplays from SimplyScripts and ScriptORama. After separating screenplays into well-formatted and ill-formatted, Descriptions components were extracted using our model (§3.1). This gave a corpus of Descriptions blocks from 996 screenplays.

The corpus contains a total of 525,708 Descriptions components. The Descriptions components contain a total of 1,402,864 sentences. Out of all the Descriptions components, 49.45 % (259,973) contain at least one verb which is in the animation list (henceforth called “action verbs”). Descriptions components having at least one action verb have in total 920,817 sentences. Out of the-
Carl touches Ellie’s shoulder as the doctor explains. Ellie drops her head in her hands.

<table>
<thead>
<tr>
<th>System output</th>
<th>Annotator I</th>
<th>Annotator II</th>
<th>BLEU$_2$(%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Carl touches Ellie’s shoulder</td>
<td>carl touches ellie’s shoulder</td>
<td>carl touches ellie’s shoulder</td>
<td>38.73</td>
</tr>
<tr>
<td>the doctor explains</td>
<td>the doctor explains</td>
<td>the doctor is talking</td>
<td>100</td>
</tr>
<tr>
<td>Ellie drops Ellie’s head in Ellie’s hands</td>
<td>elie drops her head in her hands</td>
<td>elie drops her head in her hands</td>
<td>48.79</td>
</tr>
</tbody>
</table>

Table 3: Differences between system output and annotator responses

<table>
<thead>
<tr>
<th></th>
<th>BLEU</th>
<th>SARI</th>
</tr>
</thead>
<tbody>
<tr>
<td>NTS-w2v</td>
<td>61.45</td>
<td>36.94</td>
</tr>
<tr>
<td>YATS</td>
<td>58.83</td>
<td>48.75</td>
</tr>
<tr>
<td>Our System</td>
<td>67.68</td>
<td>50.65</td>
</tr>
</tbody>
</table>

Table 4: Results on syntactic simplification

We have 3 corresponding references $r^1$, $r^2$, and $r^3$. We also evaluate using the SARI (Xu et al., 2016) score to evaluate our text simplification module.

5.1. Sentence Simplification

Each action block $a$ is reduced to a set of simple sentences $S_a = \{s_1, s_2, ..., s_{n_a}\}$. And for the same action block $a$, each annotator $t$, $t \in \{1, 2, 3\}$ produces a set of simplified sentences $R^t_a = \{r^t_1, r^t_2, ..., r^t_{m_a}\}$. Since the simplification rules in our system may not maintain the original ordering of verbs, we do not have sentence level alignment between elements in $S_a$ and $R^t_a$. For example, action block $a = He\text{ laughs after he jumps into the water}$ is reduced by our system into two simplified sentences $S_a = \{s_1 = He\text{ jumps into the water}, s_2 = He\text{ laughs}\}$ by the temporal heuristics, while annotator gives us $R^t_a = \{r^t_1 = He\text{ laughs}, r^t_2 = He\text{ jumps into the water}\}$. In such cases, sequentially matching $s_i$ to $r_j$ will result in a wrong (hypothesis, reference) alignment which is $(s_1, r^t_2)$ and $(s_2, r^t_3)$.

To address this problem, for each hypothesis $s_i \in S_a$, we take the corresponding reference $r^t_i \in R^t_a$ as the one with the least Levenshtein Distance (Navarro, 2001) to $s_i$, i.e.,

$$r^t_i = \arg \min_{r_j \in R^t_a} lev\text{-}dist(s_i, r_j), \forall j \in \{1, ..., m_a\}$$

As per this alignment, in the above example, we will have correct alignments $(s_1, r^t_2)$ and $(s_2, r^t_3)$. Thus, for each simplified sentence $s_i$ we have 3 corresponding references $r^t_1$, $r^t_2$, and $r^t_3$. The aligned sentences are used to calculate corpus level BLEU score$^6$ and SARI score.$^7$

