(1) sounds rather sweeping, and since ∃ inform, not correct? can be better qualified
(2) yes, but what are the benefits of TropICAL? CNL systems exist. It's what TropICAL does with the CNL that matters.

# Chapter 6

# TropICAL: A Language for Story Tropes

This chapter describes the design and implementation of TropICAL (the TROPe Interactive Chronical Action Language), our Domain Specific Language for story tropes. The purpose of this language is to allow for the creation of interactive narratives through the description of tropes in a constrained natural language. The tropes created through the language are designed to be reusable components that can go into a "library", from which story authors can choose the tropes ~~best suited to~~ they wish for the particular story that they are creating.

The motivation for the creation of TropICAL is the lack of any methods of creating interactive narrative that are suitable for non-programmers to use. (1) The systems described in the literature review in chapter 2, such as those using the Mimesis architecture (Young et al., 2004), a drama manager and planner such as in Façade's system (Mateas and Stern, 2003), or linear logic as in Ceptre's system (Martens, 2015), all require the story author to be familiar with planner-based systems or the description of formal logics. (2) The purpose of TropICAL is to greatly reduce the barrier to the creation of interactive stories by allowing authors to describe the components of their story using constrained natural language. In fact, many authors using our system should not even need to write their own tropes using TropICAL, and will instead simply select the tropes that they need for their story from a pre-created "library" of tropes. This process is facilitated through the "StoryBuilder" user interface described in Chapter 7.

The way in which our TropICAL language is used is as follows: wrong verb for what follows

are you describing a workflow?

- It is based around the idea of using *story tropes* as reusable components that can be combined to create stories
- It uses *controlled natural language* to describe these tropes
- The controlled natural language is then parsed and compiled to a data structure that describes the formal features of each trope
- The intermediate data structure is then used to generate InstAL code
- The InstAL code is compiled to AnsProlog, an Answer Set Programming language

these do not concern the non-programmer user?

- The AnsProlog code is then run through *clingo* (Gebser et al., 2011), an answer set solver, to generate all the possible sequences of events that can occur in the story.

This chapter begins by describing TropICAL's constrained natural language syntax in section 6.1. Section 6.2 explores the use cases for our language, deriving a set of requirements from them. Based upon these requirements, the features and design of the language are described in section 6.3. The translation of TropICAL to InstAL code is demonstrated in Section 6.4, with samples of generated InstAL institutions, Answer Set generation (section 6.5), and finally its extension for the description of legal policies (section 6.8). The chapter concludes with a summary in Section 6.9, describing how some requirements from Section 6.2.2 were met, while others are addressed through the *StoryBuilder* tool described in Chapter 7.

what happened to 6.6 and 6.7?

## 6.1 Controlled Natural Language Syntax

TropICAL uses a *Controlled Natural Language* (also referred to as *Constrained Natural Language*) syntax, meaning that it superficially resembles natural English, but is only a subset of the full language.

There are two types of Controlled Natural Language (CNL). The first type are *naturalistic* languages such as ASD Simplified Technical English (ASD-STE100, 2007), designed to be used in documentation for technical industries such as aerospace or defense, or Ogden Basic English (Ogden, 1944), a simplified language for teaching English as a second language. This type of Controlled Natural Language merely describe s a subset of the parent language. The other type of CNL are *formalistic*, with a formal syntax and semantics, which can be mapped to rules in other formal languages such as first-order logic. Attempto Controlled English (Fuchs and Schwitter, 1996) is an example of this formal type of CNL, and the one which forms the basis of our syntax for TropICAL.

### 6.1.1 Attempto Controlled English

Attempto Controlled English (ACE) is a controlled natural language that is also a formal language, and was created by the University of Zurich in 1995. It is still under development, with version 6.7 of the language announced in 2013. ACE has been used in a wide variety of fields, such as software specifications, ontologies, medical documentation and planning.

ACE's vocabulary has three components:

- predefined function words
- predefined fixed phrases (*there is*, *it is false that*, ...)
- content words (nouns, proper nouns, verbs, adjectives, adverbs)

The Attempto Parsing Engine (APE) has a lexicon of content words, and users can define their own content words. User-defined content words take precedence over the built-in lexicon.

ACE's syntax is expressed as a set of construction rules (admissible ACE sentence structures), with syntactically correct sentences described as a set of interpretation rules, which contain the actual semantics of the sentences.

The simplest ACE sentences follow a *subject + verb + complements + adjuncts* structure:

```
A dragon sleeps.
The sky is blue.
A princess owns a castle.
A king gives a sword to a knight.
```

In this structure, every sentence must have a subject and a verb. Sentences that do not contain a verb are expressed with the *there is* structure:

```
There is a kettle.
There are 3 balls.
```

Details can be added to these sentences through the use of adjectives, and the sentences can be joined together with the *and* and *then* keywords.

Other ways of modifying the sentences include *negation*, adding *quantifiers*, or making the sentences *interrogative* or *imperative*. We do not go into further details in this thesis, as we are using ACE simply as an inspiration for our syntax, rather than following its design entirely. For more details, see the "ACE in a Nutshell" page of its documentation for an overview of the language (Fuchs, 2017). → URL?

### 6.1.2 Inform 7

Besides ~~In addition to~~ ACE, the other main inspiration for TropICAL's syntax is the *Inform 7* language.

Inform is a programming language for the creation of Interactive Fiction (also called *text adventure games*). All versions of Inform generate *Z-code*, which is interpreted by the *Z-machine* virtual machine for interactive fiction.

The syntax of the language has changed multiple times since its creation in 1993. The version of the language that we are most interested in is Inform 7 (Reed, 2010), which is its most recent incarnation. Prior to Inform 7, the language used traditional programming models such as procedural and object-oriented paradigms. With version 7, however, its creator Graham Nelson redesigned its syntax completely to allow authors to create their stories using controlled natural language, so that the experience of story creation became closer to that of writing a book.

The simplest possible game authored with Inform 7 would be:

```
"Hello World" by "I.F. Author"

The world is a room.

When play begins, say "Hello, world."
```

The above code simply describes the name of the game ("Hello World"), declares that there is one room in the game, and sets the game to say "Hello World" to the player when the game starts.

An extended example, Will Crowther's cave exploration simulation, would be described like this:

```
"Cave Entrance"

The Cobble Crawl is a room. "You are crawling over cobbles in a low passage.
    There is a dim light at the east end of the passage."

A wicker cage is here. "There is a small wicker cage discarded nearby."

The Debris Room is west of the Crawl. "You are in a debris room filled with
    stuff washed in from the surface. A low wide passage with cobbles becomes
     plugged with mud and debris here, but an awkward canyon leads upward and
     west. A note on the wall says, 'Magic word XYZZY'."

The black rod is here. "A three foot black rod with a rusty star on one end
    lies nearby."

Above the Debris Room is the Sloping E/W Canyon. West of the Canyon is the
    Orange River Chamber.
```

The "Cave Entrance" example above is just one scene from the game. The code begins with a declaration of the "Cobble Crawl" room, followed by the description that appears when the player enters it. The next line declares that a wicker cage is in the room, which is an object that the player can interact with. The sentence after the declaration is the description of the item that appears when the player looks around the room ("There is a small wicker cage discarded nearby."). The "Debris Room" follows the same format described above, with a "black rod" item inside the room. The final line of code simply describes how the rooms are arranged in the game, using words like "above" and "west of".

Although Inform 7 is not based directly on ACE, we can see many of ACE's features in it. For example, it uses the same syntax for simple statements such as *X is a Y*. Most statements in the listing above are pairs of sentences, with the first in each pair declaring the existence of an object or room, and the second being an uninterpreted string that is printed out as the description of the defined object.

what is the evidence for this?

Before version 7, Inform was already widely used as a language for the creation of Interactive Fiction. Given its continued popularity after the switch to a controlled natural language syntax, it appears that the new programming paradigm is successful.

The wide adoption of Inform 7 by story authors with no other programming experience motivates our decision to create a controlled natural language syntax for TropICAL, our own trope-based programming language. The intended audience is the same, so an additional benefit would be that story authors who can code with Inform 7 would also be able to use TropICAL.

① presumably this aspect was introduced earlier? if not it is something of a surprise and the motivation/justification is unclear.
② suggests tighter control than there really is. guides (as later) conveys the right intention.

## 6.2 Requirements

The TropICAL language is designed to be used by authors of interactive narratives, enabling them to easily describe the "rules" of a story through tropes. It is designed to be used in interactive storytelling systems where intelligent software agents are acting out the roles of the characters in the story. These agents, through an architecture such as BDI (Belief, Desire, Intention), have their own goals and plans for how to achieve those goals. TropICAL does not assist in the authoring of these characters and their plans or lines of dialogue. Instead, its purpose is to direct the actions of these "character" agents so that their behaviour follows the rules of the author's story. TropICAL is designed to be used with a system where the character agents have already been authored. It guides them by allowing the characters to know what they are permitted and obliged to do according to the story. This is achieved by giving the characters new percepts so that they may *believe* they are allowed to take certain courses of action. For a description of how this is implemented in practice, see section 5.1 of chapter 5. ①

② sounds rather technical out of context

### 6.2.1 Use Cases

does not need repeating

In order to design a set of tools that is suited to interactive story authors, ~~our intended end users,~~ we derive our set of requirements from specific use cases. These use cases come from the user stories of three users: Alice, Bob and Charlie. In each case, we tell the user's story, highlighting their different levels of technical experience, along with their various needs for and ways of using tools for interactive story creation.

Are these real or hypothetical?

