

Chapter 5

Case Studies

We now give several case studies of the modeling and programming methodologies described so far. This chapter presents the results of the research; we aim to illustrate the concrete payoff that linear logic programming can have across a breadth of ludonarrative domains. We argue that the specifications presented here are much more concise and readable than they would be in a general-purpose language like JavaScript or C++, while allowing for a great deal more flexibility in expression than domain-specific languages for any of the game genres represented here. Additionally, these specifications allow for augmentation by player strategies for fuzzing, analysis in the form of visualization for causal dependencies in play traces, and the potential for deeper analysis in terms of high-level design concepts such as game balance.

In an effort to demonstrate breadth, we give the code for five complete case studies of various worlds:

- Refining an Interactive Social Story World, or *BuffyWorld*: building on the narrative modeling ideas presented in Chapters 2 and 3, we model an interactive social story world based on the television show *Buffy the Vampire Slayer*.
- Dungeon Crawler, or *DungeonWorld*: we model a “hack-and-slash” roleplaying combat mechanic with resource-management feedback loops.
- Garden Simulator, or *GardenWorld*: we model a similar mechanic to *DungeonWorld* in terms of resource management, but using a nurturing rather than violent context to inspire player affordances and autonomous world evolution.
- Settlers of Catan, or *SiedlerWorld*: we model a minimal version of the board game Settlers of Catan, in which players compete for territory and strategize about resource exchange feedback loops.
- Tamara: finally, we show an experiment that follows on from the narrative generation work in Chapter 3: we use Ceptre to model a work of participatory theater called *Tamara* [Kri89] where scenes take place in multiple rooms simultaneously. In this example, we use the result of program execution, i.e. the proof term, as the interactive artifact, rendered as a Twine text adventure that replicates the participation mechanic in the original play (that is, following characters when they exit).

5.1 Refining an Interactive Social Story World

This case study uses a similar story world to the narrative generation example from Chapter 3: characters and locations are the basic types; predicates include character locations and relationships between one another; story actions change and depend on the states described by those predicates.

The main new idea in this case study is an *action/reaction* dichotomy: in an attempt to chain together less random sequences of story, characters may perform up to one action per turn of simulation, but they also must *react* to any action performed toward them by another character.

Additionally, we let the player pick one of the characters to control for the duration of the interaction rather than letting them choose arbitrary character actions, hopefully lending a stronger sense of continuity to the experience.

The combination of these two axes, act/react and player/non-player character, make up the four stages of the program.

5.1.1 Story World Overview

Buffy the Vampire Slayer is a television franchise created by Joss Whedon, airing 1997-2003. The basic premise is that Buffy, a teenage girl who just wants an ordinary life, is the “chosen one,” or Slayer, obliged to protect the world from supernatural evils lurking in the California town of Sunnydale. Throughout the show’s seven seasons, the cast of characters and specific threats of evil vary substantially, but a few things remain constant rules for the Buffy universe. Here are a few that we use to shape our story world:

- Some characters are *evil*, e.g. by virtue of being vampires or demons. The slayer is obliged to destroy evil creatures.
- Evil people feel no remorse about killing, but will generally not do so at random; only if a particular goal (like destroying the slayer) is achieved. Vampires and demons are usually evil.
- Xander and Willow are Buffy’s best friends; Giles is the slayer’s appointed *watcher* (similar to a mentor).
- Dawn is Buffy’s younger sister, who frequently drives the plot forward by getting kidnapped or accidentally casting spells that Buffy and her friends have to thwart.
- *Witches* are humans who have mastered some control over the supernatural powers that be, and can use them to various good or ill effects.

In this encoding, we include a few other members of the Buffy universe: Anya, a no-longer-evil demon (dating Xander); Tara, another witch (dating Willow); Spike, a vampire who develops a romantic obsession with Buffy; and an abstract, unspecified villain who hunts Dawn.

5.1.2 Code

Code for this section is available at <https://github.com/chrisamaphone/interactive-lp/blob/master/examples/buffy.cep>.

Below we give the characters and predicates relating characters to each other. The predicate `npc C` marks `C` as a “non-player character.”

```
character : type.
buffy : character.
willow : character.
xander : character.
anya : character.
tara : character.
giles : character.
dawn : character.
spike : character.
villain : character.

npc character : pred.

evil character : pred.
witch character : pred.
slayer character : pred.
watcher character : pred.

weak character : pred.
strong character : pred.

friends character character : pred.
crush character character : pred.
dating character character : pred.
dislike character character : pred.
hunting character character : pred.
hurt_by character character : pred.
```

There are six central locations we will refer to: the crypt in the graveyard (Spike’s home), the graveyard itself, Buffy’s house, the Magic Box (a home for Giles’ personal library and occult collection, disguised as a novelty shop), Dawn’s high school, and the Bronze, a local hangout and bar. The world map is specified below.

```
place : type.
crypt : place.
graveyard : place.
buffy_house : place.
magic_box : place.
villain_lair : place.
high_school : place.

adjacent place place : bwd.
```

adjacent crypt graveyard.
adjacent graveyard crypt.
adjacent crypt magic_box.
adjacent magic_box crypt.
adjacent graveyard buffy_house.
adjacent buffy_house graveyard.
adjacent buffy_house magic_box.
adjacent magic_box buffy_house.
adjacent magic_box graveyard.
adjacent graveyard magic_box.
adjacent magic_box villain_lair.
adjacent villain_lair magic_box.
adjacent buffy_house villain_lair.
adjacent villain_lair buffy_house.
adjacent buffy_house high_school.
adjacent high_school magic_box.

at character place : pred.

The following intermediate states result from actions taken by characters; they represent an ongoing action sequence that needs to be resolved by some reaction.

attacking character character : pred.
hitting_on character character : pred.
spellcast character character : pred.
joking_with character character : pred.
accusing character character : pred.
pushing_away character character : pred.
acting_affectionate character character : pred.
bickering character character : pred.

The initial state specifies initial relationships between characters, as well as individual characteristics that may determine the range of actions available, for the particular season of the show and (perhaps) episode. (This one is loosely based off of Season 6.)

```
context init_season =  
{ friends buffy willow, friends willow buffy,  
  friends buffy xander, friends xander buffy,  
  friends willow xander, friends xander willow,  
  friends dawn buffy, friends buffy dawn,  
  friends dawn xander, friends xander dawn,  
  friends dawn willow, friends willow dawn,  
  friends dawn tara, friends tara dawn,  
  dating willow tara, dating tara willow,  
  dating xander anya, dating anya xander,  
  friends giles buffy, friends buffy giles,  
  friends willow giles, friends giles willow,  
  dislike giles spike, dislike spike giles,  
  dislike xander spike, dislike spike xander,
```

```

evil spike, crush spike buffy, dislike buffy spike,
dislike villain buffy, evil villain,
witch willow, slayer buffy, watcher giles, witch tara,
strong buffy, strong willow, strong villain,
weak xander, weak giles, weak anya, weak tara, weak dawn, weak spike}.

context init_episode =
{ hunting villain dawn, crush buffy spike,
  at dawn high_school,
  at buffy graveyard, at willow buffy_house, at tara buffy_house,
  at xander magic_box, at anya magic_box, at giles magic_box,
  at spike crypt, at villain villain_lair
}

```

The following machinery allows the player to select a character to be the “PC” (player character) from among the pool of NPCs (non-player characters).

```

no_pc : pred.
pc character : pred.
turn : pred.
end : pred. % end of story.
go : pred. % not end of story.

context init_game =
{npc buffy, npc willow, npc xander, npc anya, npc tara,
 npc giles, npc spike, npc villain, npc dawn,
no_pc}.

stage select_character = {
  pick : npc C * no_pc -o pc C.
}
#interactive select_character.
select_character_to_pc_choices
: qui * stage select_character -o stage pc_choices * turn.

```

Next, we describe the choices available to the player character as actions. These include moving from place to place, and acting on other characters, specifically by attacking, making romantic advances, showing affection, joking with them, or accusing them of causing hurt. Witches can also cast spells that affect whether a character is *strong* or *weak* (which in turn affects their outcomes in combat).

The player character may also choose to end the story.

```

stage pc_choices = {
  act/move
    : turn * $pc C
    * at C L * adjacent L L'
    -o at C L' * go.

  act/kidnap/evil

```

```

: turn * $pc C
* $evil C
* at C L * at C' L * weak C'
* adjacent L L'
  -o at C L' * at C' L' * go.

act/attack/evil
: turn * $pc C
* $evil C
* $at C L * $at C' L
  -o attacking C C' * go.

act/attack/slayer
: turn * $pc C
* $slayer C * $at C L * $at C' L * $evil C'
  -o attacking C C' * go.

act/attack/revenge/friend
: turn * $pc C
* $at C L * $at C' L
* $friends C Victim * hurt_by Victim C'
* $evil C'
  -o attacking C C' * go.

act/attack/revenge/lover
: turn * $pc C
* $at C L * $at C' L
* $dating C Victim * hurt_by Victim C'
* $evil C'
  -o attacking C C' * go.

act/hit_on
: turn * $pc C
* $at C L * $at C' L * $crush C C'
  -o hitting_on C C' * go.

act/affection
: turn * $pc C
* $at C L * $at C' L * $dating C C'
  -o acting_affectionate C C' * go.

act/bicker
: turn * $pc C
* $at C L * $at C' L * $dating C C'
  -o bickering C C' * go.