The evaluation results for text simplification are summarized in Table 4. We compare against YATS (Ferrés et al., 2016) and neural end-to-end text simplification system NTS-w2v (Nisioi et al.,

---

$^6$We used NLTK’s API with default parameters: [link]

$^7$Implementation available at [link]
YATS is also a rule-based text simplification system. As shown in Table 4, our system performs better than YATS on both the metrics, indicative of the limitations of the YATS system. A manual examination of the results also showed the same trend. However, the key point to note is that we are not aiming for text simplification in the conventional sense. Existing text simplification systems tend to summarize text and discard some of the information. Our aim is to break a complex sentence into simpler ones while preserving the information.

An example of a Descriptions component with BLEU scores is given in Table 3. In the first simplified sentence, the space between Ellie and ‘s causes the drop in the score. But it gives exactly the same answer as both annotators. In the second sentence, the system output is the same as the annotator I’s answer, so the BLEU score is 1. In the last case, the score is low, as annotators possibly failed to replace her with the actual Character Cue Ellie. Qualitative examination reveals, in general, that our system gives a reasonable result for the syntactic simplification module. As exemplified, BLEU is not the perfect metric to evaluate our system, and therefore in the future we plan to explore other metrics.

### 5.1.2. ARF Evaluation

We also evaluate the system’s output for action representation fields against gold annotations. In our case, some of the fields can have multiple (2 or 3) words such as owner, target, prop, action, origin action, manner, location and direction. We use BLEU as the evaluation metric to measure the BOW similarity between system output and ground truth references. The results are shown in Table 5.

<table>
<thead>
<tr>
<th>Field</th>
<th>BLEU1</th>
</tr>
</thead>
<tbody>
<tr>
<td>owner</td>
<td>56.16</td>
</tr>
<tr>
<td>target</td>
<td>41.85</td>
</tr>
<tr>
<td>prop</td>
<td>28.98</td>
</tr>
<tr>
<td>action</td>
<td>70.46</td>
</tr>
<tr>
<td>emotion</td>
<td>57.89</td>
</tr>
</tbody>
</table>

Table 5: Results on textual ARFs (%)  

<table>
<thead>
<tr>
<th>Field</th>
<th>P</th>
<th>R</th>
<th>F1</th>
</tr>
</thead>
<tbody>
<tr>
<td>s_time</td>
<td>86.49</td>
<td>68.63</td>
<td>76.53</td>
</tr>
<tr>
<td>rot.</td>
<td>82.04</td>
<td>81.16</td>
<td>81.60</td>
</tr>
<tr>
<td>duration</td>
<td>94.72</td>
<td>73.92</td>
<td>83.04</td>
</tr>
<tr>
<td>transl.</td>
<td>75.49</td>
<td>86.47</td>
<td>80.61</td>
</tr>
<tr>
<td>speed</td>
<td>94.41</td>
<td>79.50</td>
<td>86.32</td>
</tr>
</tbody>
</table>

Table 6: Result on Non-textual ARFs( %)

In identifying owner, target and prop, the system tends to use a fixed long mention, while annotators prefer short mentions for the same character/object. The score of prop is relatively lower than all other fields, which is caused by a systematic SRL mapping error. The relatively high accuracy on the action field indicates the consistency between system output and annotator answers.

Annotation on the emotion ARF is rather subjective. Responses on the this field are biased and noisy. The BLEU score on this is relatively low. For the other non-textual ARFs, we use precision and recall to measure the system’s behavior. Results are shown in Table 6. These fields are subjective: annotators tend to give different responses for the same input sentence.

rotation and translation have Boolean values. Annotators agree on these two fields in most of the sentences. The system, on the other hand, fails to identify actions involving rotation. For example, in the sentence “Carl closes CARL’s door sharply” all four annotators think that this sentence involves rotation, which is not found by the system. This is due to the specificity of rules on identifying these two fields.

speed, duration and start time have high precision and low recall. This indicates the inconsistency in annotators’ answers. For example, in the sentence “Woody runs around to the back of the pizza truck”, two annotators give 2 seconds and another gives 1 second in duration. These fields are subjective and need the opinion of the script author or the director. In the future, we plan to involve script editors in the evaluation process.