#### Alice

Our first user story focuses on *Alice*, a user with some programming knowledge and specialist expertise in the field of intelligent agents:

> Alice is an interactive narrative researcher using the JASON intelligent agent framework to create the characters of a story. The story is a classic detective noir story, with a private investigator as the main character and several suspects, one of which is the killer. She has coded their behaviour and dialogue as a set of plans, which each agent will follow according to their beliefs, desires and intentions as they "act out" the story.
>
> Alice sets the simulation running, but sees that although the agents are interacting with each other as intended, there is no structure or sense of drama to the story. It merely seems to be one event happening after another.
>
> She decides to use the TropICAL framework to guide the actions of the character agents. The tropes that she chooses to describe her story are the "Three Act Structure", "Chekov's Gun", "Murder Mystery" and "Rooftop Chase" tropes. These tropes are used to not only give the story structure, but also to encourage the

characters in it to behave in a way that makes the simulation "feel" like a murder-mystery genre novel.

When using TropICAL, she selects the "Three Act Structure" and "Chekov's Gun" tropes from a pre-written library of tropes. However, she cannot find tropes that describe what a "Murder Mystery" story is, or what should happen in a "Rooftop Chase" scene. She creates these by writing her own tropes in TropICAL:

```
``Murder Mystery'' is a trope where:
  The Detective is a role
  The Killer is a role
  The Victim is a role
  The Killer kills the Victim
  The Detective investigates the Killer
  Then the Detective catches the Killer
    Or the ``Rooftop Chase'' happens

``Rooftop Chase'' is a trope where:
  The Detective is a role
  The Fugitive is a role
  The Rooftop is a place
  The Detective goes to the Rooftop
  The Fugitive goes to the Rooftop
  The Detective chases the Fugitive
  Then the Detective catches the Fugitive
    Or the Fugitive escapes
```

From Alice's story, we can derive the following requirements:

- She needs a system for describing stories that can be used with a multi-agent system framework such as JASON
- The system needs to direct the behaviour of the agents so that their actions fit within the description of a story
- She needs to be able to write and re-use story components that fit the theme of her story (in this case, "film noir").
- These story components must be able to express both sequences of events and diverging ("branching") events
- She wants to be able to select existing story components from a library or database
- She also wants the ability to create her own story components and save them to the database

use enumeration and labels eg Alice R1 to enable xref.

### Bob

Our second hypothetical user, *Bob*, has no technical expertise at all:

Bob is a traditional story author interested in writing interactive narratives with branching plot lines. He has written a few stories in the style of the "Choose Your Own Adventure" books, using the traditional method of using paper and pen to write down the stories. He implements the story branches by labelling each scene with a name such as "The Hero Sets Off on a Journey", or "The Villain Deceives a Victim", and referring to them by name whenever a branching point occurs. For example, when the Hero of a story is tasked with a quest to complete, the decision could be to go to the "Leave Home and Go on a Journey" scene, or the "Stay at Home and Avoid Adventure" scene, which would be the names of scenes leading either to further choices, or the end of the story.

While he is writing the story, he finds it difficult to keep a mental model of the whole plot, with all its branches, in his head. As he creates new choices and branching points in the story, he finds himself having to pin up notes on a board, and connect them with string. Frustrated, he searches for software to help him with the process of authoring non-linear stories, eventually finding a piece of software called "Twine". Twine's user interface uses the analogy of notes on a pinboard connected with string, the exact process he is used to. He quickly inputs the story he is working on into Twine, and it produces an HTML website with hyperlinks to pages representing the scenes of his story. Next, Bob wants to get more ambitious by abstracting away different parts of his story and re-using them in different places. He wants his story to be heavily quest-driven, with the player being given many different quests and missions over the course of the game. He decides that all of these quests follow a common pattern:

```
The Hero is at Home
Then the Hero meets the Dispatcher
Then the Hero receives a Quest
Then the Hero completes the Quest
  Or the Hero ignores the Quest
```

In this case, the actual content of the "Quest" would change for each individual quest, but the way the player receives and embarks on the Quest remains the same. Bob wants to write this combination of nodes and branches once, and then re-use it as a single node labelled "Quest". Unfortunately, he finds no way to do this using Twine, and he is forced to copy and paste the existing nodes every time he wants to put a Quest into his story.

Bob shares some requirements with Alice:

as above

- He wants to be able to create story components with sequences of events and branching events
- He wants to be able to save and re-use the components (to a database, for example)
- He also wants the ability to create his own story components to add to the database

However, Bob also has some different requirements from Alice:

- He needs to embed tropes inside other tropes (as "sub-tropes")
- He needs to be able to visualise all of the story branches that result when one or more tropes are combined together

### Charlie

Our final user, *Charlie*, has technical expertise, and is also familiar with existing research techniques for interactive narrative generation:

> Charlie is an experienced Interactive Narrative author that wants to automatically generate many of the small narrative details. By using intelligent agents to simulate the characters in the story, she hopes that their intelligence and autonomy will result in some surprising narrative arcs.
>
> First, she authors the character agents using the JASON library for intelligent agents. To do this, she must give them a library of plans to execute in order to carry out certain goals. At any moment in the story, each agent has a set of *beliefs* about its environment, plans that they *desire* to carry out, and plans that they *intend* to execute.
>
> She sets the simulation running, but finds that the characters actions do not follow the structure of a story. She starts out by directing their actions using a planner-based system, but finds that the resulting stories don't make use of the agents' intelligence. The planner is simply telling the agents what to do, based on its goals. She thinks that this is fine most of the time, but nonetheless desires to occasionally see some unexpected (but plausible) actions from the characters. For example, if a character is close to finishing a plan to achieve a goal, she wants them to choose to override what they are told to do by the director planner, and accept some penalty that comes with the violation of the story.
>
> She attempts to write this functionality into the agents' decision-making process, but this turns out to be too time-consuming, and she soon gives up. If only there were a better way.

Charlie's only unique requirement is:

- A way to describe the story such that the agents are told what to do next, but are able to break away from these commands when necessary.

Could add from which use case each requirement derives.

### 6.2.2 Requirement Specification

Based on the use cases above (section 6.2.1), we can create the following requirements:

R1. The software must be able to integrate into a multi-agent framework such as JASON is this distinction intentional?

R2. The system must direct the behaviour of agents to fit a story description

must?

R3. The user should be able to write and re-use existing story components (from a library)

R4. The components must be able to express sequences of events

R5. The components must be able to express branching (diverging) events

R6. The user should have the ability to nest existing components inside new components

R7. The user should be able to visualise the branches of the story that result in the addition or modification of components to the story.

R8. The story must tell the character agents what to do, but they should be free to break away from it in extreme circumstances, to add a degree of unpredictability

The next section describes the design of the TropICAL language, relating each design decision back to these requirements derived from the use cases.

## 6.3 Language Design and Features

In this section, we describe the main features of the TropICAL programming language:

- Entity Declarations (Section 6.3.1)
- Sequences of Events (Section 6.3.2)
- Obligations (Section 6.4.6)
- Branching Events (Section 6.3.4)
- Subtropes (Section 6.3.5)

Each part of this section contains code samples to demonstrate the use and implementation of each language feature. Many of these samples are only extracts from full trope definitions in TropICAL, however. More extensive examples of trope definitions can be found in Appendix A.

This section begins by showing how the entities (roles, objects and places) of a trope are defined by the author (Section 6.3.1), then follows this with a description of how these entities' actions are composed together to form sequences of events (Section 6.3.2). Normally, these events give the character entities *permission* to perform certain actions, so section 6.4.6 shows how an author may describe events that *must* happen in the story through obliging characters to carry out specific actions. In cases where multiple story paths may be taken, TropICAL allows for the description of alternative narrative branches, the syntax for which is described in section 6.3.4. Finally, the means for creating trope abstractions by embedding existing tropes into new ones is demonstrated in section 6.3.5.

### 6.3.1 Entity Declarations

A trope may contain three types of entities that take part in a trope:

- Roles (characters acted out by the agents)
- Objects (items that are used or manipulated by the characters)
- Places (locations that the characters visit)

Before an author can use any characters, objects or places in their tropes, they must first declare them at the top of the trope file. For example, to declare that the *Hero* and *Evil Villain* are character roles, one would write:

description

Listing 6.1: Role declarations

```
The Hero is a role
Evil Villain is a role
```

In TropICAL, the "The" at the beginning of role, object or place names is optional. If the name of an entity begins with "The", the parser omits it from the name. Also, entity names can consist of multiple words, as is the case for the *Evil Villain* character. However, each word in a name must always start with a capital letter.

Similarly, to declare that our trope contains *Sword* and *Magic Potion* objects, the author would write:

Listing 6.2: Object declarations

```
The Sword is an object
The Magic Potion is an object
```

Finally, *Home* and *The Land of Adventure* locations in the trope are declared in the same way:

Listing 6.3: Place declarations

```
Home is a place
The Land of Adventure is a place
```

Once the author has declared the entities of their trope at the top of the trope definition, they are able to use them inside the events of the trope itself.

#### Entity Instances

In cases where multiple characters fulfil the same role (a story could have two hero characters, for example), or there are multiple objects or locations of the same type, these separate entity instances can be passed to the compiler in separate files. An example with multiple Heroes, Swords and Evil Lairs would be:

this risks mixing technicality with the language description and distracting from the latter

Listing 6.4: Entity instances

```
Harry Potter is a Hero
```

① this mixes getting ahead of where you are: mixes semantics with implementation. At this point it would be sufficient to say that it means the event can happen, but not that it must. And perhaps refer forward to the ~~comp~~ actual compilation.

```
Luke Skywalker is a Hero
Excalibur is a Sword
Oathbreaker is a Sword
The Volcano is an Evil Lair
The Secret Cave is an Evil Lair
The Underground Bunker is an Evil Lair
```

not clear (at this stage) what this means.