```

```

act/spellcast/weak_to_strong
: turn * $pc C * $switch C
* $at C L * $at C' L
* weak C'
-o strong C' * spellcast C C' * go.

act/spellcast/strong_to_weak
: turn * $pc C * $switch C
* $at C L * $at C' L
* strong C'
-o weak C' * spellcast C C' * go.

act/watcher/push_slayer_away
: turn * $pc C * $watcher C
* $at C L * $at C' L * $slayer C'
* $strong C'
-o pushing_away C C' * go.

act/accuse
: turn * $pc C
* $at C L * $at C' L
* $hurt_by C C'
-o accusing C C' * go.

act/joke_with
: turn * $pc C
* $at C L * $at C' L
-o joking_with C C' * go.

act/none : turn -o go.

end_story : turn -o end.
}
#interactive pc_choices.

pc_to_npc :
qui * stage pc_choices * go
-o stage npc_reactions.

pc_to_done :
qui * stage pc_choices * end
-o stage done.

stage done = {}

```

Next, we describe how NPCs may *react* to character actions. Extending the analogy of “social physics,” this technique metaphorically reflects Newton’s Third Law—every

intermediate state introduced by (appearing on the right-hand-side of) an action must be processed by (appear on the left-hand-side of) a reaction. Reactions to a given action may be nondeterministic and may depend on other parts of social state.

```

stage npc_reactions = {

  react/attacking/flee
    : $npc C
    * attacking C' C * $weak C * at C L * adjacent L L' -o
      at C L' * hunting C' C.
  react/attacking/hurt
    : $npc C
    * attacking C' C * $weak C
      -o hurt_by C C'.
  react/attacking/fight/lose
    : $npc C
    * attacking C' C * $strong C -o hurt_by C C'.
  react/attacking/fight/win
    : $npc C
    * attacking C' C * $strong C -o hurt_by C' C.

  react/hit_on/date
    : $npc C
    * hitting_on C' C * $crush C C'
      -o dating C C' * dating C' C.
  react/hit_on/dislike
    : $npc C
    * hitting_on C' C * $dislike C C'
      -o hurt_by C' C.
  react/hit_on/reciprocate_crush
    : $npc C
    * hitting_on C' C
      -o crush C C'.

  react/spellcast/thanks
    : $npc C * spellcast C' C * $strong C
      -o friends C C' * friends C' C.
  react/spellcast/wtf
    : $npc C * spellcast C' C
      -o accusing C C'.
  react/spellcast/bad
    : $npc C * spellcast C' C * $weak C
      -o dislike C C' * accusing C C'.

  react/accusing/counter
    : $npc C * accusing C' C
      -o accusing C C'.

```

```

react/accusing/apologize
    : $npc C * accusing C' C
      -o ().
react/accusing/breakup
    : $npc C * accusing C' C * dating C C' * dating C' C
      -o hurt_by C C' * hurt_by C' C.

react/joke_with/shoot_down
    : $npc C * joking_with C' C * $dislike C C'
      -o hurt_by C' C.
react/joke_with/laugh
    : $npc C * joking_with C' C -o ().

react/affection/good
    : $npc C * acting_affectionate C' C -o ().
react/affection/bad
    : $npc C * acting_affectionate C' C * $hurt_by C C'
      -o hurt_by C' C.

react/hurt_accuse
    : $npc C * hurt_by C C' * hurt_by C C'
      -o accusing C C'.
}

```

Next we give the NPC action stage, with a little bit of setup so that each character only takes one turn:

```

qui * stage npc_reactions
    -o stage gen_npc_turns.

stage gen_npc_turns = {
    npc C -o npc_turn C.
}
qui * stage gen_npc_turns -o stage npc_choices.

```

NPC actions mirror PC actions except that, not being guided by player choice, they sometimes require additional premises to drive their behavior in a coherent way.

```

stage npc_choices = {
    act/move_toward_friend
        : npc_turn C
          * at C L * $at C' L' * $friends C C' * adjacent L L'
          -o at C L' * npc C.

    act/move_toward_enemy
        : npc_turn C
          * at C L * $at C' L' * $hunting C C' * adjacent L L'
          -o at C L' * npc C.
}

```

```

act/attack_hunting
: npc_turn C
* $at C L * $at C' L * hunting C C'
-o attacking C C' * npc C.

act/evil_attack_dislike
: npc_turn C
* $evil C * $at C L * $at C' L * $dislike C C'
-o attacking C C' * npc C.

act/evil_attack_slayer
: npc_turn C
* $evil C * $at C L * $at B L * $slayer B
-o attacking C B * npc C .

act/slayer_attack_evil
: npc_turn C
* $slayer C * $at C L * $at V L * $evil V
-o attacking C V * npc C.

act/attack/vengeance/friend
: npc_turn C
* $at C L * $at C' L
* $friends C Victim * hurt_by Victim C'
-o attacking C C' * npc C.

act/hit_on_crush
: npc_turn C
* $at C L * $at C' L * $crush C C'
-o hitting_on C C' * npc C.

act/affection_date
: npc_turn C
* $at C L * $at C' L * $dating C C'
-o acting_affectionate C C' * npc C.

act/spellcast/weak_to_strong
: npc_turn C
* $switch C
* $at C L * $at C' L
* weak C'
-o strong C' * spellcast C C' * npc C.

act/spellcast/strong_to_weak
: npc_turn C
* $switch C

```

```

    * $at C L * $at C' L
    * strong C'
      -o weak C' * spellcast C C' * npc C.

act/accuse
: npc_turn C
  * $at C L * $at C' L
  * $hurt_by C C'
  -o accusing C C' * npc C.

act/joke_with
: npc_turn C
  * $at C L * $at C' L
  * $friends C C'
  -o joking_with C C' * npc C.

act/none : npc_turn C -o npc C.

}

Finally, we give the stage for player character reactions. Again, these basically mirror the NPC reactions, except that they sometimes have fewer premises (allowing the player to act out of their own volition and playful goals). Reactions must be selected until no more actions on the player are left to be reacted to. After the player has chosen all reactions, we go back to the pc_choices stage for the player to make new actions.

qui * stage npc_choices -o stage pc_reactions.

stage pc_reactions = {
  %%% Player Reactions %%%

  react/attacking/fight
    : $pc C * attacking C' C
      -o attacking C C'.
  react/attacking/flee
    : $pc C * attacking C' C * at C L * adjacent L L'
      -o at C L' * hunting C' C.

  react/hitting_on/reciprocate
    : $pc C * hitting_on C' C
      -o dating C C' * dating C' C.
  react/hitting_on/shoot_down
    : $pc C * hitting_on C' C
      -o hurt_by C' C.

  react/joke_with/shoot_down
    : $pc C * joking_with C' C
      -o dislike C C'.

```

```

react/joke_with/laugh
  : $pc C * joking_with C' C
    -o friends C' C * friends C C'.

react/spellcast/thanks
  : $pc C * spellcast C' C * $strong C
    -o friends C C' * friends C' C.
react/spellcast/wtf
  : $pc C * spellcast C' C
    -o accusing C C'.
react/spellcast/bad
  : $pc C * spellcast C' C * $weak C
    -o dislike C C' * accusing C C'.

react/accusing/counter
  : $pc C * accusing C' C
    -o accusing C C'.
react/accusing/apologize
  : $pc C * accusing C' C
    -o ().
react/accusing/breakup
  : $pc C * accusing C' C * dating C C' * dating C' C
    -o hurt_by C C' * hurt_by C' C.

react/acting_affectionate/happy
  : $pc C * acting_affectionate C' C
    -o ().
react/acting_affectionate/shoot_down
  : $pc C * acting_affectionate C' C
    -o hurt_by C' C.
}
#interactive pc_reactions.
qui * stage pc_reactions -o stage pc_choices * turn.

#trace _ select_character {init_story, init_locations, init_game}.

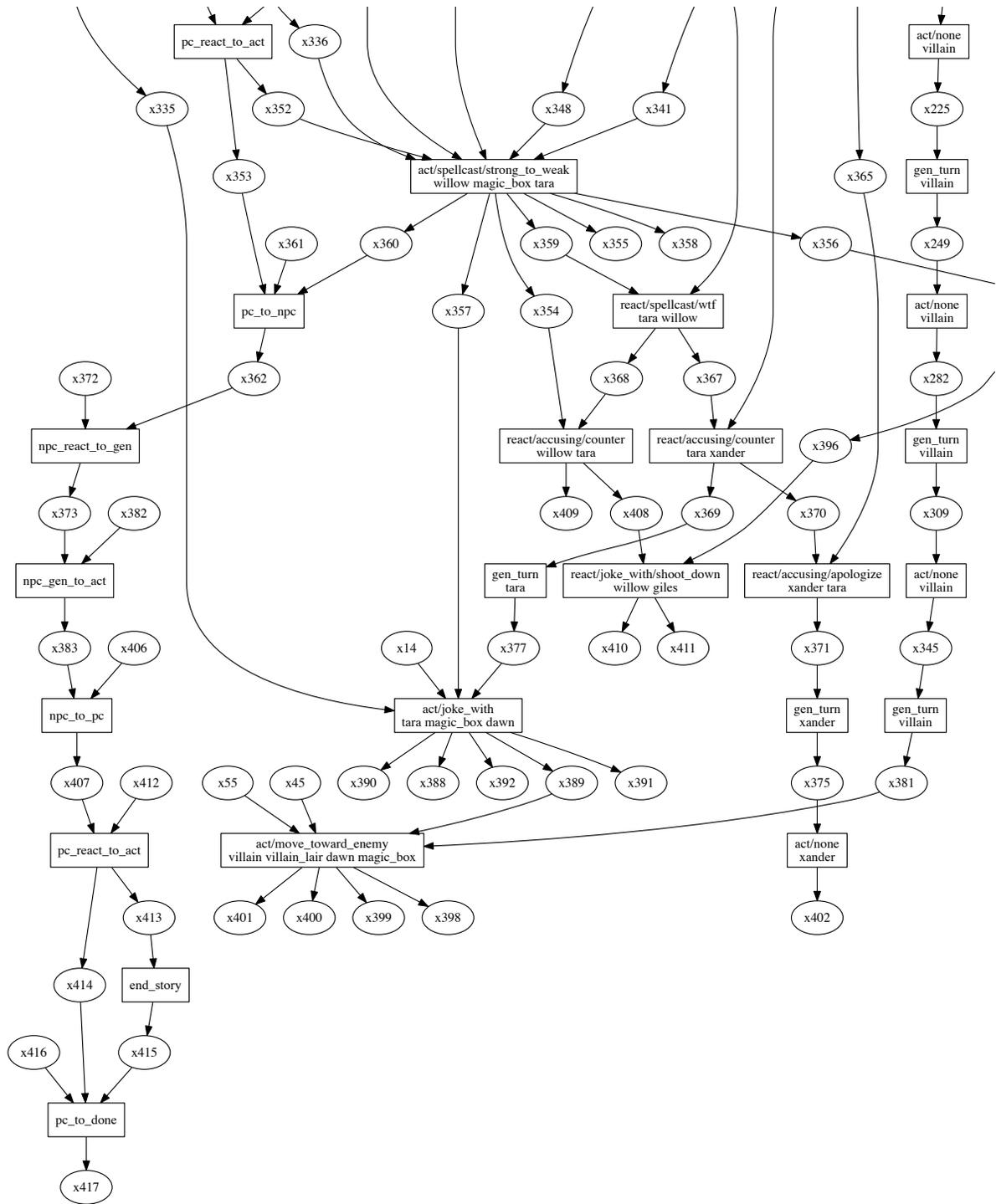
```