### 5.2. Extrinsic Evaluation

We conducted a user study to evaluate the performance of the system qualitatively. The focus of the study was to evaluate (from the end user’s perspective) the performance of the NLP component w.r.t. generating reasonable animations.

We developed a questionnaire consisting of 20 sentence-animation video pairs. The animations were generated by our system. The questionnaire was filled by 22 participants. On average it took around 25 minutes for a user to complete the study.

We asked users to evaluate, on a five-point Likert scale (Likert, 1932), if the video shown was a reasonable animation for the text, how much of
the text information was depicted in the video and how much of the information in the video was present in the text (Table 7). The 68.18% of the participants rated the overall pre-visualization as neutral or above. The rating was 64.32% (neutral or above) for the conservation of textual information in the video, which is reasonable, given limitations of the system that are not related to the NLP component. For the last question, 75.90% (neutral or above) agreed that the video did not have extra information. In general, there seemed to be reasonable consensus in the responses. Besides the limitations of our system, disagreement can be attributed to the ambiguity and subjectivity of the task.

We also asked the participants to describe qualitatively what textual information, if any, was missing from the videos. Most of the missing information was due to limitations of the overall system rather than the NLP component: facial expression information was not depicted because the character 3-D models are deliberately designed without faces, so that animators can draw on them. Information was also missing in the videos if it referred to objects or actions that do not have a close enough match in the object list or animations list. Furthermore, the animation component only supports animations referring to a character or object as a whole, not parts, (e.g. “Ben raises his head” is not supported).

However, there were some cases where the NLP component can be improved. For example, lexical simplification failed to map the verb “watches” to the similar animation “look”. In one case, syntactic simplification created only two simplified sentences for a verb which had three subjects in the original sentence. In a few cases, lexical simplification successfully mapped to the most similar animation (e.g.”argue” to “talk”) but the participants were not satisfied - they were expecting a more exact animation. We plan to address these shortcomings in future work.

### 6. Conclusion and Future Work

In this paper, we proposed a new text-to-animation system. The system uses linguistic text simplification techniques to map screenplay text to animation. Evaluating such systems is a challenge. Nevertheless, intrinsic and extrinsic evaluations show reasonable performance of the system. The proposed system is not perfect, for example, the current system does not take into account the discourse information that links the actions implied in the text, as currently the system only processes sentences independently. In the future, we would like to leverage discourse information by considering the sequence of actions which are described in the text (Modi and Titov, 2014; Modi, 2016). This would also help to resolve ambiguity in text with regard to actions (Modi et al., 2017; Modi, 2017). Moreover, our system can be used for generating training data which could be used for training an end-to-end neural system.

### Acknowledgments

We would like to thank anonymous reviewers for their insightful comments. We would also like to thank Daniel Inversini, Isabel Simó Aynés, Carolina Ferrari, Roberto Comella and Max Grosse for their help and support in developing the Cardinal system. Mubbasir Kapadia has been funded in part by NSF IIS-1703883 and NSF S&AS-1723869.

### References


Appendix A

Figure 3: FSM of Well-formatted Screenplay Parser. Numbers of link are Rules and ~ means logical NOT

<table>
<thead>
<tr>
<th>ID</th>
<th>Rule Summary</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>If input is uppercase and contains heading words such as ‘INT’, ‘EXT’. etc, return True, otherwise False</td>
</tr>
<tr>
<td>2</td>
<td>If input is uppercase and contains character words such as ‘CONT.’, ‘(O.S)’, or if #indentation &gt; most frequent #indentation, return True. Otherwise False</td>
</tr>
<tr>
<td>3</td>
<td>If input starts with ‘(‘, return True, otherwise False</td>
</tr>
<tr>
<td>4</td>
<td>If input ends with ‘)’, return True, otherwise False</td>
</tr>
<tr>
<td>5</td>
<td>If</td>
</tr>
<tr>
<td>6</td>
<td>If the input is uppercase and contains transition words such as ‘DISSOLVE’, ‘CUT TO’. etc, return True, otherwise False</td>
</tr>
<tr>
<td>7</td>
<td>If the input equals to ‘THE END’, return True. Otherwise False.</td>
</tr>
</tbody>
</table>