If these instances are not input into the compiler, then each entity has just one instance with the same name as its type (a hero called "Hero", a sword called "Sword", an evil lair called "Evil Lair", for example).

### 6.3.2 Sequences of Events

TropICAL is an event-based language: it describes events that happen as part of a story, where the events involve actions taken by roles (the agents in a multi-agent system) interacting with other roles as well as objects and places. The simplest possible statement in the language, given the declarations above, would be to write:

Listing 6.5: An event

```
The Hero goes to the Land of Adventure
```

This specifies that the first event that occurs in the trope is that the *Hero* character goes to a place called *The Land of Adventure*. Due to the fact that this statement is compiled to a norm, it states only what is possible in the story, giving *permission* for events to happen, not mandating that they occur. We do this to fulfil requirement R8, so that we guide the actions of character agents to fit our story, rather than regimenting their behaviour. It is possible to tell the agents what to do in a stronger fashion using *obligations*, which we describe in section 6.4.6. ① emph

Typically, tropes tend to consist of a sequence of several events, as is stated in requirement R4. So we can extend this trope further to meet this requirement:

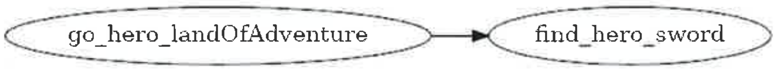
to show how TropICAL expresses event sequences in order to meet this requirement.

Listing 6.6: Two consecutive events

```
The Hero goes to the Land of Adventure
Then the Hero finds the Sword
```

This series of events can be visualised graphically (using a prefix notation for the verbs and objects) thus:

Figure 6-1: Two consecutive events



The *Then* keyword denotes that the event is the next stage in a sequence of events. However, this keyword is ~~syntactical~~ syntactic sugar, and can be omitted ~~with~~ for the same effect:

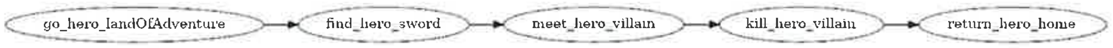
Listing 6.7: "The" instead of "Then"

```
The Hero goes to the Land of Adventure
The Hero finds the Sword
```

Events can be strung together in a sequence that is as long as the author needs:

Listing 6.8: A sequence of events

```
The Hero goes to the Land of Adventure
Then the Hero finds the Sword
Then the Hero meets the Villain
Then the Hero kills the Villain
Then the Hero returns Home
```



goes, but say why

The author can use any verb they wish to describe an event. The verbs are stemmed using *WordNet* (Miller, 1995) so that, for example, *goes* becomes *go* and *wanted* becomes *want* when the tropes are compiled.

### 6.3.3 Obligations

All of the code examples ~~until~~ up to this point express events that *may* occur in the story. When compiled to InstAL, and expressed in terms of norms, they describe the *permitted* behaviour of the agents in a story. This way, the character agents are aware of the actions available to them that follow the path of the author's narrative. However, there are other actions that an author may want to strongly encourage an agent to take, so that they can direct their actions with more control (as per requirement R2). The syntax for this type of action is expressed with the *must* keyword, and is shown in listing 6.9. In this case, the first action (*The Hero goes Home*) is expressed as a permission, but the second action (*Then the Hero must go to the Land of Adventure*) is an obligation.

Listing 6.9: An obliged event within a trope

```
The Hero goes Home
Then the Hero must go to the Land of Adventure
```

#### Deadline Events and Consequences

Obligations may have optional deadline events and consequences. The consequence event is triggered if the obligation has not been fulfilled before the deadline event has occurred. Listing 6.10 shows an example trope where the Hero is obliged to go to the Land of Adventure before the Villain character kills the Mentor character. If the Villain kills the Mentor and the Hero has not yet gone to the Land of Adventure, then a possibility opens up for the Villain to kill the Hero in the story.

Listing 6.10: An obliged event with a deadline and a consequence

```
The Hero must go to the Land of Adventure before the Villain kills the Mentor
  Otherwise, the Villain kills the Hero
```

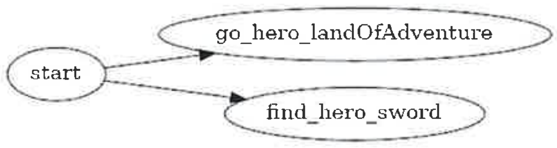
This mechanism could be used as a method of creating branches in the story. If the specified event happens before the deadline event, the path of the trope goes one way. If it doesn't, the story follows the path of the consequence event. Section 6.4.6 explains in detail how obligations, deadlines and consequences are compiled to InstAL code.

## 6.3.4 Branching Events

As requirement R 5 states, a trope author may want to describe alternative possibilities in their story, where either the player or a character in the story may choose ~~decide~~ from one of multiple actions to take. This is implemented using the *Or* keyword in TropICAL, along with a single indentation level of two spaces to indicate its dependency on the preceding statement?:

Listing 6.11: Two branches

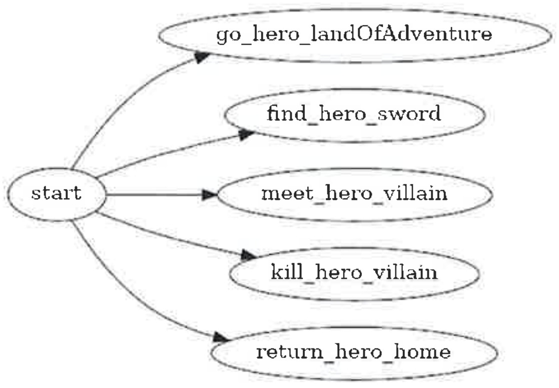
```
The Hero goes to the Land of Adventure
  Or Hero finds the Sword
```



The above code describes two alternative events: either the Hero can go to the Land of Adventure, or the Hero can find the Sword. Multiple events can be chained together on the same level of indentation to create multiple possible alternatives:

Listing 6.12: Five branches

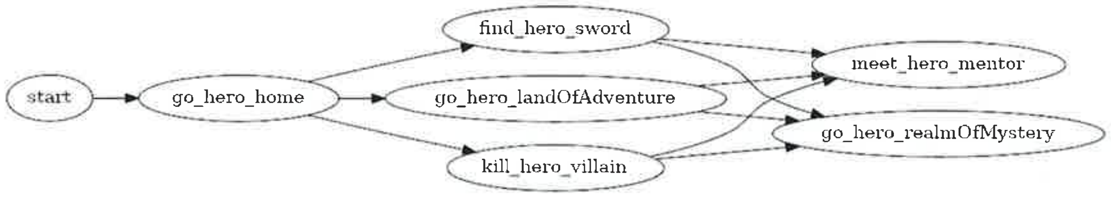
```
The Hero goes to the Land of Adventure
  Or the Hero finds the Sword
  Or the Hero meets the Villain
  Or the Hero kills the Villain
  Or the Hero returns Home
```



In the above example, the trope begins with five different possible alternative events, representing five paths through the story. These branching events can be combined with the event sequences described previously to create more complex tropes:

Listing 6.13: A combination of branches and sequences

```
The Hero goes Home
Then the Hero finds a Sword
  Or the Hero goes to the Land of Adventure
  Or the Hero kills the Villain
Then the Hero meets the Mentor
  Or the Hero goes to the Realm of Mystery
```



In the previous example, only one event occurs in each branching story path before merging back into the main course of the story. In order to extend each story path further, one must indent an extra level: we use a further level of indentation

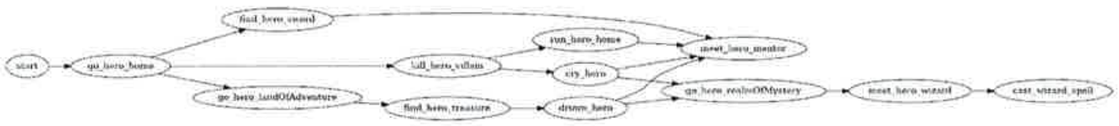
Matt: How can we improve this graph layout?

Seems OK to me. What do you want?

Listing 6.14: Extending branches

```
The Hero goes Home
Then the Hero finds a Sword
  Or the Hero goes to the Land of Adventure
    Then the Hero finds a Treasure
    Then the Hero drowns
  Or the Hero kills the Villain
    Then the Hero runs Home
      Or the Hero cries
Then the Hero meets the Mentor
```

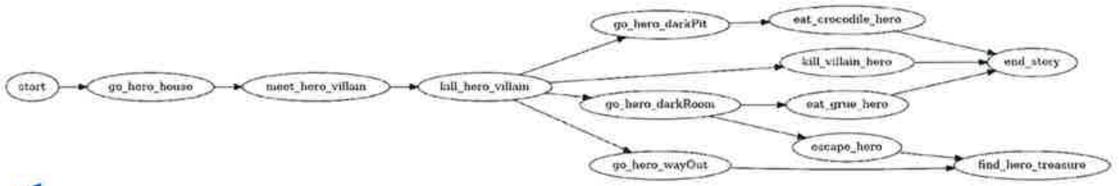
```
  Or the Hero goes to the Realm of Mystery
    Then the Hero meets the Wizard
    Then the Wizard casts a Spell
```



It should be noted that all branches will merge back into the "main" story path (the bottom level of indentation) once they have completed. To prevent this from happening, and terminate the story, the author may write *"Then the Story ends"*. In this example, the story will always end unless the Hero goes to the Way Out or escapes the Dark Room:

Listing 6.15: Terminating branches

```
The Hero goes to the House
Then the Hero meets the Villain
Then the Hero kills the Villain
  Or the Villain kills the Hero
    Then the Story ends
Then the Hero goes to the Way Out
  Or the Hero goes to the Dark Pit
    Then the Crocodile eats the Hero
    Then the Story ends
  Or the Hero goes to the Dark Room
    Then the Hero escapes
      Or the Grue eats the Hero
        Then the Story ends
Then the Hero finds the Treasure
```



is code the right word to use to refer to the narrative?