5.1.3 Sample Interaction

To illustrate interaction with the program, rather than giving a transcript, we show graphical output of a few interesting play traces (automatically generated with GraphViz¹).

In the first interaction, we select the character Willow. In the interactive stages for determining her actions, we choose transitions that correspond to her casting spells, arguing with Tara, and joking with her friends. A portion of the resulting trace follows:

¹<http://www.graphviz.org/>



This graph depicts the action of Willow casting a spell on Tara, Tara reacting negatively, and other actions by the non-player characters. One of the final scenes (transitions whose output resources have no out-edges) depicts the villain pursuing Dawn. A side argument between Xander arises, due to Tara having hurt Willow (Xander’s friend) in a prior argument.

Re-enacting an Episode

We created an augmented version of the story world where every character may be controlled by interaction for the sake of re-enacting a specific *Buffy* episode: the critically acclaimed “musical” episode, featuring a demon who causes the town to express their inner thoughts through song and dance.

In the episode, Dawn is kidnapped by the dance demon, Anya and Xander’s relationship falters, Tara discovers Willow used a spell to erase her memory of them arguing, and Giles pushes Buffy away from his mentorship. The episode culminates on Buffy’s charge to rescue Dawn, where she is eventually followed by all of her friends for a climactic final scene.

The character interactions involved in this climactic build-up can be seen in Figure 5.1.

5.1.4 Discussion

This case study has presented Ceptre code for a collaborative storytelling system between a human player and several NPCs, who all may act and react according to similar logic. It does not contain any scripted scenes or endings and is in that sense a “sandbox” or “open world” narrative system, but it is also somewhat asymmetric in that characters are assigned different traits and abilities. We could imagine easily extending this example to model supernatural powers such as vampiric siring, summoning demons, and casting more intricate spells.

The development of this example also serves to suggest the limitations of linear logic programming for narrative expression. This action/reaction model serves as substitute for characters having their own goals pertaining to the narrative and solve for those goals to create plans using the same inference mechanisms available to the logic itself. The use of planning for interactive storytelling includes that kind of character-specific inference [CCM02, RY10], though a logical formalism for both projective inference of this kind *and* state change would need at least the fully-reflexive (dependently-typed) capabilities of CLF in which predicates may refer to proofs as terms. We leave further investigations of this potential logical correspondence to future work.

Another limitation of the exemplified technique, though not necessarily of the programming language, is that each action or reaction must specify the complete set of characters it involves; any social interactions involving three or more characters would require additional rules. We believe that the “broadcast” pattern shown in Section 4.3.3 could be a basis for an interaction models wherein all characters are given the opportunity to react to any action taking place, say, in the same room as them, but it could substantially complicate the program.

As future work, we would also like to investigate the potential to use “social practices” a la Versu [ES] to constrain social actions. The basic idea is that at any given time, a finite number of social practices, or sets of roles and rules for those roles, apply to the world state, and characters select their actions on the basis of their roles within those practices. As an example, Buffy might only attack someone during a *combat* social

practice, in which she inhabits her role as the slayer, but she might navigate that role separately from (or simultaneously with) navigating her role as a supportive friend to Willow in the social practice of friends confiding in one another.

In summary, we have presented an investigation into the use of Ceptre to specify an interactive social story world based on *Buffy the Vampire Slayer*, using an action/reaction model and allowing the interactor to select a character to play throughout the interaction. This concludes the case study.

5.2 Dungeon Crawler

Next, we examine the task of specifying a “dungeon crawler”-inspired game design. This game might have state such as player character skill level, equipment such as weapons and armor, treasure that can be spent, player character health, and NPC health. It might have actions like adventuring or combat (in which randomness as well as player-determined properties can influence the outcome), resting to recover from damage, shopping to spend money on weapons or armor, and training to increase skill level.

We choose this domain as an exemplar of a common and popular operational logic: for example, the basic idea can be found in analog games like *Dungeons and Dragons* [GAH74] and digital games ranging in time from *Rogue* and *Hack* to the *Legend of Zelda* series to contemporary games made by large studios, like Blizzard’s *Hearthstone*. Representing this operational logic, at the core of so many existing games, should be a good benchmark for the expressiveness of Ceptre.

5.2.1 Game Design Goals

We speculate that the reason this operational logic (and common mechanics created with it) has enjoyed so much success has to do with the *dynamics* afforded by feedback loops in the mechanics: easy, small tasks (training) lead to medium-sized successes (defeating an enemy), which has payoff (treasure) that accumulates until larger successes (purchasing a shiny new weapon) may be enjoyed—and those successes feed back into the ease of the smaller and medium-sized tasks until a satisfying feeling of *momentum* is achieved. Typically, as rewards scale, they also enable more *strategic agency*: they present choices (e.g. between a large number of upgrade items for purchase) that give the player not just a sense of difficulty in selecting the “right” one, but also a sense of personalization in choosing between several equally-good alternatives (particularly when those choices are reflected in the form of an avatar).

In our experiment, we do not aspire to creating the same kind of aesthetic enjoyment made possible by these mechanics, but we do hope to create at least a similar-feeling *dynamic* of play: the sense that *increase in momentum* occurs when the player finds a working strategy. That goal will inform the choices we make.

5.2.2 Iteration 1

Code for this section is available at <https://github.com/chrisamaphone/interactive-lp/blob/master/examples/rpg-tiny.cep>.

We start with a minimal iteration of the game wherein resources—treasure, stamina, damage, and skill—are represented as atomic predicates.

```
treasure : pred.
stamina  : pred.
damage   : pred.
skill    : pred.
```

A few intermediate predicates are used as well. (We use these by invariant as Boolean values—at every program step, each of these is either in the context once or zero times.)

```
alive    : pred.
fighting : pred.
turn     : pred.
```

The do stage is controlled by the player, and contains actions for training, fighting, healing, resting, and upgrading a weapon (which is in effect simply training without consuming stamina).

```
stage do = {
  train : turn * stamina * $alive -o skill.
  fight : turn * stamina * $alive -o fighting.
  heal  : turn * $alive * damage -o stamina.
  rest  : turn * $alive -o stamina.
  buy_weapon : turn * treasure * treasure * treasure * $alive -o skill.
}
#interactive do.
qui * stage do -o stage happen.
```

The happen stage processes the player’s selected action, nondeterministically resolving fights (though they are more likely to result in a hit the more skill a player has) and determining when the player gets tired or dies.

```
stage happen = {
  turn -o ().
  fight/hit : fighting * $skill -o treasure.
  fight/miss : fighting -o damage.
  get_tired : $damage * stamina * $alive -o ().
  die : damage * damage * damage * alive -o ().
}
qui * stage happen -o stage do * turn.
```

We can run the simulation with a starting configuration like the one below, and fiddle with the amount of starting stamina if needed:

```
context init = { stamina, stamina, stamina, alive, turn}.
#trace _ do init.
```

5.2.3 Iteration 2

Code for this section is available at <https://github.com/chrisamaphone/interactive-lp/blob/master/examples/rpg.cep>.

In our second iteration, we will use numerically-indexed predicates rather than multiple copies of an atomic predicate, so that we may do simple arithmetic on their values.