Table 8: FSM Transition Rules
<table>
<thead>
<tr>
<th>COMPLEX: Another parent, Mike Munson, sits on the bench with a tablet and uses an app to track and analyze the team’s shots.</th>
</tr>
</thead>
<tbody>
<tr>
<td>NSELSTM-B: Another parent, Mike Munson, sits on the bench with a tablet and uses an app to track.</td>
</tr>
<tr>
<td>YATS: Another parent sits on the bench with a tablet and uses an app to track and examines the team’s shots. This parent is Mike Munson.</td>
</tr>
<tr>
<td>OURS: Another parent is Mike Munson. Another parent sits on the bench with a tablet. Another parent uses an app.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>COMPLEX: Stowell believes that even documents about Lincoln’s death will give people a better understanding of the man who was assassinated 150 years ago this April.</th>
</tr>
</thead>
<tbody>
<tr>
<td>NSELSTM-B: Stowell believes that the discovery about Lincoln’s death will give people a better understanding of the man.</td>
</tr>
<tr>
<td>YATS: Stowell believes that even documents about Lincoln’s death will give people a better reason of the man. This man was assassinated 150 years ago this April.</td>
</tr>
<tr>
<td>OURS: Stowell believes. Even documents about Lincoln’s death give people a better understanding of the man. Somebody assassinates the man 150 years ago this April.</td>
</tr>
</tbody>
</table>

Table 9: Example Model Outputs
Appendix B: Algorithms

Algorithm 2 Identify Adverbial Clause

1: procedure ADVERBIAL CLAUSE IDENTIFY PROCEDURE(sent)  
2: find tokens in sentences with dependency tag ROOT and ADVCL (we call it advcl)  
3: if no ADVCL token in sent then  
4: return False  
5: find father token of advcl as father  
6: if father is not a VERB then  
7: we make it as a VERB  
8: find conjunction tokens of father as conjuncts  
9: if NOUN in advcl's left subtree then  
10: subject ← this NOUN  
11: else  
12: subject ← NOUN in father's left subtree  
13: return True

Algorithm 3 Transform Adverbial Clause

1: procedure ADVERBIAL CLAUSE TRANSFORM PROCEDURE(sent)  
2: cut edge between root and advcl token  
3: if advcl verb does not have its own subject then  
4: add subject as advcl's most left direct child  
5: remove PREP and MARK token in advcl's children, modify temporal id accordingly  
6: str1 ← traverse_a_string(root)  
7: str2 ← correct_tense(traverse_a_string(advcl))  
8: return [str1, str2]

Algorithm 4 Identify() in Relative Clause Analyzer

1: procedure RELATIVE CLAUSE IDENTIFY PROCEDURE(sent)  
2: if no RELCL token in sent then  
3: return False  
4: anchor ← RELCL token  
5: head ← anchor's head token  
6: wg ← NULL  
7: for token t in anchor's children do  
8: if t.tag ∈ [WDT, WP, WP$, WRB] then  
9: wh ← t  
10: return True