Using this technique, branches can be extended indefinitely through increasing levels of indentation. Deeply indented code is often undesirable, however, and is usually a symptom that the code needs to be subdivided into modules. We can achieve this by embedding subtropes inside ~~of~~ other tropes.

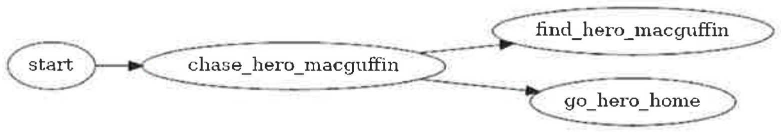
### 6.3.5 Subtropes

Requirement R6 states that we need a mechanism for the nesting of tropes, so that we may create new abstractions by combining ~~old~~ existing tropes ~~into~~ to create new ones. We do this through *Subtropes*, which are simply previously-written tropes that are embedded inside a new trope. Take this simple trope example:

Listing 6.16: Subtrope te be embedded

```
"Item Search" is a trope where:
    The Macguffin is an object
    The Hero is a role
    Home is a place

    The Hero chases the Macguffin
    Then the Hero finds the Macguffin
      Or the Hero goes Home
```

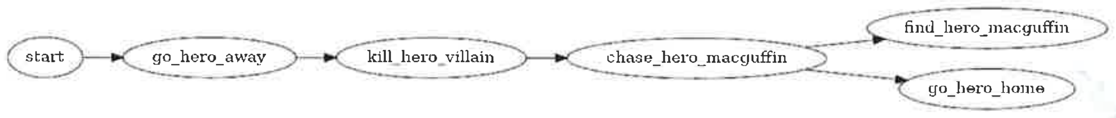


To embed this trope inside another trope, we refer to it by name by writing *Then the "Item Search" trope happens.* It is important to put the name of the trope inside quotation marks. An example of putting this trope at the end of another trope is shown in the following example: say why

Listing 6.17: Trope containing a subtrope

```
"Kill then Search" is a trope where:
   Away is a place
   The Hero is a role
   The Villain is a role

   The Hero goes Away
   Then the Hero kills the Villain
   Then the "Item Search" trope happens
```

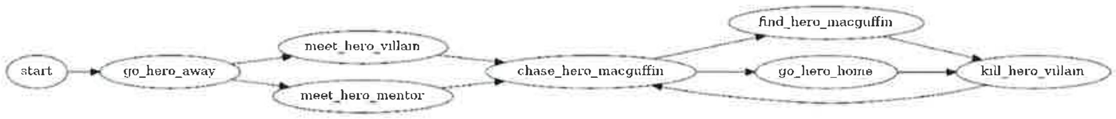


The "Item Search" subtrope is thus embedded inside the "Kill then Search" trope, with all of the events and branches of "Item Search" being appended to the end of "Kill then Search". Rather than only being embedded at the end of a trope however, subtropes can appear at any point in a trope, or even at several points in a trope. Take this example that uses the "Item Search" trope inside nested branches of its story:

Listing 6.18: Subtrope in multiple places

```
"Futile Search" is a trope where:
  The Hero is a role
  The Villain is a role
  The Mentor is a role
  Away is a place
  Home is a place
  The Land of Adventure is a place
  The Realm of Mystery is a place

  The Hero goes Away
  Then the Hero meets the Villain
    Or the Hero meets the Mentor
      Then the "Item Search" trope happens
  Then the Hero kills the Villain
  Then the "Item Search" trope happens
```



An author can create complex branching narrative by combining these tropes together. To see visualisations of the story paths of multiple tropes combined, see section 7.6 of chapter 7.

Matt: This is a good demonstration of why a graph won't work as a visualisation: we don't want *kill_hero_villain* to happen after the second time the *Item Search* trope happens, just the first time. Will redo as a tree visualisation instead.

For technical details of the parser implementation, including the EBNF grammar of the TropICAL language, refer to Appendix B.3. Appendix B.4 shows visualisations of the parse tree output for some of the code examples described in this section.

## 6.4 InstAL Code Generation

Once the tropes written in TropICAL have been parsed and converted into an intermediate hash map, this data structure is then used to generate InstAL code. This section describes how and why specific language features translate to InstAL, with fully translated examples of InstAL code appearing in Appendix B.

As described in section 3.3.1, InstAL features three types of events: *external*, *institutional* and *violation*. We begin by examining the translation of trope events to institutional events.

### 6.4.1 Initial Conditions

The *initially* clause in InstAL specifies the fluents that initially hold when the institution is effected. Role, object and place declarations are translated into fluent declarations as part of this clause. For example, for a trope with the following declarations:

```
The Hero is a role
The Sword is an object
The Land of Adventure is a place
```

And the following entity instances:

```
Luke Skywalker is a Hero
The Lightsaber is a Sword
The Death Star is a Land of Adventure
```

The following InstAL code is produced, with type and fluent declarations automatically appearing at the top of the file:

```
type Agent;
type Role;
type Place;
type PlaceName;
type Object;
type ObjectName;

fluent role(Agent, Role);
fluent place(PlaceName, Place);
fluent object(ObjectName, Object);

initially:
  role(lukeSkywalker, hero),
  object(lightSaber, sword),
  place(deathStar, landOfAdventure);
```

The *initially* clause also specifies the permitted and obliged events that occur at the start of the trope. Say our trope contains a series of events (as in listing 6.8):

```
The Hero goes to the Land of Adventure
Then the Hero finds the Sword
Then the Hero meets the Villain
Then the Hero kills the Villain
Then the Hero returns Home
```

In this case, the first event of the trope (*The Hero goes to the Land of Adventure*) must be permitted to happen at the very beginning of the institution, inside the *initially* clause. In combination with the above entity declarations, this results in the following code being generated (omitting the type and fluent declarations this time):

```
initially:
  perm(go(X, Y)) if role(X, hero), place(Y, landOfAdventure),
  role(lukeSkywalker, hero),
  object(lightSaber, sword),
  place(deathStar, landOfAdventure);
```

This means that the Luke Skywalker character is able to go to the Death Star at the beginning of the institution, as is specified in the trope.

### 6.4.2 Generation

Tropes usually consist of multiple events, so it is not enough to have one event permitted at the start of a trope. Once this first event has occurred, the next event in the sequence must unless this has been introduced already the reader is not going to follow

have permission to happen.

The institution watches for events to happen as *external* events in the environment, and then has these events generate *institutional* events that occur within the institution and create permissions or obligations for further external events to occur. Returning to listing 6.8 above, the external events that happen are:

- The Hero finds the Sword
- The Hero meets the Villain
- The Hero kills the Villain
- The Hero returns Home

These are translated into *generates* statements in InstAL so that they generate the required institutional events: no listing reference

```
find(R, T) generates
    intSequence3(R, S, T, U, V) if
        role(R, hero),
        object(T, sword);
kill(R, S) generates
    intSequence3(R, S, T, U, V) if
        role(S, villain),
        role(R, hero);
meet(R, S) generates
    intSequence3(R, S, T, U, V) if
        role(S, villain),
        role(R, hero);
go(R, V) generates
    intSequence3(R, S, T, U, V) if
        role(R, hero),
        place(V, landOfAdventure);
return(R, U) generates
    intSequence3(R, S, T, U, V) if
        role(R, hero),
        place(U, home);
```

need to explain compilation strategy so that examples make sense. intSequence3 is clearly important as is the use of conditions

same comment as for 6.19 and 6.20

(1) Each institution only has one institutional event that is triggered by the external events, which is named after the trope itself (in this case, the event is named *intSequence3* after the trope name "Sequence 3"). The permissions and obligations that this institutional event initiates depend on the *phase* of the trope that is currently active (see section 6.4.3) below.

### 6.4.3 Initiation

External events generate an internal event for the institution, which permits the next event or events in a sequence to occur. To ensure that the events of the trope are permitted or obliged to occur one after another, we use a mechanism called *event phases.*

(1) reader does not ~~[illegible]~~ know there are several. This underlines the point above about explaining the strategy first.

### Event Phases

An *event phase* is a fluent that holds the state of the current institution. At the beginning of the trope, its state is simply *active*. This is represented in the institution through the fluent *phase(tropeName, active)*. Once the first event has occurred, the trope enters *phase A*, which means that the fluent *phase(tropeName, phaseA)* then holds, and the *phase(tropeName, active* fluent is terminated (this is described in section 6.4.4). This initiation and termination process then repeats through all phases of the trope, through phases B, C and D if they exist, until the final event occurs and the last phase is terminated.

this is also part of the target model which needs explains or at least outlining in advance. It is too difficult for the reader to 'join the bits up' as they emerge.

In the case of our example sequence of events from listing 6.8, there are five phases in total:

- The Hero goes to the Land of Adventure (active)
- The Hero finds the Sword (phase A)
- The Hero meets the Villain (phase B)
- The Hero kills the Villain (phase C)
- The Hero returns Home (phase D)

The events and phases of this trope are initiated in InstAL through the following generated code:

Listing 6.19: Institutional event initiation code for listing 6.8

```
intSequence3(R, S, T, U, V) initiates
    phase(sequence3, phaseA),
    perm(find(R, T)) if
        phase(sequence3, active),
        role(R, hero),
        object(T, sword);
intSequence3(R, S, T, U, V) initiates
    phase(sequence3, phaseB),
    perm(meet(R, S)) if
        phase(sequence3, phaseA),
        role(S, villain),
        role(R, hero);
intSequence3(R, S, T, U, V) initiates
    phase(sequence3, phaseC),
    perm(kill(R, S)) if
        phase(sequence3, phaseB),
        role(S, villain),
        role(R, hero);
intSequence3(R, S, T, U, V) initiates
    phase(sequence3, phaseD),
    perm(return(R, U)) if
        phase(sequence3, phaseC),
        role(R, hero),
        place(U, home);
```

needs more explanation of the framework into which the story facts are slotted.