Ceptre allows the definition of permanent facts via backward-chaining predicates, so for instance, we can define the arithmetic operation of addition capped at a certain maximum value as a predicate `cplus A B Cap C` which can be read as *A plus B capped at Cap is C*. We omit the definition of the predicate for brevity, but make use of it in later rules. We also use backward-chaining predicates to define a few constants, such as the player's maximum health and the damage and cost of various weapons:

```
max_hp 10.  damage sword 4.  cost sword 10.
```

We then define a initial context and an initial stage that sets up the game's starting state:

```
context init_ctx = {init_tok}.
stage init = {
  i : init_tok * max_hp N
    -o health N * treasure 0 * ndays 0
    * weapon_damage 4.
}
```

We define the rest of the game using a "screen" idiom, with predicates representing the main, rest, adventure, and shop screens. (Some type header information is omitted.)

```
qui * stage init -o stage main * main_screen.

stage main = {
  do/rest : main_screen -o rest_screen.
  do/adventure : main_screen -o adventure_screen.
  do/shop : main_screen -o shop_screen.

  do/quit : main_screen -o quit.
}
#interactive main.

qui * stage main * $rest_screen -o stage rest.
qui * stage main * $shop_screen -o stage shop.
qui * stage main * $adventure_screen -o stage adventure.
qui * stage main * quit -o ().
```

The rest and shop stages allow recharging health (at the cost of an increment to the number of days) and upgrading one's weapon damage in exchange for treasure, respectively:

```
stage rest = {
  recharge : rest_screen
    * health HP * max_hp Max * recharge_hp Recharge
```

```

        * cplus HP Recharge Max N
        * ndays NDAYS
        -o health N * ndays (NDAYS + 1).
    }
    qui * stage rest -o stage main * main_screen.

    stage shop = {
        leave : shop_screen -o main_screen.
        buy : treasure T * cost W C * damage_of W D * weapon_damage _
            * subtract T C (some T')
            -o treasure T' * weapon_damage D.
    }
    #interactive shop.
    qui * stage shop * $main_screen -o stage main.

```

The adventure stages are the most complex part of the code, involving the random generation of a monster and random spoils are collected upon player victory. Spoils are only added to the treasure bank if the player does not flee from a fight in progress. To do all of this, we need an adventure initialization stage (`init`), a monster generating stage (`fight_init`), a way of responding to player actions in context (`fight_auto`), and a choice for the player between fighting and fleeing (`fight`):

```

    stage adventure = {
        init : adventure_screen -o spoils 0.
    }
    qui * stage adventure -o stage fight_init * fight_screen.

    % drop_amount M N means a monster of size M can drop N coins
    drop_amount nat nat : bwd.
    drop_amount X X. % for now

    stage fight_init = {
        init : fight_screen -o gen_monster * fight_in_progress.
        gen_a_monster : gen_monster * monster_size Size
                    -o monster Size * monster_hp Size.
    }
    qui * stage fight_init -o stage fight * choice.

    try_fight : pred.
    fight_in_progress : pred.
    stage fight_auto = {
        fight/hit
        : try_fight * $fight_in_progress * monster_hp MHP * $weapon_damage D
        * subtract MHP D (some MHP')
        -o monster_hp MHP'.
    win
        : fight_in_progress * monster_hp MHP * $weapon_damage D
        * subtract MHP D none
    }

```

```

        -o win_screen.
fight/miss
    : try_fight * $fight_in_progress * $monster Size * health HP
      * subtract HP Size (some HP')
      -o health HP'.
die_from_damages
    : health 0 * fight_in_progress -o die_screen.
fight/die
    : try_fight * fight_in_progress * monster Size * health HP
      * subtract HP Size none
      -o die_screen.
}
choice : pred.
qui * stage fight_auto * $fight_in_progress -o stage fight * choice.
qui * stage fight_auto * $win_screen -o stage win.
qui * stage fight_auto * $die_screen -o stage die.

```

```

stage fight = {
    do_fight : choice * $fight_in_progress -o try_fight.
    do_flee  : choice * fight_in_progress -o flee_screen.
}
#interactive fight.
qui * stage fight * $fight_in_progress -o stage fight_auto.
qui * stage fight * $flee_screen -o stage flee.

```

Choosing flee takes you back to the main screen without any spoils:

```

stage flee = {
    % lose spoils
    do/flee : flee_screen * spoils X * monster _ * monster_hp _
            -o ().
}
qui * stage flee -o stage main * main_screen.

```

Finally, we need stages for winning and dying in combat:

```

go_home_or_continue : pred.
stage win = {
    win
    : win_screen * monster Size * drop_amount Size Drop
      -o drop Drop.
    collect_spoils
    : drop X * spoils Y * plus X Y Z
      -o spoils Z * go_home_or_continue.
    go_home
    : go_home_or_continue
      * spoils X * treasure Y * plus X Y Z
      -o treasure Z * main_screen.
}

```

```

    continue
    : go_home_or_continue -o fight_screen.
}
#interactive win.
qui * stage win * $main_screen -o stage main.
qui * stage win * $fight_screen -o stage fight_init.

end : pred.
stage die = {
    quit : die_screen -o end.
    restart : die_screen * monster_hp _
              * spoils _ * ndays _ * treasure _
              * weapon_damage _ -o init_tok.
}
#interactive die.

qui * stage die * end -o ().
qui * stage die * $init_tok -o stage init.

```

The program can then be run with the directive `#trace _ init init_ctx.`

The development of this example benefited extensively from the ability to specify interactivity modularly, occasionally making non-player-directed stages (like `fight_auto`) interactive to debug them. Additionally, we can test the design by “scripting” certain player strategies. For instance, we could augment the two rules in the `fight` stage to be deterministic, fighting when the monster can’t kill us in one turn and fleeing otherwise:

```

stage fight = {
    do_fight :
        choice * $fight_in_progress
        * $monster Size * $health HP * Size < HP
          -o try_fight.
    do_flee :
        choice * fight_in_progress
        * $monster Size * $health HP * Size >= HP
          -o flee_screen.
}

```

If we remove interactivity from this stage, then we get automated combat sequences that should never result in the player’s death.

5.2.4 Discussion

In summary, this case study illustrates the use of Ceptre to program simple combat strategy mechanics found in several dungeon-crawler, RPG, and roguelike game genres. We use stages extensively to coordinate not just interaction between the game and player, but also different autonomous parts of the game.

This case study suggests potential analyses that could be carried out on gameplay traces. The ability to script player strategies means that we could draft a few and

compare them, perhaps by charting the various numeric quantities over time (program steps). Such analyses are suggested by Dormans [Dor11] in the development of strategy game mechanics to study things like feedback loops.

5.3 Settlers of Catan

Our next case study is an encoding of a *non-digital* game, *The Settlers of Catan*. Settlers of Catan is a strategic board game designed by Klaus Teuber and published in 1995 [SCS10], a notable, award-winning exemplar of a trend of similarly-styled board games achieving popularity in the 1990s.

At a high level, Settlers is a game in which players take the role of colonizers of an island, Catan, and position their settlements in such a way as to maximize their benefit from the (somewhat randomized) production of resources. Resources can then be used to build more settlements, and ultimately to achieve victory (through a points system).

We choose Settlers as an object of study due to its popular success as well as its amenability to formalization in Ceptre due to its use of resource exchange as a central mechanic. It gives us an opportunity to illustrate how we can codify non-digital processes in a computational way such that one could imagine prototyping such rules in Ceptre before physical implementation. It allows us to illustrate operational logics like rolling dice, placing pieces, and drawing cards from a shuffled, but pre-determined deck. This will also be our first formalization of a multiplayer game, drawing attention to turn-taking in a new light and raising some interesting research questions about peer-to-peer protocols (as in trading) and information hiding (as in secret hands of cards).

Below we summarize the game rules described by Teuber [Teu07].

5.3.1 Game Rules

The Settlers of Catan is a 2-4-player board game played on a grid of hexagonal tiles (hexes), where each hex represents some terrain which may produce *resources*—wool, lumber, grain, brick, or ore—which players need to collect in order to build roads, settlements, and cities on the grid. Roads are placed along edges of the hexes, settlements are placed at intersections of hexes, and cities are *upgrades* of settlements.

Every turn, the player whose turn it is rolls a pair of six-sided dice to determine resource production. Every hex is marked with a number corresponding to a possible roll of the dice, and when that number comes up, those hexes produce whatever resources correspond to their terrain. Every player who has a settlement next to one of those hexes receives a copy of the resource, and every player who has a city next to one of them receives *two* copies.

After resource production, the player whose turn it is may *trade* resources with other players. They do so by making an offer, such as “I will trade one *wool* for either *lumber* or *grain*,” which may or may not be accepted by another player. Other players can make their own counter-offers as well. Trade takes place when (and only when) another player accepts the offer.

Finally, after trading, the player whose turn it is may *build* within the limits of their resources. They may build a road in any empty edge that connects to one of their pre-existing roads, settlements, or cities; they may build a settlement in any empty intersection that connects to one of their roads; and they may build a city as an upgrade to a settlement, replacing it on the game board. Building costs resources, summarized in the table below:

Building	Ore	Grain	Lumber	Wool	Brick
Road	-	-	1	-	1
Settlement	-	1	1	1	1
City	3	2	-	-	-
Development card	1	1	-	1	-

A *development card* is drawn from a shuffled stack of *knight*, *progress*, and *victory point* cards. Victory points are how the game ends: a player wins once they accrue 10 or more victory points (see the Victory Points section below). We consider the game’s core mechanics to be characterized by the above rules where the *only* development cards are victory points.