Algorithm 5 Transform() in Relative Clause Analyzer

1: procedure RELATIVE CLAUSE TRANSFORM PROCEDURE(root, anchor, head, wh)  
2: cut relcl edge between head and anchor  
3: str1 ← traverse_a_string(root)  
4: if wh = NULL then  
5: str2 ← traverse_a_string(anchor) + traverse_a_string(head)  
6: else  
7: wh-dep ← dependency tag of wh  
8: wh-head ← head of wh  
9: remove wh in anchor's children  
10: if wh-dep = DOBJ then  
11: str2 ← traverse_a_string(wh-head) + traverse_a_string(anchor)  
12: else if wh-dep = POBJ then  
13: put head after wh-head in anchor's children  
14: str2 ← traverse_a_string(anchor)  
15: else if wh-dep ∈ [NSUBJ, NSUBJPASS] then  
16: str2 ← traverse_a_string(wh-head) + traverse_a_string(anchor)  
17: else if wh-dep = ADVMOD then  
18: prep ← 'at'  
19: str2 ← traverse_a_string(wh-head) + prep + traverse_a_string(wh-head)  
20: else if wh-dep = POSS then  
21: str2 ← traverse_a_string(wh-head) + be verb + traverse_a_string(anchor)  
22: correct verb tense in str1 and str2  
23: return [str1, str2]
Algorithm 6 Transform() in Coordination Analyzer

```
1: procedure COORDINATION TRANSFORM PROEDURE(sents)
2: results ← empty list
3: main-root ← root of dependency tree of sents
4: conj-root ← find conj token in dependency tree and their head token main
5: embed all conjugate words of main in a list conj
6: cut conj edge between main and cc and conj
7: if no object for main then
8: try find object in conj’s right children
9: str1 ← traverse_a_string(root)
10: results.append(str1)
11: for conj in conj’s do
12: type ← get_conj_type(main, conj)
13: if type=VERB&VERB then
14: correct part-of-speech tag if necessary
15: if conj has its own subject then
16: new-root ← conj
17: else
18: if main-root then
19: new-root ← conj
20: else
21: replace main with conj in root’s children
22: new-root ← root
23: else
24: main-head ← head of main
25: replace main with conj in main-head’s children
26: new-root ← root
27: str2 ← traverse_a_string(new-root)
28: results.append(str2)
29: return results
```

Algorithm 7 Identify() in Passive Analyzer

```
1: procedure PASSIVE VOICE IDENTIFY PROCEDURE(sents)
2: is-passive ← False
3: for token t in sents do
4: if t.dep ∈ [CSUBPASS, NSUBPASS] then
5: subj-token ← t
6: verb-token ← t
7: is-passive ← True
8: if t.dep ∈ [AUXPASS] and t.head = verb-token then
9: auxpass-token ← t
10: if t.dep ∈ [AGENT] and t.text = 'by' then
11: by-token ← t
12: is-passive ← True
13: if no object for main then
14: correct verb tense for verb-token
15: return is-passive
```

Algorithm 8 Transform() in Passive Analyzer

```
1: procedure PASSIVE VOICE TRANSFORM PROCEDURE(sents)
2: if auxpass-token ≠ NULL then
3: cut auxpass edge
4: cut nsubjpass or csubj edge
5: prepend subject-token to verb-token’s right children
6: if by-token ≠ NULL then
7: cut by-agent edge
8: add by-token’s right children to verb-token’s left children
9: else
10: add ‘Somebody’ to verb-token’s left children
11: correct verb tense for verb-token
12: return traverse_a_string(root-token)
```

Algorithm 9 Transform() in Appositive Clause Analyzer

```
1: procedure APPO SITIVE CLAUSE TRANSFORM PROCEDURE(anchor, head)
2: cut edge between anchor and head
3: str1 ← traverse_a_string(root token of input sentence)
4: str2 ← traverse_a_string(head) + be_verb + traverse_a_string(anch or)
5: correct verb tense in str1 and str2
6: return [str1, str2]
```

Algorithm 10 Identify() in Inverted Clausal Subject Analyzer

```
1: procedure INVERTED CLAUSAL SUBJECT IDENTIFY PROCEDURE(sents)
2: for Token t in sents do
3: if t.dep = CSUBJ and t.tag ∈ [VBN, VBG] and t.head.lemma='be' then
4: attr ← token with dependency label attr in t.head’s right children
5: if attr = NULL then
6: return False
7: actual-verb ← t
8: be-verb ← t.head
9: return True
10: return False
```