### 6.4.4 Termination

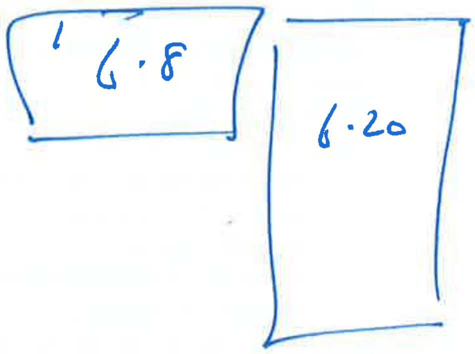
The generated code that terminates the phases and permissions corresponding to those initiated in listing 6.19 is shown in listing 6.20 below. Once an external event occurs, its permission to occur again is terminated along with the phase in which it occurred:

Listing 6.20: Institutional event termination code for listing 6.8

```
intSequence3(R, S, T, U, V) terminates
    phase(sequence3, active),
    perm(go(R, V)) if
        phase(sequence3, active),
        role(R, hero),
        place(V, landOfAdventure);
intSequence3(R, S, T, U, V) terminates
    phase(sequence3, phaseA),
    perm(find(R, T)) if
        phase(sequence3, phaseA),
        role(R, hero),
        object(T, sword);
intSequence3(R, S, T, U, V) terminates
    phase(sequence3, phaseB),
    perm(meet(R, S)) if
        phase(sequence3, phaseB),
        role(S, villain),
        role(R, hero);
intSequence3(R, S, T, U, V) terminates
    phase(sequence3, phaseC),
    perm(kill(R, S)) if
        phase(sequence3, phaseC),
        role(S, villain),
        role(R, hero);
intSequence3(R, S, T, U, V) terminates
    phase(sequence3, phaseD),
    perm(return(R, U)) if
        phase(sequence3, phaseD),
        role(R, hero),
        place(U, home);
```

as above also suggests reproducing 6.8 here rather than referring back.

Could do



using multicolumn or minipage

### 6.4.5 Branches

To represent branching events in InstAL, multiple events are permitted to occur during one phase. Take as an example the combination of event sequences and branches shown in listing 6.13:

```
The Hero goes Home
Then the Hero finds a Sword
  Or the Hero goes to the Land of Adventure
  Or the Hero kills the Villain
Then the Hero meets the Mentor
  Or the Hero goes to the Realm of Mystery
```

① Say what, but not why or how
As earlier lacks explanation
of compilation strategy.
How are you
using InstAL semantics
for your purposes.
where is the rest of the sentence?

When compiled with the following entity instances:

```
Harry Potter is a Hero
Voldemort is a Villain
Dumbledore is a Mentor
The Gryffindor Sword is a Sword
Hogwarts is a Land of Adventure
The Chamber of Secrets is a Realm of Mystery
```

The full institution appears in Section B.2 of Appendix B.

① The branching events are implemented in the *initiates* and *terminates* clauses. In listing 6.21 below, we see that three possible events are given permission to occur as part of Phase A: *The Hero finds a Sword*, *The Hero goes to the Land of Adventure*, and *The Hero kills the Villain*. In Phase B, two events corresponding to the possible branches are given permission to occur: *The Hero meets the Mentor* and *The Hero goes to the Realm of Mystery*:

Listing 6.21: Initiation events for the branching trope in listing 6.13

```
intBranch3(R, S, T, U, V, W, X) initiates
    phase(branch3, phaseA),
    perm(find(R, U)),
    perm(go(R, X)),
    perm(kill(R, S)) if
        phase(branch3, active),
        object(U, sword),
        role(R, hero),
        place(X, landOfAdventure),
        role(S, villain);
intBranch3(R, S, T, U, V, W, X) initiates
    phase(branch3, phaseB),
    perm(meet(R, T)),
    perm(go(R, V)) if
        phase(branch3, phaseA),
        place(V, realmOfMystery),
        role(R, hero),
        role(T, mentor);
```

explain intBranch3

where is this?

This means that during Phase A, there are three possible narrative branches, which get closed off when any one of them occurs through the corresponding termination event in the institution.

## 6.4.6 Obligations

### Obligations Without Deadlines or Consequences

In our tropes, obligations have optional deadline events and consequences. At the time of implementation, both of these were mandatory in InstAL, and so it was necessary to add "dummy" events for both if they were not specified in TropICAL. Returning to listing 6.9 of section , which specifies an obligation without a deadline or a consequence:

```
The Hero goes Home
Then the Hero must go to the Land of Adventure
```

Line 3 of listing 6.23 shows the initiation rule for an obligation event with no deadline or consequence, and listing 6.22 shows the fluent declaration for the obligation fluent. The *intNoDeadline* and *noViolation* events are dummy events that never occur either inside or outside of the institution. There are there only because InstAL always requires deadline and consequence events in obligation fluents. The first parameter of the obligation fluent specifies an institutional event that must occur before the deadline. In this case, the event is *intGo(hero, landOfAdventure)*. Listing 6.24 shows how the *go(hero, landOfAdventure)* external event generates the *intGo(hero, landOfAdventure* institutional event. Due to the fact that it is this external event that triggers the obliged institutional event, it is given permission to occur on line 3 of listing 6.23, along with the obliged institutional event itself.

Listing 6.22: Fluent declaration for the obligation event in listing 6.9

```
obligation fluent obl(intGo(Agent, PlaceName), intNoDeadline, noViolation);
```

Listing 6.23: Initiation rule for the second (obligation) event of the trope in listing 6.9

```
1  intObligation1(R, S, T) initiates
2      phase(obligation1, phaseA),
3      obl(intGo(R,T), intNoDeadline, noViolation), perm(go(R, T)), perm(intGo(R
          ,T)), pow(intGo(R,T)) if
4          phase(obligation1, active),
5          role(R, hero),
6          place(T, landOfAdventure);
```

Listing 6.24: Generation rule for the obligation event

```
go(R, S) generates
    intGo(R,S) if
        role(R, hero),
        place(S, landOfAdventure);
```

reminds me that I did not see anything explaining ~~[illegible]~~ go vs intGo etc.

The full InstAL institution that this generates appears in listing B.1 of appendix B.

### Obligations with Deadlines and Consequences

The trope shown in listing 6.25 describes a situation where an obligation with both a deadline and a consequence occurs.

Listing 6.25: Example of a trope containing an obligation with both a deadline and a consequence

```
The Hero must go to the Land of Adventure before the Villain kills the Mentor
 Otherwise, the Villain may kill the Hero
```

Listing 6.26 shows the fluent declaration of the obligation event in listing 6.25. The deadline event *intKill(villain, mentor)* is an institutional event, and so is generated by the *kill(villain,*

this may be lost on the reader without the necessary background *mentor)* external event on lines 5 to 8 of listing 6.29. Listing 6.27 shows the obligation fluent itself. As this obligation is the first event to occur in this trope, it appears in the *initially* clause of the InstAL code, rather than as an initiation event.

If the violation event contained within this obligation were to occur, it would enable the story to begin another course of events. In this case, it initiates the permission of the villain to kill the hero. This is shown in the code of listing 6.25.

Listing 6.26: Fluent declaration for an obligation fluent with both deadline and consequence events

```
obligation fluent obl(intGo(Agent, PlaceName), intKill(Agent, Agent),
    violHeroGoLandOfAdventure);
```

Listing 6.27: An obligation fluent with both a deadline and a consequence, translated from the trope in listing 6.25

```
obl(intGo(R,U), intKill(S,T), violHeroGoLandOfAdventure), perm(go(R, U)),
    perm(intGo(R,U)), pow(intGo(R,U)) if role(R, hero), place(U,
    landOfAdventure);
```

Listing 6.28: Permissions generated by the violation event of the trope in listing 6.25

```
violHeroGoLandOfAdventure initiates
    perm(kill(R, S)) if
        role(R, villain),
        role(S, hero);
```

where does the reader understand that R is a variable and that unification is taking place?

Listing 6.29: Generation events for the trope in listing 6.25

```
1 go(R, U) generates
2     intGo(R,U) if
3         role(R, hero),
4         place(U, landOfAdventure);
5 kill(S, T) generates
6     intKill(S,T) if
7         role(S, villain),
8         role(T, mentor);
```

The full InstAL code listing for this institution appears in listing B.2 of appendix B.

(1) Obligations work best when both a deadline and consequence are specified. Without any consequence for its violation, an agent has no real obligation to carry out an action, and it serves the same role as a permission. For this reason, it would be worth considering (2) either making both deadline and consequence events mandatory, or implementing some kind of default deadline and consequence. A default deadline could be the next event in a sequence, for example, and a default consequence could be the loss of health of a character, or emotional deterioration in cases where an emotional model is used.

(1) not clear to me what this means or can mean. Obligations work either way.
(2) is this with reference to InstAL semantics? Reasonable points but ~~not sure~~ don't think they are in the right place.

① no, they do not implement conflict resolution, they just connect events/fluents in one institution with another

we're using the term Institutional Bridges and then just bridges in the paper we are writing right now

## 6.4.7 Bridge Institution

In section 6.3.5, we describe how *subtropes* are implemented in InstAL through the embedding of existing tropes inside of new ones. This is translated into InstAL code through the use of a *Bridge Institution*, a separate institution which is used to link two other institutions together. The concept of a bridge institution is developed by Li (2014) as a means of conflict resolution in interacting institutions, and involves the creation of an intermediary institution to coordinate events between two other institutions. The two institutions linked are referred to as the *source* and *sink* institutions, where an event that occurs in the source institution has the power to generate an event inside of the sink institution. ①

but this is right

The *Item Search* and *Kill then Search* tropes described in listings 6.16 and 6.17 can be connected ~~described~~ as the *sink* and *source* of a bridge ~~institution,~~ respectively. In this case, the first event of the sink institution (the *Item Search* trope) is given permission to occur when the final event of the source institution (the *Kill then Search* trope) happens.