The other development cards, and more complex, special-case mechanics are described next. We list these mainly to supply the palette of game design ideas from which we will draw to give interesting use cases for formalization.

Game Setup

The official rules of the game supply two possibilities for setting up the game: one in which a provided board determines the locations of the hexes and their dice roll values (see Figure 5.2), the other of which is for a *variable* board, recommended for advanced players, wherein hexes and their corresponding dice roll values are randomly determined.

In either case, the players also do two rounds of initial settlement placement: they get to place one settlement and one road adjacent to that settlement in an open intersection and edge during each round. For more details on initial placement, see the rulebook [Teu07].

The Robber

A single “desert” hex tile always appears on the board. This tile produces no resources. A piece called the *robber* is placed initially in the desert. It moves when the dice roll totals to 7 (which does not otherwise trigger any resource production). Whoever makes that dice roll decides which adjacent hex the robber should move to. Whenever the robber is on a hex, that hex does not produce any resources.

Knights

A *knight* is a development card that a player may keep in their hand and play on their turn. It allows them to move the robber to an adjacent hex.

Figure 5.2: Settlers of Catan beginner's starting map.

Illustration A

STARTING MAP FOR BEGINNERS

To make it as easy as possible for you to get started with *Catan*, we use an award-winning rules system, which consists of 3 parts—the *Overview*, the *Game Rules*, and the *Almanac*.

If you've never played *Catan*, please read the *game Overview* first—it's on the back cover of this booklet. Next, read the *Game Rules* and start to play. And finally, if you have questions during the game, please consult the *Almanac* (it begins on page 6).

Begin the game with the resource cards produced by the settlements marked with white stars. See ☆

ODDS FOR DICE ROLLS

2 & 12 = 3%

3 & 11 = 6%

4 & 10 = 8%

5 & 9 = 11%

6 & 8 = 14%

7 = 17%

RESOURCE PRODUCTION

Hills Produce Brick	Forest Produces Lumber	Mountains Produce Ore	Fields Produce Grain	Pasture Produces Wool	Desert Produces Nothing

Maritime Trade

On a player's turn, they can trade resources without involving another player at an exchange rate of 4 : 1—that is, they may trade four identical resource cards (to the ambient supply) for one resource card of their choice. *Harbors*, or settlements built at special points on the edge of the game board, can enable more effective maritime trade rates. (See the rulebook for details [Teu07].)

Progress Cards

Another kind of card that can be drawn from the pool of development cards is a *progress* card, each of which grant the player some one-time special ability when they play it (which they are able to do on their turn). Progress cards include:

- **Road Building:** When the player plays this card, they may immediately place two free roads on the board (according to normal building rules).
- **Year of Plenty:** When the player plays this card, they may immediately take any two resource cards from the supply.
- **Monopoly:** When the player plays this card, they must name one type of resource, and then the other players must give them all of the resource cards of that type from their hands.

Victory Points

Several facets of the game count toward victory points. Players win when it is both their turn *and* they have 10 or more victory points.

- A settlement is worth one victory point.
- A city is worth two victory points.
- Whoever has the longest road (of at least five individual road pieces) has two additional victory points (this designation can change throughout the game).
- Whoever has the “largest army” (number of Knight cards played) of at least three plays has two additional victory points (similar to the longest road designation).
- A victory point card is worth one victory point.

5.3.2 Discussion of the Game's Design

We have described the game's rules in more detail than necessary for our case study, but we want to give the reader a sense of appreciation for the numerous subtleties and edge cases. At its heart, *Settlers* is a simple game of territory acquisition and resource management; undoubtedly, its designer started with those ideas and recognized the need for more intricate rules through iteration and playtesting.

Therefore, to study its design pragmatically, we start with a codification of a single central mechanic. We identify the exchange of resources for building settlements, cities,

and roads as such, but there are other candidates: the randomness of resource production is another (amplified by other randomness-introducing elements, like the Robber); the socioeconomics of the *trading* mechanic is yet another; and finally, the intricate dynamics created by the geometry of the board is another—all of the faces, edges, and intersections formed by the hexagons are interdependent and relevant to play.

5.3.3 Iteration 1

Code for this section is available at https://github.com/chrisamaphone/interactive-lp/blob/master/examples/siedler_core.cep.

In the first iteration, we model the central mechanic of *exchanging resources for development* from the point of view of a single player. We will illustrate the dichotomy between player choices (in the building phase) and the “external” choice of the random deck draw in the selection of development cards. However, we will not model the meaning of development cards in this iteration.

Resources are represented as predicates (i.e. resources in the meta-logic, conveniently enough).

```
brick : pred.
lumber : pred.
road : pred.
wool : pred.
grain : pred.
settlement : pred.
ore : pred.
city : pred.
development_card : pred.
knight : pred.
progress : pred.
victory_point : pred.
```

We have two predicates to sequentialize turn taking between the player and the random draw from the deck.

```
turn : pred.
success : pred.
```

The first stage is a collection of rules for transforming basic resources (brick, lumber, wool, grain, and ore) into settlements, cities, roads, and development cards.

```
stage building = {

build_road : turn * brick * lumber -o road * success.
build_settlement : turn * brick * lumber * wool * grain
  -o settlement * success.
build_city : turn * ore * ore * ore * grain * grain
  -o city * success.
build_development : turn * ore * wool * grain
  -o development_card * success.
```

```

}
#interactive building.

When the building stage quiesces, we transition to the development card drawing
stage to process any possible development card draws. There is one rule in this stage
per possible development card.

qui * stage building * success -o stage draw_dev_card.

stage draw_dev_card = {

dev_knight : development_card -o knight.
dev_prog   : development_card -o progress.
dev_victory_point : development_card -o victory_point.

}

qui * stage draw_dev_card -o stage building * turn.

```

Finally, we specify an initial state with a somewhat arbitrary collection of initial resources for testing.

```

context init =
{turn,
 brick, brick, brick, lumber, lumber, lumber, wool, wool, wool,
 grain, grain, grain, ore, ore, ore}.

#trace _ building init.

```

Sample Interaction

In Figure 5.3, we show two distinct ways of partitioning the available resources according to Settlers' building mechanics.

5.3.4 Iteration 2

Code for this section is available at <https://github.com/chrisamaphone/interactive-lp/blob/master/examples/siedler1.cep>.

In the second iteration of our formalization of Settlers, we introduce multiple players with hands of resources, the production of resources by settlements and cities, and trading. The building of settlements is still not constrained by the geometry of the board, but the production of resources is. We will use a miniature version of the game board, too (but large enough to provide variability in play).

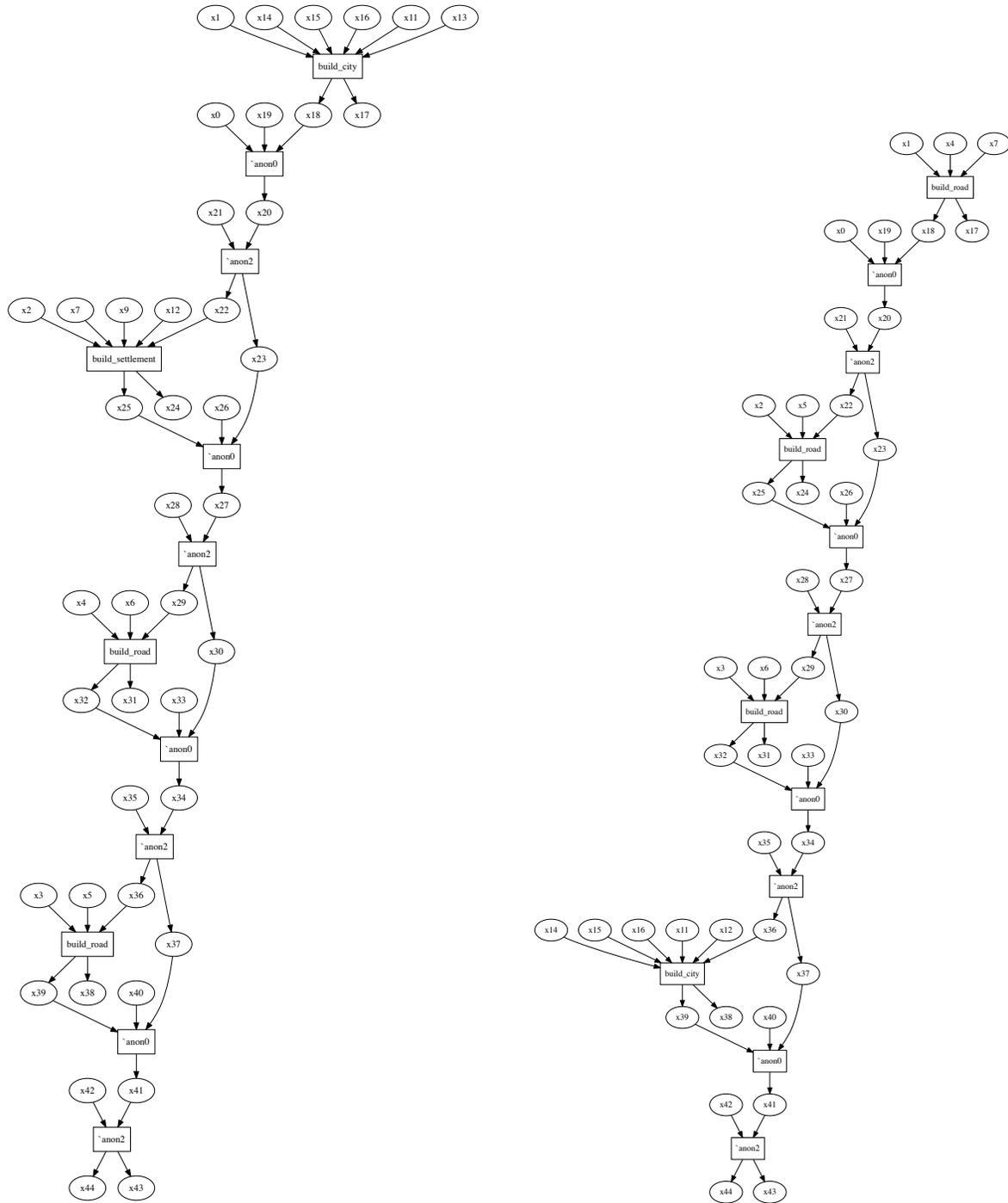
We model players as inhabitants of a `player` type related by a turn ordering. The `turn` and `done` predicates are used to manage turn taking.

```

player : type.
red : player. blue : player. yellow : player.