Algorithm 11 Identify() in Naive Appositive Clause Analyzer

```
1: procedure NAIVE APPOSITIVE CLAUSAL IDENTIFY PROCEDURE(sents)
2: try find object in sents
3: if attr does not exist then
4: return False
5: attr ← find CSUBJ in sents
6: if attr does not exist then
7: return False
8: find first object in sents
9: if attr does not exist then
10: return False
11: if attr is a proper noun then
12: return False
13: if attr is a person then
14: return False
15: if attr is a place then
16: return False
17: if attr is an organization then
18: return False
19: if attr is a work of art then
20: return False
21: return True
```

Algorithm 12 Transform() in Appositive Clause Analyzer

```
1: procedure APPOSITIVE CLAUSE TRANSFORM PROCEDURE(sents)
2: if attr is a head token then
3: cut auxpass edge
4: cut nsubjpass or csubj edge
5: prepend subject-token to verb-token’s right children
6: if by-token ≠ NULL then
7: cut by-agent edge
8: add by-token’s right children to verb-token’s left children
9: else
10: add ‘Somebody’ to verb-token’s left children
11: correct verb tense for verb-token
12: return traverse_a_string(root-token)
```

Algorithm 13 Transform() in Passive Analyzer

```
1: procedure PASSIVE VOICE TRANSFORM PROCEDURE(sents)
2: if auxpass-token ≠ NULL then
3: cut auxpass edge
4: cut nsubjpass or csubj edge
5: prepend subject-token to verb-token’s right children
6: if by-token ≠ NULL then
7: cut by-agent edge
8: add by-token’s right children to verb-token’s left children
9: else
10: add ‘Somebody’ to verb-token’s left children
11: correct verb tense for verb-token
12: return traverse_a_string(root-token)
```

Algorithm 14 Identify() in Passive Analyzer

```
1: procedure PASSIVE VOICE IDENTIFY PROCEDURE(sents)
2: is-passive ← False
3: for token t in sents do
4: if t.dep ∈ [CSUBPASS, NSUBPASS] then
5: subj-token ← t
6: verb-token ← t
7: is-passive ← True
8: if t.dep ∈ [AUXPASS] and t.head = verb-token then
9: auxpass-token ← t
10: if t.dep ∈ [AGENT] and t.text = 'by' then
11: by-token ← t
12: is-passive ← True
13: if no object for main then
14: correct verb tense for verb-token
15: return is-passive
```

Algorithm 15 Transform() in Passive Analyzer

```
1: procedure PASSIVE VOICE TRANSFORM PROCEDURE(sents)
2: if auxpass-token ≠ NULL then
3: cut auxpass edge
4: cut nsubjpass or csubj edge
5: prepend subject-token to verb-token’s right children
6: if by-token ≠ NULL then
7: cut by-agent edge
8: add by-token’s right children to verb-token’s left children
9: else
10: add ‘Somebody’ to verb-token’s left children
11: correct verb tense for verb-token
12: return traverse_a_string(root-token)
```

Algorithm 16 Transform() in Coordination Analyzer

```
1: procedure COORDINATION TRANSFORM PROCEDURE(sents)
2: results ← empty list
3: main-root ← root of dependency tree of sents
4: conj-root ← find conj token in dependency tree and their head token main
5: embed all conjugate words of main in a list conj
6: cut conj edge between main and cc and conj
7: if no object for main then
8: try find object in conj’s right children
9: str1 ← traverse_a_string(root)
10: results.append(str1)
11: for conj in conj’s do
12: type ← get_conj_type(main, conj)
13: if type=VERB&VERB then
14: correct part-of-speech tag if necessary
15: if conj has its own subject then
16: new-root ← conj
17: else
18: if main-root then
19: new-root ← conj
20: else
21: replace main with conj in root’s children
22: new-root ← root
23: else
24: main-head ← head of main
25: replace main with conj in main-head’s children
26: new-root ← root
27: str2 ← traverse_a_string(new-root)
28: results.append(str2)
29: return results
```