Listing 6.30: Bridge ~~institution~~ for the *Item Search* and *Kill then Search* tropes

```
1   bridge killThenSearchItemSearch;
2
3   source killThenSearch;
4   sink itemSearch;
5
6   cross fluent ipow(killThenSearch, perm(chase(Agent, ObjectName)), itemSearch)
        ;
7   cross fluent ipow(killThenSearch, phase(Trope, Phase), itemSearch);
8
9   intStartItemSearch xinitiates phase(itemSearch, active);
10  intStartItemSearch xinitiates perm(chase(R, S)) if
11          role(R, hero),
12          object(S, macguffin);
13
14  initially ipow(killThenSearch,  perm(chase(R, S)), itemSearch),
15      ipow(killThenSearch, phase(itemSearch, active), itemSearch);
```

Listing 6.30 shows the bridge ~~institution~~ that links the *Item Search* and *Kill then Search* institutions together. It starts by defining *killThenSearch* as the source institution, and *itemSearch* as the sink. Then a cross fluent for the first event in the subtrope (which is *itemSearch*, the sink institution) is declared in line 6. This line only states that this is a fluent that is to be initiated in the sink institution, from the source institution.

Lines 10 to 12 describe the actual sequence of events that lead to the *The Hero chases the MacGuffin* event being triggered inside the sink institution: when the institutional event *intStartItemSearch* occurs inside the source institution, it gives the Hero permission to chase the MacGuffin in the sink institution. The same mechanism is used on lines 7 and 9 to declare and initiate the first phase (the *active* phase) of the sink (*Item Search*) institution from the source (*Kill then Search*) institution. The phase is initiated from the same institutional event inside the source institution: the *intStartItemSearch* event.

Listing 6.31: The generation event for the final action in the *Kill then Search* source institution

```
kill(R, S) generates
    intStartItemSearch if
        role(S, villain),
        role(R, hero),
        phase(killNSearch, phaseA);
```

Listing 6.31 shows the generation event corresponding to the final action that occurs in the *Kill then Search* trope: *The Hero kills the Villain.* The line after this event in the trope embeds the *Item Search* subtrope within this trope: *Then the "Item Search" trope happens.* For this reason, the *intStartItemSearch* institutional event is initiated, which further initiates the permission for the first event in the *Item Search* institution to happen through the bridge institution in listing 6.30.

Full listings for the generated InstAL code for these tropes, along with several other examples, appear in Appendix B.

this needs to be clear at the start

## 6.5 Answer Set Generation

Once the code has been translated from TropICAL into InstAL, it can be compiled further into AnsProlog, an Answer Set Programming (ASP) language, using a process described by Cliffe et al. (2007). We use the Potassco project's Clingo (Gebser et al., 2011) solver to formally verify that certain sequences of events conform to the tropes that form our institutions. This solver can also be used to generate all of the possible sequences of events that fit a story described using one or more tropes, by generating all of the *answer sets* that correspond to a set of institutions.

## 6.6 Adding Constraints

When our answer sets (traces) are generated by the answer set solver, we want to see sequences of events that can happen as part of a story with a set of given tropes. However, the answer set solver may include some uninteresting events as part of the grounding process, such as *null* events, the dummy events we used to replace unspecified deadlines and consequences for obligations, and events that otherwise violate the rules in each institution.

Listing 6.32 shows the AnsProlog rules used to constrain the answer sets produced by the solver, so that they only consist of events that are relevant to our stories. Line 1 specifies that events are only valid if they are not violation or null events. Line 2 filters out any event
① that has a *null* institution. Line 3 ignores any answer sets where a valid event occurs after a sequence of non-valid events. Lines 5 and 6 ignore our dummy deadline events for obligations where no deadline is specified. Finally, lines 7 and 8 remove any *null* events at all from the answer sets.

Listing 6.32: Constraint rules to remove invalid and irrelevant events

float to avoid break

① doesn't look like that to me.

```
validEvent(I, In) :- instant(I), inst(In), event(E), occurred(E, In, I), not
    occurred(viol(_), In, I), E != null.
validEvent(I) :- validEvent(I, In), inst(In).
:- validEvent(I2), not validEvent(I), I < I2, instant(I), instant(I2).
:- instant(I), not validEvent(I), occurred(viol(_), In, I).
deadEvent :- observed(noDeadline(X), I, T).
:- deadEvent.
nullInst :- occurred(X, null, T).
:- nullInst.
```

The next section lists example answer set (trace) outputs of the solver with two tropes, both separately and combined.

## 6.7 Example Answer Sets (Traces)

Answer Sets (also called "traces") are produced by the answer set solver when given the institutions compiled as ASP code, along with a set of constraints. The traces generated represent all of the *possible* sequences of events that may occur, given the described institutions. It is worth noting that by default this also includes any event that violate any of the institutions, as these violations are not prevented by the institutions. Instead, *violation events* are generated which carry consequences with them. By using a constraint such as the one in Listing ~~listed in~~ 6.32, we can tell the answer set solver to only generate sequences of events that do not contain these violation events. This is what we have done in this section. For visualisations of these answer sets, refer to the diagrams in section 7.6.

### 6.7.1 Evil Empire

Our first example trope is a simple sequence of events called the "Evil Empire":

Listing 6.33: The "Evil Empire" trope

```
``Evil Empire'' is a trope where:
  The Empire is a role
  The Hero is a role

  The Empire chases the Hero
  Then the Empire captures the Hero
  Then the Hero escapes
```

This trope, once compiled to InstAL, then AnsProlog, and run through Clingo, produces four answer sets. The compiled institution is listed in Appendix B.5, and all of the resulting traces appear in Appendix B.5.1. The listed traces are solved by looking up to three events into the trope, as that is its maximum length. how is that obvious?

explain why

Listing 6.34 shows one of the four traces from the solver output. This trace contains each event of the trope, happening one after another. The other traces stop short of the final event, so that trace one has zero events, trace two has only the first event, and trace three has the first two events of the trope.

Listing 6.34: Example trace for the “Evil Empire” trope

```
Answer Set 4:

Time Step 1:

holdsat(role(hero,hero),evilEmpire)
holdsat(role(empire,empire),evilEmpire)
holdsat(phase(evilEmpire,phaseA),evilEmpire)
holdsat(pow(intEvilEmpire(hero,empire)),evilEmpire)
holdsat(perm(null),evilEmpire)
holdsat(perm(intEvilEmpire(hero,empire)),evilEmpire)
holdsat(perm(capture(hero,empire)),evilEmpire)
holdsat(live(evilEmpire),evilEmpire)
occurred(intEvilEmpire(hero,empire),evilEmpire)
occurred(chase(hero,empire),evilEmpire)


Time Step 2:

holdsat(role(hero,hero),evilEmpire)
holdsat(role(empire,empire),evilEmpire)
holdsat(phase(evilEmpire,phaseB),evilEmpire)
holdsat(pow(intEvilEmpire(hero,empire)),evilEmpire)
holdsat(perm(null),evilEmpire)
holdsat(perm(intEvilEmpire(hero,empire)),evilEmpire)
holdsat(perm(escape(hero)),evilEmpire)
holdsat(live(evilEmpire),evilEmpire)
occurred(intEvilEmpire(hero,empire),evilEmpire)
occurred(capture(hero,empire),evilEmpire)


Time Step 3:

holdsat(role(hero,hero),evilEmpire)
holdsat(role(empire,empire),evilEmpire)
holdsat(pow(intEvilEmpire(hero,empire)),evilEmpire)
holdsat(perm(null),evilEmpire)
holdsat(perm(intEvilEmpire(hero,empire)),evilEmpire)
holdsat(live(evilEmpire),evilEmpire)
occurred(intEvilEmpire(hero,empire),evilEmpire)
occurred(escape(hero),evilEmpire)
```

think this would be better in more human friendly output format.

( what is the purpose of this?

use line numbers to refer

The events that appear at the start of the listing (starting with the *holdsat* predicate) describe the *fluents* that hold for a certain institution at that particular time step in the answer set. After these statements, the events that have *occurred* in that institution appear. These events can be both institutional or external events, which can be distinguished due to the institutional event names beginning with “int” (such as *intEvilEmpire*, for example). External events usually trigger institutional events with the institution name, which is why the *chase*, *capture* and *escape* events all trigger the *intEvilEmpire* institutional event in listing 6.34.

### 6.7.2 The Hero's Journey

```
``The Hero's Journey'' is a trope where:
  The Hero is a role
  The Villain is a role
  Home is a place
  The Evil Lair is a place

  The Hero goes to the Evil Lair
  Then the Hero kills the Villain
    Or the Villain escapes
  Then the Hero goes Home
```

needs to be clearer re # answer sets

The answer set solver produces six traces for this trope. The branching alternatives (the Hero can either kill the Villain or the Villain can escape at one point in the trope) add an extra three answer sets to the output. If this trope were just three events long, without the branching alternatives, then only three answer sets would be produced (corresponding with stories of one, two and three events long). The fourth answer set produced is shown in listing 6.35 as an example.