```

Figure 5.3: Two possible play traces under the Settlers building mechanics.



```
% turn order
next player player : bwd.
next red blue.
next blue yellow.
next yellow red.
```

```
turn player : pred.
done player : pred.
```

Basic resources are now terms rather than predicates; their predicate form is additionally indexed by a player.

```
resource : type.
brick : resource.
lumber : resource.
wool : resource.
grain : resource.
ore : resource.
```

```
holds player resource : pred.
```

We establish a small board made of six hexes, each individual inhabitants of the hex type. We associate with each hex the resource that it produces.

```
hex : type.
h1 : hex. h2 : hex. h3 : hex. h4 : hex. h5 : hex. h6 : hex.
```

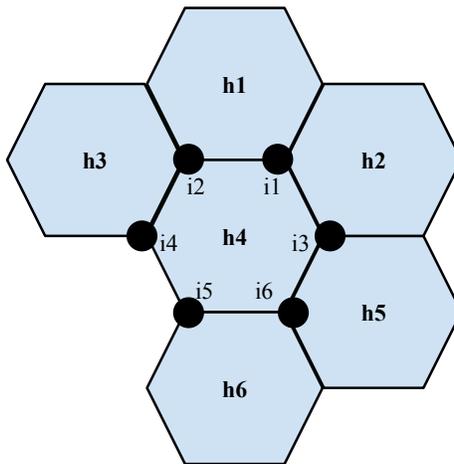
```
produces hex resource : bwd.
produces h1 grain.
produces h2 lumber.
produces h3 ore.
produces h4 wool.
produces h5 grain.
produces h6 brick.
```

We introduce six intersections, each at one point of the central hex. The locations of these intersections relative to the hexes are specified by associating each hex with a *list* of intersections it touches. The map we represent corresponds to the one shown in Figure 5.4.

```
intersection : type.
i1 : intersection.
i2 : intersection.
i3 : intersection.
i4 : intersection.
i5 : intersection.
i6 : intersection.
```

```
ilist : type.
nil : ilist.
```

Figure 5.4: Sample map for second iteration of Settlers of Catan encoding.



```
cons intersection ildist : ildist.
```

```
adjacent hex ildist : bwd.
```

```
adjacent h1 (cons i1 (cons i2 nil)).
```

```
adjacent h2 (cons i1 (cons i3 nil)).
```

```
adjacent h3 (cons i2 (cons i4 nil)).
```

```
adjacent h4
```

```
  (cons i1 (cons i2 (cons i3 (cons i4 (cons i5 (cons i6 nil)))))).
```

```
adjacent h5 (cons i3 (cons i6 nil)).
```

```
adjacent h6 (cons i5 (cons i6 nil)).
```

For each intersection, we track whether it is available (empty) or has a settlement or city at it.

```
available_land intersection : pred.
```

```
has_settlement player intersection : pred.
```

```
has_city player intersection : pred.
```

We specify the starting board with all intersections available. We specify (without loss of generality) the red player starting first, and give each player a token for initialization. In this minimal version of the game, each player gets to place only *one* settlement, as codified in the setup stage:

```
context board =
```

```
{ available_land i1,
```

```
  available_land i2,
```

```
  available_land i3,
```

```
  available_land i4,
```

```
  available_land i5,
```

```
  available_land i6 }.
```

```

init_settlement player : pred.
context players =
{turn red,
  init_settlement red,
  init_settlement yellow,
  init_settlement blue }.

context init = {board, players}.

stage setup = {
  % everyone gets one settlement to start,
  % going in random order.
  settle
  : init_settlement P * available_land X
  -o has_settlement P X.
}
#interactive setup.

```

For dice rolling, we introduce a die predicate (an unrolled die) and a stage with as many rules as possible rolls, each causing the die to transform into an indication of a certain hex.

```

die : pred.
setup_to_rolling : qui * stage setup -o stage roll * die.

roll hex : pred.
stage roll = {
  roll1 : die -o roll h1.
  roll2 : die -o roll h2.
  roll3 : die -o roll h3.
  roll4 : die -o roll h4.
  roll5 : die -o roll h5.
  roll6 : die -o roll h6.
}
roll_to_prod : qui * stage roll -o stage produce.

```

Next, we do rule production by propagating the die roll to every intersection adjacent to the corresponding hex. The mete rules process the list of adjacent intersections, stopping when there are none left, skipping empty ones, and introducing new resources into players' hands when there are cities or settlements there.

```

prod intersection : pred.
mete resource ilist : pred.
stage produce = {
  process_roll :
  roll H * produces H R * adjacent H Is
  -o mete R Is.
}

```

```

mete/nil : mete R nil -o ().
mete/skip : mete R (cons I Is) * $available_land I
           -o mete R Is.
mete/settlement
  : mete R (cons I Is) * $has_settlement Player I
    -o holds Player R * mete R Is.
mete/city
  : mete R (cons I Is) * $has_city Player I
    -o holds Player R * holds Player R
      * mete R Is.
}

```

Next is the trading stage. We model trading with a single rule indicating that any two cards in someone's hands can swap places.

```

prod_to_trade : qui * stage produce -o stage trade.

stage trade = {
  do/trade : $turn P * holds P R * holds P' R'
            -o holds P R' * holds P' R.
  donetrading : turn P -o done P.
}
#interactive trade.

```

This simplistic model of trading fails to capture a few important details, such as the ability to trade many-for-one or one-for-many cards. Also, the default presentation of all available rules with the `#interactive` mechanism reveals all of the resource cards in a player's hand (information that is not actually secret except by virtue of the limitations of human memory, but still intended to be kept hidden). However, for our second iteration, still meaning only to capture the essence of the game, this model suffices.

The next stage after trading is building. The building rules look similar to those in Iteration 1, except that they refer to the availability of an intersection (for settlements) and to the prior establishment of a settlement (for cities). The player may build arbitrarily many times, so the turn token is preserved in each rule application, until they explicitly choose to be done with the finished rule.

```

trade_to_build : qui * stage trade * done P -o stage build * turn P.

stage build = {
  build_settlement :
    $turn P
    * holds P brick * holds P lumber * holds P wool * holds P grain
    * available_land L
    -o has_settlement P L.

  build_city :
    $turn P
    * has_settlement P I * holds P ore * holds P ore * holds P ore
    * holds P grain * holds P grain
}

```

```

    -o has_city P I.

    finished : turn P -o done P.
  }
  #interactive build.

```

Finally, the building stage changes the turn to the next player upon quiescence and transfers control back to the roll stage.

The top-level program begins in the setup stage and with the initial context provided at the top of the program.

```

build_to_roll :
  qui * stage build * done P * next P P'
  -o stage roll * turn P' * die.

#trace _ setup init.

```

Sample Interaction

We can play through this version of the game with the following transition sequence, starting with setup and concluding with the red player building a second settlement:

```

(settle red i1)
(settle blue i6)
(settle yellow i4)
roll6
(process_roll h6 brick (cons i5 (cons i6 nil)))
(mete/skip brick i5 (cons i6 nil))
(mete/settlement brick i6 nil blue)
(mete/nil brick)
(donetrading red)
(finished red)
roll2
(process_roll h2 lumber (cons i1 (cons i3 nil)))
(mete/settlement lumber i1 (cons i3 nil) red)
(mete/skip lumber i3 nil)
(mete/nil lumber)
(donetrading blue)
(finished blue)
roll2
(process_roll h2 lumber (cons i1 (cons i3 nil)))
(mete/settlement lumber i1 (cons i3 nil) red)
(mete/skip lumber i3 nil)
(mete/nil lumber)
(donetrading yellow)
(finished yellow)
roll4
(process_roll h4 wool (cons i1 (cons i2 (cons i3 (cons i4 (cons i5 (cons i6 nil)))))))