Algorithm 17 Identify() in Passive Analyzer

```
1: procedure PASSIVE VOICE IDENTIFY PROCEDURE(sents)
2: is-passive ← False
3: for token t in sents do
4: if t.dep ∈ [CSUBPASS, NSUBPASS] then
5: subj-token ← t
6: verb-token ← t
7: is-passive ← True
8: if t.dep ∈ [AUXPASS] and t.head = verb-token then
9: auxpass-token ← t
10: if t.dep ∈ [AGENT] and t.text = 'by' then
11: by-token ← t
12: is-passive ← True
13: if no object for main then
14: correct verb tense for verb-token
15: return is-passive
```

Algorithm 18 Transform() in Passive Analyzer

```
1: procedure PASSIVE VOICE TRANSFORM PROCEDURE(sents)
2: if auxpass-token ≠ NULL then
3: cut auxpass edge
4: cut nsubjpass or csubj edge
5: prepend subject-token to verb-token’s right children
6: if by-token ≠ NULL then
7: cut by-agent edge
8: add by-token’s right children to verb-token’s left children
9: else
10: add ‘Somebody’ to verb-token’s left children
11: correct verb tense for verb-token
12: return traverse_a_string(root-token)
```

Algorithm 19 Transform() in Coordination Analyzer

```
1: procedure COORDINATION TRANSFORM PROCEDURE(sents)
2: results ← empty list
3: main-root ← root of dependency tree of sents
4: conj-root ← find conj token in dependency tree and their head token main
5: embed all conjugate words of main in a list conj
6: cut conj edge between main and cc and conj
7: if no object for main then
8: try find object in conj’s right children
9: str1 ← traverse_a_string(root)
10: results.append(str1)
11: for conj in conj’s do
12: type ← get_conj_type(main, conj)
13: if type=VERB&VERB then
14: correct part-of-speech tag if necessary
15: if conj has its own subject then
16: new-root ← conj
17: else
18: if main-root then
19: new-root ← conj
20: else
21: replace main with conj in root’s children
22: new-root ← root
23: else
24: main-head ← head of main
25: replace main with conj in main-head’s children
26: new-root ← root
27: str2 ← traverse_a_string(new-root)
28: results.append(str2)
29: return results
```
Algorithm 11 Transform() in Inverted Clausal Subject Analyzer

1: procedure INVERTED_CLAUSAL_SUBJECT_TRANSFORM_PROCEDURE(sents) ▷ Input sentence
2: get access to actual-verb, be-verb and attr in identify procedure
3: change position of actual-verb and attr in be-verb’s children
4: return [traverse_a_string(be-verb)]

Algorithm 12 Transform() in CCOMP Analyzer

1: procedure CLAUSE_COMPONENT_TRANSFORM_PROCEDURE(sents) ▷ Input sentence
2: cut CCOMP link in dependency tree
3: subject ← find ccomp verb’s subject
4: if subject ≠ NULL and subject ≠ DET(e.g. ‘that’) then
5: if original verb do not have object then
6: make subject as original verb’s object
7: else if subject = DET (e.g. ‘that’) then
8: find root verb’s object, substitute ‘that’
9: str1 ← traverse_a_string(ccomp verb)
10: str2 ← traverse_a_string(original verb)
11: return [str1, str2]

Algorithm 13 Transform() in XCOMP Analyzer

1: procedure OPEN_CLAUSAL_COMPONENT_TRANSFORM_PROCEDURE(sents) ▷ Input sentence
2: subject-token ← find_subject(xcomp-verb-token) ▷ find subject of the actual verb
3: results ← empty list
4: if AUX ∉ sents then ▷ for some cases two verbs needs to be output
5: cut xcomp link
6: results.add(traverse_a_string(xcomp-verb-token))
7: remove AUX token in the dependency tree
8: replace xcomp-verb-token in subject’s children with actual-verb-token
9: results.add(traverse_a_string(actual-verb-token))
10: return results