Listing 6.35: Example trace for the "Hero's Journey" trope

Otherwise

```
Answer Set 4:

Time Step 1:

holdsat(role(villain,villain),herosJourney)
holdsat(role(hero,hero),herosJourney)
holdsat(place(home,home),herosJourney)
holdsat(place(evilLair,evilLair),herosJourney)
holdsat(phase(herosJourney,phaseA),herosJourney)
holdsat(pow(intHerosJourney(hero,villain,evilLair,home)),herosJourney)
holdsat(perm(null),herosJourney)
holdsat(perm(kill(hero,villain)),herosJourney)
holdsat(perm(intHerosJourney(hero,villain,evilLair,home)),herosJourney)
holdsat(perm(escape(villain)),herosJourney)
holdsat(live(herosJourney),herosJourney)
occurred(intHerosJourney(hero,villain,evilLair,home),herosJourney)
occurred(go(hero,evilLair),herosJourney)


Time Step 2:

holdsat(role(villain,villain),herosJourney)
holdsat(role(hero,hero),herosJourney)
holdsat(place(home,home),herosJourney)
holdsat(place(evilLair,evilLair),herosJourney)
holdsat(phase(herosJourney,phaseB),herosJourney)
holdsat(pow(intHerosJourney(hero,villain,evilLair,home)),herosJourney)
holdsat(perm(null),herosJourney)
holdsat(perm(intHerosJourney(hero,villain,evilLair,home)),herosJourney)
```

(1) But what does it mean to 'combine' the two tropes

```
holdsat(perm(go(hero,home)),herosJourney)
holdsat(live(herosJourney),herosJourney)
occurred(kill(hero,villain),herosJourney)
occurred(intHerosJourney(hero,villain,evilLair,home),herosJourney)


Time Step 3:

holdsat(role(villain,villain),herosJourney)
holdsat(role(hero,hero),herosJourney)
holdsat(place(home,home),herosJourney)
holdsat(place(evilLair,evilLair),herosJourney)
holdsat(pow(intHerosJourney(hero,villain,evilLair,home)),herosJourney)
holdsat(perm(null),herosJourney)
holdsat(perm(intHerosJourney(hero,villain,evilLair,home)),herosJourney)
holdsat(live(herosJourney),herosJourney)
occurred(intHerosJourney(hero,villain,evilLair,home),herosJourney)
occurred(go(hero,home),herosJourney)
```

So far, the *Evil Empire* and *Hero's Journey* tropes produce unremarkable results. Due to their simple natures, they only produce a small number of answer sets when run through a solver. The results are more interesting when both tropes are input into the solver, however.

### 6.7.3 The Hero's Journey *and* The Evil Empire

(1)

Using both institutions as inputs into the solver, combining the two tropes ~~together~~, produces 124 answer sets. Again, the reason that the solver produces so many answer sets is that it includes stories that are less than the full length of every event from both tropes combined (which is six events in this case). This is desirable when we are combining tropes, as sometimes we need to incorporate events from a trope without needing it to run its full course. In this example, if the Hero's Journey has come to an end, then that would seem like a more suitable place to finish a story, rather than waiting for all of the events in the Evil Empire trope to occur.

seems written from an ~~[illegible]~~ internal perspective: what does the reader need to understand about this.

An example of a trace that runs the full six event length of both tropes put together is shown in listing 6.36.

Listing 6.36: Example trace for both the "Evil Empire" and "Hero's Journey" tropes combined together

```
Answer Set 70:

Time Step 1:

holdsat(role(villain,villain),herosJourney)
holdsat(role(hero,hero),herosJourney)
holdsat(role(empire,empire),herosJourney)
holdsat(place(home,home),herosJourney)
holdsat(place(evilLair,evilLair),herosJourney)
holdsat(phase(herosJourney,phaseA),herosJourney)
```

(Listing is not at all informative

```
holdsat(pow(intHerosJourney(hero,villain,evilLair,home)),herosJourney)
holdsat(perm(null),herosJourney)
holdsat(perm(kill(hero,villain)),herosJourney)
holdsat(perm(intHerosJourney(hero,villain,evilLair,home)),herosJourney)
holdsat(perm(escape(villain)),herosJourney)
holdsat(live(herosJourney),herosJourney)
holdsat(role(villain,villain),evilEmpire)
holdsat(role(hero,hero),evilEmpire)
holdsat(role(empire,empire),evilEmpire)
holdsat(place(home,home),evilEmpire)
holdsat(place(evilLair,evilLair),evilEmpire)
holdsat(phase(evilEmpire,active),evilEmpire)
holdsat(pow(intEvilEmpire(hero,empire)),evilEmpire)
holdsat(perm(null),evilEmpire)
holdsat(perm(intEvilEmpire(hero,empire)),evilEmpire)
holdsat(perm(chase(hero,empire)),evilEmpire)
holdsat(live(evilEmpire),evilEmpire)
occurred(intHerosJourney(hero,villain,evilLair,home),herosJourney)
occurred(go(hero,evilLair),herosJourney)
occurred(null,evilEmpire)


Time Step 2:

holdsat(role(villain,villain),herosJourney)
holdsat(role(hero,hero),herosJourney)
holdsat(role(empire,empire),herosJourney)
holdsat(place(home,home),herosJourney)
holdsat(place(evilLair,evilLair),herosJourney)
holdsat(phase(herosJourney,phaseA),herosJourney)
holdsat(pow(intHerosJourney(hero,villain,evilLair,home)),herosJourney)
holdsat(perm(null),herosJourney)
holdsat(perm(kill(hero,villain)),herosJourney)
holdsat(perm(intHerosJourney(hero,villain,evilLair,home)),herosJourney)
holdsat(perm(escape(villain)),herosJourney)
holdsat(live(herosJourney),herosJourney)
holdsat(role(villain,villain),evilEmpire)
holdsat(role(hero,hero),evilEmpire)
holdsat(role(empire,empire),evilEmpire)
holdsat(place(home,home),evilEmpire)
holdsat(place(evilLair,evilLair),evilEmpire)
holdsat(phase(evilEmpire,phaseA),evilEmpire)
holdsat(pow(intEvilEmpire(hero,empire)),evilEmpire)
holdsat(perm(null),evilEmpire)
holdsat(perm(intEvilEmpire(hero,empire)),evilEmpire)
holdsat(perm(capture(hero,empire)),evilEmpire)
holdsat(live(evilEmpire),evilEmpire)
occurred(null,herosJourney)
occurred(intEvilEmpire(hero,empire),evilEmpire)
occurred(chase(hero,empire),evilEmpire)

```

```

Time Step 3:

holdsat(role(villain,villain),herosJourney)
holdsat(role(hero,hero),herosJourney)
holdsat(role(empire,empire),herosJourney)
holdsat(place(home,home),herosJourney)
holdsat(place(evilLair,evilLair),herosJourney)
holdsat(phase(herosJourney,phaseB),herosJourney)
holdsat(pow(intHerosJourney(hero,villain,evilLair,home)),herosJourney)
holdsat(perm(null),herosJourney)
holdsat(perm(intHerosJourney(hero,villain,evilLair,home)),herosJourney)
holdsat(perm(go(hero,home)),herosJourney)
holdsat(live(herosJourney),herosJourney)
holdsat(role(villain,villain),evilEmpire)
holdsat(role(hero,hero),evilEmpire)
holdsat(role(empire,empire),evilEmpire)
holdsat(place(home,home),evilEmpire)
holdsat(place(evilLair,evilLair),evilEmpire)
holdsat(phase(evilEmpire,phaseA),evilEmpire)
holdsat(pow(intEvilEmpire(hero,empire)),evilEmpire)
holdsat(perm(null),evilEmpire)
holdsat(perm(intEvilEmpire(hero,empire)),evilEmpire)
holdsat(perm(capture(hero,empire)),evilEmpire)
holdsat(live(evilEmpire),evilEmpire)
occurred(kill(hero,villain),herosJourney)
occurred(intHerosJourney(hero,villain,evilLair,home),herosJourney)
occurred(null,evilEmpire)


Time Step 4:

holdsat(role(villain,villain),herosJourney)
holdsat(role(hero,hero),herosJourney)
holdsat(role(empire,empire),herosJourney)
holdsat(place(home,home),herosJourney)
holdsat(place(evilLair,evilLair),herosJourney)
holdsat(pow(intHerosJourney(hero,villain,evilLair,home)),herosJourney)
holdsat(perm(null),herosJourney)
holdsat(perm(intHerosJourney(hero,villain,evilLair,home)),herosJourney)
holdsat(live(herosJourney),herosJourney)
holdsat(role(villain,villain),evilEmpire)
holdsat(role(hero,hero),evilEmpire)
holdsat(role(empire,empire),evilEmpire)
holdsat(place(home,home),evilEmpire)
holdsat(place(evilLair,evilLair),evilEmpire)
holdsat(phase(evilEmpire,phaseA),evilEmpire)
holdsat(pow(intEvilEmpire(hero,empire)),evilEmpire)
holdsat(perm(null),evilEmpire)
holdsat(perm(intEvilEmpire(hero,empire)),evilEmpire)
holdsat(perm(capture(hero,empire)),evilEmpire)
holdsat(live(evilEmpire),evilEmpire)
occurred(intHerosJourney(hero,villain,evilLair,home),herosJourney)
occurred(go(hero,home),herosJourney)
occurred(null,evilEmpire)


Time Step 5:

holdsat(role(villain,villain),herosJourney)
holdsat(role(hero,hero),herosJourney)
holdsat(role(empire,empire),herosJourney)
holdsat(place(home,home),herosJourney)
holdsat(place(evilLair,evilLair),herosJourney)
holdsat(pow(intHerosJourney(hero,villain,evilLair,home)),herosJourney)
holdsat(perm(null),herosJourney)
holdsat(perm(intHerosJourney(hero,villain,evilLair,home)),herosJourney)
holdsat(live(herosJourney),herosJourney)
holdsat(role(villain,villain),evilEmpire)
holdsat(role(hero,hero),evilEmpire)
holdsat(role(empire,empire),evilEmpire)
holdsat(place(home,home),evilEmpire)
holdsat(place(evilLair,evilLair),evilEmpire)
holdsat(phase(evilEmpire,phaseB),evilEmpire)
holdsat(pow(intEvilEmpire(hero,empire)),evilEmpire)
holdsat(perm(null),evilEmpire)
holdsat(perm(intEvilEmpire(hero,empire)),evilEmpire)
holdsat(perm(escape(hero)),evilEmpire)
holdsat(live(evilEmpire),evilEmpire)
occurred(null,herosJourney)
occurred(intEvilEmpire(hero,empire),evilEmpire)
occurred(capture(hero,empire),evilEmpire)


Time Step 6:

holdsat(role(villain,villain),herosJourney)
holdsat(role(hero,hero),herosJourney)
holdsat(role(empire,empire),herosJourney)
holdsat(place(home,home),herosJourney)
holdsat(place(evilLair,evilLair),herosJourney)
holdsat(pow(intHerosJourney(hero,villain,evilLair,home)),herosJourney)
holdsat(perm(null),herosJourney)
holdsat(perm(intHerosJourney(hero,villain,evilLair,home)),herosJourney)
holdsat(live(herosJourney),herosJourney)
holdsat(role(villain,villain),evilEmpire)
holdsat(role(hero,hero),evilEmpire)
holdsat(role(empire,empire),evilEmpire)
holdsat(place(home,home),evilEmpire)
holdsat(place(evilLair,evilLair),evilEmpire)
holdsat(pow(intEvilEmpire(hero,empire)),evilEmpire)
holdsat(perm(null),evilEmpire)
```