```

```

(mete/settlement wool i1 (cons i2 (cons i3 (cons i4 (cons i5 (cons i6 nil))))) red)
(mete/skip wool i2 (cons i3 (cons i4 (cons i5 (cons i6 nil)))))
(mete/skip wool i3 (cons i4 (cons i5 (cons i6 nil))))
(mete/settlement wool i4 (cons i5 (cons i6 nil)) yellow)
(mete/skip wool i5 (cons i6 nil))
(mete/settlement wool i6 nil blue)
(mete/nil wool)
(do/trade red lumber blue brick)
(donetrading red)
(finished red)
roll3
(process_roll h3 ore (cons i2 (cons i4 nil)))
(mete/skip ore i2 (cons i4 nil))
(mete/settlement ore i4 nil yellow)
(mete/nil ore)
(donetrading blue)
(finished blue)
roll1
(process_roll h1 grain (cons i1 (cons i2 nil)))
(mete/settlement grain i1 (cons i2 nil) red)
(mete/skip grain i2 nil)
(mete/nil grain)
(donetrading yellow)
(finished yellow)
roll5
(process_roll h5 grain (cons i3 (cons i6 nil)))
(mete/skip grain i3 (cons i6 nil))
(mete/settlement grain i6 nil blue)
(mete/nil grain)
(donetrading red)
(build_settlement red i5)

```

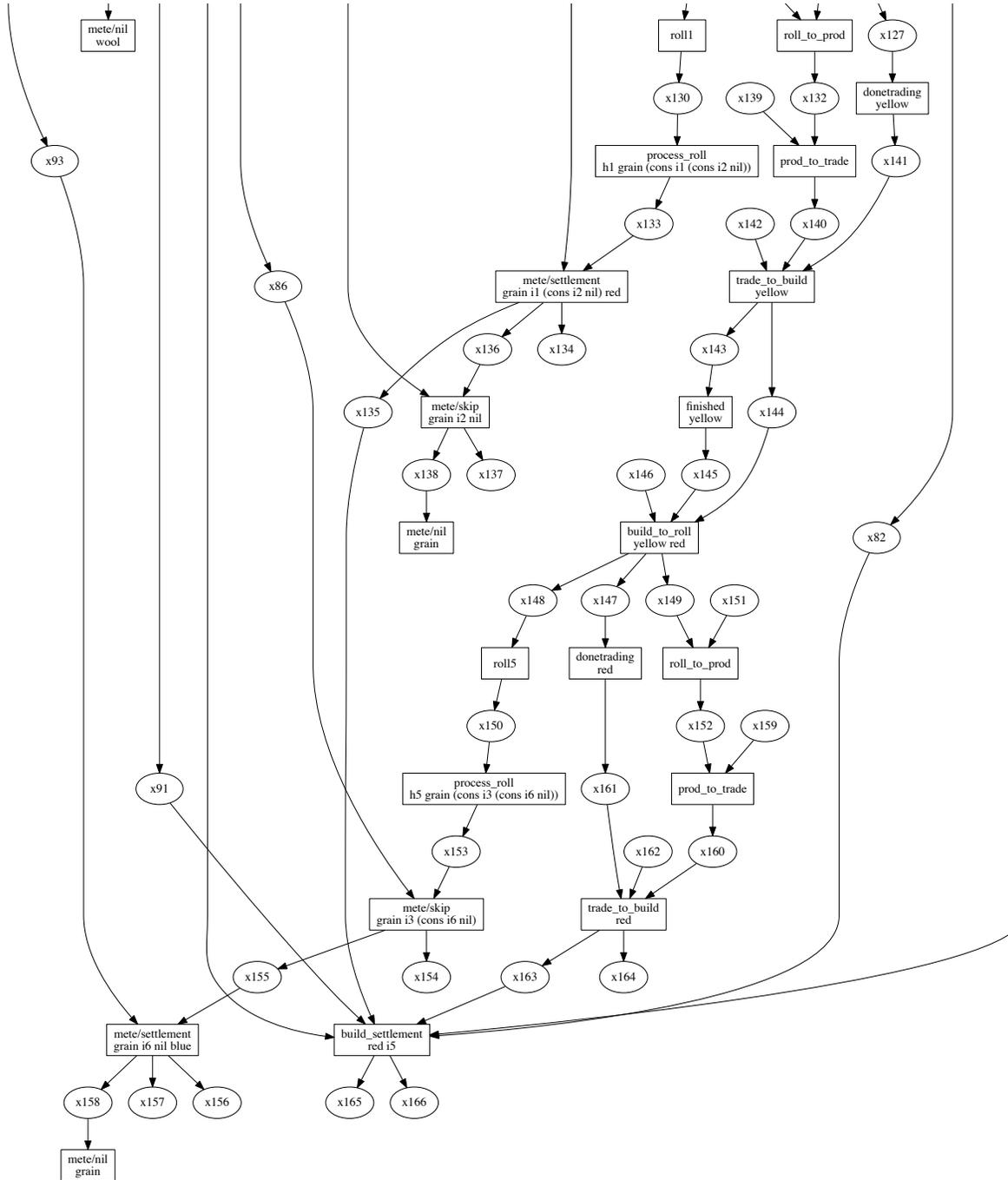
The causal structure of the final sequence can be found in Figure 5.5.

5.3.5 Discussion

The second iteration we have presented is a playable prototype of all of the game's core rules, amenable to further iteration. There are a few ways in which it fails to capture some of the subtleties of Settlers' mechanics:

- *Trade*: as mentioned briefly, the formalization of the trading mechanic does not quite model what is interesting about trading in Settlers. If we think of hands of cards as secret information, then trading really emphasizes a model of distributed information, which would map onto distributed proof search in the formalism. An interesting future research direction might be to investigate epistemic modal logic as a basis for modeling this knowledge distribution.

Figure 5.5: Trace graph for part of a Settlers of Catan playthrough.



Additionally, the trading stage of Settlers follows a *protocol* of offering and accepting trades that is not modeled in this formalism. An offer might be rejected or replaced with a counter-offer, and only once both parties agree will it be accepted. We could potentially extend this formalization to account for the peer-to-peer player interaction in more detail.

- *Building*: we have not expressed the constraints on building described by the Settlers rule book, i.e. the requirement that new settlements be built next to roads owned by the building player and the requirement that new roads be built next to anything owned by the building player. It should be straightforward to introduce those constraints in a future iteration, although tracking *edges* between hexes in addition to intersections requires careful modeling.

Additionally, there are several aspects of Settlers' play that could be built into further iterations on this encoding, each of which might only be introduced to balance some other shortcoming of the simpler set of mechanics, determined through playtesting:

- *Board map*: being able to express arbitrarily large and complex board layouts, at least up to the size and expressiveness of the physical board game, should be possible. We would need to introduce another stage and several more predicates to create the appropriate geometric constraints.
- *Win conditions*: we have not modeled victory points nor ending conditions based on them, but this should be straightforward, especially since they can only take effect for a player during their turn.
- *Random factors*: things like the robber, development cards, and alternative routes to victory points (longest road and largest army) add additional unpredictability into the game that can lead to subtle strategies.
- *Maritime trade*: we have not modeled maritime trade in this iteration of the game, though it would be straightforward to add it.
- *Finite resource pools*: in Settlers, each player can only build up to the allowance of their initially-allotted supply of cities, settlements, and roads. Our encoding currently allows for building as many of these as players' hands of cards allow.

5.3.6 Potential Analyses

Even with the simplicity of our second iteration compared to the full game, we can still learn from it. By removing the `#interactive` directives from the stages and generating enough random play traces, we could perform analysis to answer the following questions:

- Is there a set of initial conditions, e.g. board size, number of tiles of each resource, et cetera, that leads to more balanced games? One way to answer this would be to randomly generate those conditions in the setup stage, then measure "balance" by the proportion of wins by each player on a sufficiently large playthrough sample. This technique resembles that used by the Ludi system [Bro08] for the procedural generation of game rules via evolutionary algorithms.

- If we were to build a physical version of the game, what is the right number of cities, settlements, and resource cards to include? We can answer this question by doing a resource usage analysis of the game playthroughs, noting the maximum number of each predicate in the context corresponding to the physical resource we are interested in (as well as averages of those numbers).

This concludes our Settlers of Catan case study.

5.4 Garden Simulator

Our next case study is an original simulation design built using an iterative process like that described in Section 5.3. This study is an open-ended simulation that puts the player in the role of a gardener.

5.4.1 Game Idea

At a high level, we want to use autonomous proof search to model the growth of plants in a garden, and interactive proof search to tend the garden. We would like to create interesting tension between expanding the variety and size of the garden and keeping all of the plants healthy. There should be several types of plant, each of which takes up some soil space, grows if given the proper amount of water, and dies otherwise. Once plants have reached maturity, they may be harvested for new seeds.

To introduce tension between expansion and plant health, plants need to require just enough attention from the gardener (player) that on any given turn, the decision between planting something new and tending to the needs of existing plants is difficult. Real-world phenomena that we might want to model include the presence of weeds, overgrowth of some plants we are trying to grow (that might impede the growth of other plants in the garden), sensitivity to overwatering, and plant pests or diseases.

5.4.2 Iteration 1

Code for this section is available at <https://github.com/chrisamaphone/interactive-lp/blob/master/examples/garden-small.cep>.

For the first iteration of the game, we will just model the basic mechanic: planting, watering, and harvesting plants as player actions; and growth and death of plants as autonomous actions.

We start with three types of plants that will not be distinguished by the mechanics.

```
plant_type : type.  
flower : plant_type.  
herb : plant_type.  
vine : plant_type.
```

Plants have three growth phases (seed, seedling, and mature) and three health states (healthy, wilting, and dead).

```

plant_phase : type.
seed : plant_phase.
seedling : plant_phase.
mature : plant_phase.

next plant_phase plant_phase : bwd.
next seed seedling.
next seedling mature.

health : type.
healthy : health.
wilting : health.
dead : health.