Algorithm 14 Transform() in ACL Analyzer

1: procedure ADJECTIVE_CLAUSE_TRANSFORM_PROCEDURE(sents) ▷ Input Sentence
2: cut acl edge in dependency tree
3: strl ← traverse_a_string(root-token)
4: mid-fix ← empty string
5: if acl-token.tag = VBN and by-token in acl-token’s right children then
6: mid-fix ← ‘be’
7: update acl-verb-token’s left children with [acl-noun, mid-fix, [t ∈ acl-verb-token.lefts where t.dep ∉ [AUX]]
8: correct acl-verb-tense
9: str2 ← traverse_a_string(acl-verb-token)
10: return [str1, str2]

Algorithm 15 Get-Temporal Function at Line 5 in Algorithm 3

1: procedure ADVERBIAL_CLAUSE_TEMPORAL_INFO_EXTRACTION_PROCEDURE(prep – or – mark, cur – temp): The PREP or MARK token in input sentence
2: flag ← False ▷ whether we change the temp
3: if prep-or-mark.type = PREP then
4: sign ← -1
5: else
6: sign ← 1
7: text ← prep-or-mark.text.lower()
8: if text = as then
9: flag ← True
10: else if text ∈ [until, till] then
11: flag ← True
12: cur-temp ← cur-temp + sign
13: else if text = after then
14: flag ← True
15: cur-temp ← cur-temp - sign
16: else if text = before then
17: flag ← True
18: cur-temp ← cur-temp + sign
19: return [flag, cur-temp]
Appendix C: Action Representation Fields

Action Representation Fields (ARFs) in the demo sentence *James gently throws a red ball to Alice in the restaurant from back*, extracted with SRL:

- **owner**: James
- **target**: a red ball
- **prop**: to Alice
- **action**: throw
- **origin**: throws
- **manner**: gently
- **modifier_location**: in the restaurant
- **modifier_direction**: from back

In this case, our output for the prop and target is not correct; they should be swapped. This is one example where this module can introduce errors.

Additional ARFs, extracted heuristically:

- **startTime**: Calculated by current scene time
- **duration**: We have a pre-defined list of words that when appearing in the sentence, they will indicate a short duration (e.g. “run” or “fast”) and therefore the duration will be set to 1 second; in contrast, for words like “slowly” we assign a duration of 4 seconds; otherwise, the duration is 2 seconds.
- **speed**: Similarly to duration, we have pre-defined lists of words that would affect the speed of the pre-baked animation: “angrily” would result in faster movement, but “carefully” in slower movement. We have 3 scales: 0.5, 1, 2 which corresponds to slow, normal and fast.
- **translation**: We have a list of actions which would entail a movement in from one place to another, e.g. “go”. If the value of the action exists in this list, it is set to True, otherwise False.
- **rotation**: If the action exists in our list of verbs that entail rotation, this field is True, otherwise False. Rotation refers to movement in place e.g. “turn” or “sit”.
- **emotion**: We find the closest neighbor of each word in the sentence in list of words that indicate emotion, using word vector similarity. If the similarity exceeds an empirically tested threshold, then we take the corresponding emotion word as the emotion field of this action.
- **partial_start_time**: an important field, since it controls the sequence order of each action. It determines which actions happen in parallel and which happen sequentially. This is still an open question. We solve this problem when doing sentence simplification. Together with the input sentence, current time is also fed into each Analyzer. There are several rules in some of the Analyzers to obtain temporal information. For example, in Line 5 of the Adverbial Clause Analyzer (c.f.3), we assign different temporal sequences for simplified actions. The algorithm is shown in Algorithm 15. The sign together with specific prepositions determines the change direction of current temporal id. In the Coordination Analyzer, the current temporal id changes when it encounters two verbs sharing same subject. Then the later action will get a bigger temporal id.