① This linkage could be a lot better.

```
holdsat(perm(intEvilEmpire(hero,empire)),evilEmpire)
holdsat(live(evilEmpire),evilEmpire)
occurred(escape(hero),herosJourney)
occurred(viol(escape(hero)),herosJourney)
occurred(intEvilEmpire(hero,empire),evilEmpire)
occurred(escape(hero),evilEmpire)
```

or use alternative visualisation

Matt: Move to appendix?

These trace outputs do not assist the reader in visualising the possible sequences of events that they represent. For this reason, we suggest that the reader refer to the visualisations produced by the StoryBuilder tool in section 7.6.

quite right

## 6.8 Application to ~~Extending for~~ Legal Policies

Right now as I encounter this, it seems like the wrong place... let's see.

①

Though TropICAL is designed to describe non-linear stories in systems with intelligent agents such as interactive computer games, with minor changes it can also be adjusted to describe legal contracts. Where a story describes the constraints on character behaviour that any character can in theory break away from, a legal contract performs a similar role by describing the expected behaviour of its parties and the consequences for breaking those expectations.

In the case of contract law, though each individual contract between plaintiff and defendant would be different, common patterns exist between contracts. For example, contracts typically have a fixed duration and penalty for premature termination. There are usually clauses in a contract that describe the conditions for termination in the case of one of the parties performing certain unpermitted actions. These are what we refer to as a "policy" throughout this section: an abstract description of a commonly-occurring legal pattern that is analogous to a trope in a story.

In order to divide legal contracts into reusable components, the concept of tropes can easily be applied to capture fragments of the law. Some examples are:

- **The Warranty**: A *seller* sells an item to a *buyer*. If the item is defective, then the *buyer* has the right to return it within a certain period of time.
- **The Lease**: A *lessor* leases an item or property to a *lessee*. The lessee is obliged to pay rental fees on time, and to keep the item or property in good condition. The lessor is obliged to perform maintenance and necessary repairs on the item or property.
- **The Deposit**: A sum of money is given from one party to another with the understanding that it is to be returned upon expiration of a contract.

As with tropes that contain sub-tropes, policies can have sub-policies. For example, a "sales contract" policy could contain a "warranty" policy as a sub-policy. This can be thought of as adding a clause to a contract. Similarly, a "lease" policy could be a sub-policy of a "tenancy agreement" policy, which keeps the spirit of the lease policy but adds the requirement for the lessee to not make excessive noise, for example. These policies describe the actions that both parties are permitted and obliged to carry out, in a sequence of events.

As an example of describing a legal policy in the manner of a story trope, consider the subleasing of an object or property from a lessor to a lessee. In such an arrangement, the following dispute could occur:

- The *lessor* leases property to the *lessor*.
- The *lessee* then subleases the property to a *third party*.
- The *lessor* cancels the lease contract, stating that the subleasing is a violation of the contract.

These policies can be expressed in terms of social norms, and translated into TropICAL code:

Listing 6.37: Example policies in TropICAL

```
``Lease'' is a policy where:
    The Lessor is a role
    The Lessor is a role
    The Thing is an object
    The Lessor leases the Thing to the Lessee
    Then the ``Maintenance of Confidence'' policy applies

``Sublease'' is a policy where:
    The Lessor is a role
    The Third Party is a role
    The Thing is an object
    The Lessor may sublease the Thing to a Third Party

``Sublease Permission'' is a policy where:
    The Lessee is a role
    The Lessor is a role
    The Thing is an object
    The Lessee may ask permission to sublease the Thing
    If the Lessor gives permission to the Lessee:
      The ``Sublease'' policy applies

``Maintenance of Confidence'' is a policy where:
    The Lessee is a role
    The Lessor is a role
    The Due Date is an object
    The Lease is an object
    The Thing is an object
    The Lessee must pay the Lessor before the Due Date arrives
      Otherwise, the Lessor may cancel the Lease
    The Lessor must maintain the Thing
      Otherwise, the Lessee may cancel the Lease
```

The use of TropICAL to describe legal policies would be useful for lawyers that want to validate or generate arguments from any given set of policies. This would be valuable as a tool to aid lawyers in argument creation, for example. Suppose a defense lawyer is trying to argue

Ths reads much more as a use case than part of the description of TropICAL.

that her client is not guilty. The system could generate traces (sequences of events) of a given length, whose outcome does not result on the violation of any contract or law. The lawyer would then be able to choose from amongst a list of generated arguments to make their case.

Adapting this idea to our *sublease* example described above, we can generate sequences of events where a lessee has subleased a property, and find out whether or not their actions have been in violation of a policy. We achieve this through the specification of constraints. For example, a prosecution lawyer wishing to find all sequences of events where both sublease event and violation events have occurred would specify this ASP constraint.

```
violEvent :- occurred(viol(X), I, T).
:- not violEvent.
subleaseEvent :- occurred(sublease(X, Y, Z), I, T), holdsat(role(X, lessee),
    I, T).
:- not subleaseEvent.
```

This constraint specifies that we want to generate all possible sequences where the lessee has subleased something, and where a violation has occurred. Headless rules in ASP (where the leftmost part of the rule is simply ":-") state what we do *not* want in generated answer sets, so in this example both headless statements are double negatives.

The solver outputs several answer sets, containing event sequences (traces) of a specified length. We can then parse these answer sets into a human-readable "executive summary" that can be used by lawyers. An example of such a summary would be:

```
Possibility 0:

The following occurred:
  Alice Leases Bob House
Then:
  Bob Subleases Charlotte House
  VIOLATION: Bob Subleases Charlotte House
```

Each summary lists a number of "possibilities", with each possibility corresponding to an answer set produced by the solver, listing events and violations that occur in a trace. To find an argument, a lawyer can simply read through the generated possibilities to find one that suits her needs.

## 6.9 Summary

Returning to the requirements listed in section 6.2.2 (listed again here for the reader's convenience), we can now examine our language specification to see if the requirements are met:

R1. The software must be able to integrate into a multi-agent framework such as JASON

R2. The system must direct the behaviour of agents to fit a story description

R3. The user should be able to write and re-use existing story components (from a library)

R4. The components must be able to express sequences of events

R5. The components must be able to express branching (diverging) events

integrate in addition.

R6. The user should have the ability to nest existing components inside new components

R7. The user should be able to visualise the branches of the story that result in the addition or modification of components to the story.

R8. The story must tell the character agents what to do, but they should be free to break away from it in extreme circumstances, to add a degree of unpredictability

Chapter 5 describes how our system integrates with the JASON multi-agent framework. The output of an answer set solver such as Clingo can be used to add beliefs to an agent's mental model to let it know what actions it is permitted and obliged to carry out as part of a story. This fulfills requirement 1.

The use of permissions and obligations to describe story actions to agents in either a weakly (permissions) or strongly (obligations) enforced way gives the author two ways to direct agent behaviour. At times when the author wants agents to conform tightly with a trope description, they can give the agent an obligation with a strict consequence for non-compliance. This matches the needs of requirement 2. Requirement 8 states that authors need a way to add some flexibility into a story, so that character agents are given some freedom to break away from the story if needed. This is achieved through the use of permissions to regulate agent actions.

Section 6.3.2 shows how sequences of events can be described in TropICAL, fulfilling requirement 4. Section 6.3.4 does the same for branching story structures, meeting requirement 5. A means of abstraction is an important feature of our language, as stated in requirement 6. Section 6.3.5 describes the use of subtropes to meet the need for users to create their own abstractions.

As a text-based programming language, TropICAL addresses the requirements mentioned above (numbers 1, 2, 4, 5, 6 and 8). However, there are two requirements that cannot be fulfilled through a text-based programming paradigm alone: selecting pre-existing story components from a library (requirement 3), and visualising the branches of the story as it is modified (requirement 7). These requirements are best addressed through an Interactive Development Environment (IDE) with which we can add graphical features for the browsing, selection and visualisation of tropes. The next chapter (Chapter 7) introduces *StoryBuilder*, a browser-based tool that adds these features to our TropICAL language.