```

A plant can be harvested, in seed form (not planted), or planted at a particular place in the garden with a particular growth phase and health, as modeled by the following predicates.

```

harvested plant_type : pred.
have_seed plant_type : pred.
plant plant_type plant_phase health : pred.

```

The play stage models gardener actions: planting, watering, harvesting, clearing away dead plants, and extracting seeds from harvested plants. Planting requires soil space. The stage is provided with one step token, which is consumed and re-produced by every action except for done, meaning that we can perform as many actions as we want before ending our turn and passing control back to the autonomy of the garden.

```

step : pred.
soil_space : pred.

stage play {
  get_seeds : $step * harvested Type -o have_seed Type * have_seed Type.
  plant : $step * have_seed Type * soil_space -o plant Type seed healthy.
  harvest : $step * plant Type mature healthy -o harvested Type * soil_space.
  clear : $step * plant Type _ dead -o soil_space.
  water : $step * plant Type Phase wilting -o plant Type Phase healthy.
  done : step -o ().
}
#interactive play.

qui * stage play -o stage grow * step.

```

The grow stage may also take an arbitrary number of steps before passing control back to the gardener. Its possible actions are growing, wilting, and dying.

```

stage grow = {
  grow : $step * plant Type Phase healthy * next Phase Phase'
        -o plant Type Phase' healthy.
  wilt : $step * plant Type Phase healthy

```

```

        -o plant Type Phase wilting.
    die : $step * plant Type Phase wilting
        -o plant Type Phase dead.
    done : step -o ().
}
qui * stage grow -o stage play * step.

```

We test the game with three starting seeds and four soil spaces.

```

context init = {have_seed flower, have_seed herb, have_seed vine, step,
                soil_space, soil_space, soil_space, soil_space}.
#trace _ play init.

```

Sample Interaction

In Figure 5.6, we give a partial trace graph for a sample interaction with the system.

5.4.3 Iteration 2

Code for this section is available at <https://github.com/chrisamaphone/interactive-lp/blob/master/examples/garden-med.cep>.

In our next iteration, we introduce some additional synchronization in the form of a day counter.²

```

nat : type.
z : nat.
s nat : nat.

day nat : pred.

```

The plant types, health and maturity levels are the same as before. And, as before, a plant can either be harvested, in seed form, or planted, but the plant predicate now tracks some additional information for synchronization: the last day it has taken a step in the grow stage.

```

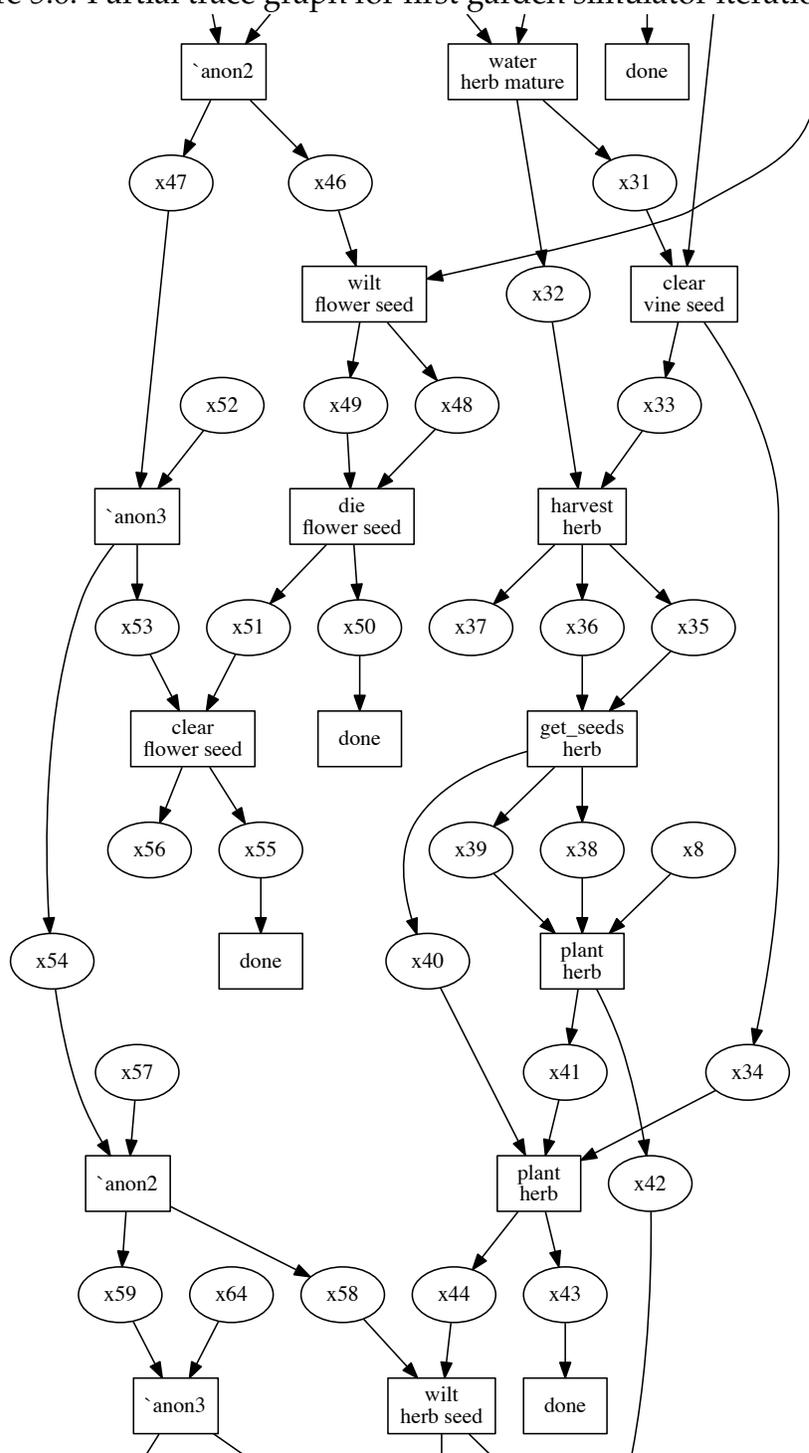
harvested plant_type : pred.
have_seed plant_type : pred.
% last var is for synchronization
plant plant_type plant_phase health nat : pred.

```

We give the player three turns in this iteration. Planting a new seed marks its last argument with the current day D .

²In this example, we will represent numbers in unary notation: z for 0 and $s N$ for $N+1$. Ceptre's implementation allows for the designation of user-defined natural numbers in this fashion to be mapped to built-in integers, thus giving the programmer some flexibility in which notation to use—note that the unary notation is convenient for the sake of pattern-matching in the premises to rules, but treating numbers as built-in integers has the advantage that operators like multiplication and addition can be written in-line without defining them as logic programs.

Figure 5.6: Partial trace graph for first garden simulator iteration.



```

step : pred.
turns nat : pred.
soil_space : pred.

stage play {
  get_seeds :
    turns (s N) * harvested Type
      -o have_seed Type * have_seed Type * turns N.

  plant :
    turns (s N) * have_seed Type * soil_space * $day D
      -o plant Type seed healthy D * turns N.

  harvest : turns (s N) * plant Type mature healthy _
      -o harvested Type * soil_space * turns N.

  clear : turns (s N) * plant Type _ dead _
      -o soil_space * turns N.

  water : turns (s N) * plant Type Phase wilting D
      -o plant Type Phase healthy D * turns N.

  done : turns _ -o ().
}
#interactive play.
qui * stage play -o stage advance_day * step.

Before the growth stage, we advance the day.

stage advance_day = {
  advance : day N * step -o day (s N).
}
qui * stage advance_day -o stage grow.

```

In the growth stage, each plant only takes a step if its day index lags behind the current day, and the step updates it to be current. This technique enforces that each plant takes exactly one turn of growth or ailment in this stage. (This technique for enforcing that behavior is an alternative to the `npc_turn` trick used in Section 5.1).

```

stage grow = {
  grow : plant Type Phase healthy D * next Phase Phase' * $day (s D)
      -o plant Type Phase' healthy (s D).
  wilt : plant Type Phase healthy D * $day (s D)
      -o plant Type Phase wilting (s D).
  die : plant Type Phase wilting D * $day (s D)
      -o plant Type Phase dead (s D).
}
qui * stage grow -o stage play * turns (s (s (s z))).

```

```

context init = {day z,
  have_seed flower, have_seed herb, have_seed vine, have_seed succulent,
  soil_space, soil_space, soil_space, soil_space, turns (s (s (s z)))}.
#trace _ play init.

```

Sample Interaction

In Figure 5.7, we give a partial trace graph for a sample interaction with the system.

5.4.4 Iteration 3

Code for this section is available at <https://github.com/chrisamaphone/interactive-lp/blob/master/examples/garden-iter3.cep>.

In our third iteration, we introduce concrete terms for the spaces that plants can inhabit and allow the player to trade harvested plants for space upgrades. We also fix the bug from the second iteration by adding a bit of extra state (watered/needing water) and adding a drinking action to the autonomous growth stage.

The structure of this iteration now resembles the action/reaction model from the social story world described in Section 5.1.

The code below starts by introducing a type for garden space and giving the predicates over spaces (omitting the type header common to prior iterations):

```

space : type.
s1 : space. s2 : space. s3 : space. s4 : space.
s5 : space. s6 : space. s7 : space. s8 : space.

max_space : pred.

next_space space space : bwd.
next_space s4 s5.
next_space s5 s6.
next_space s6 s7.
next_space s7 s8.

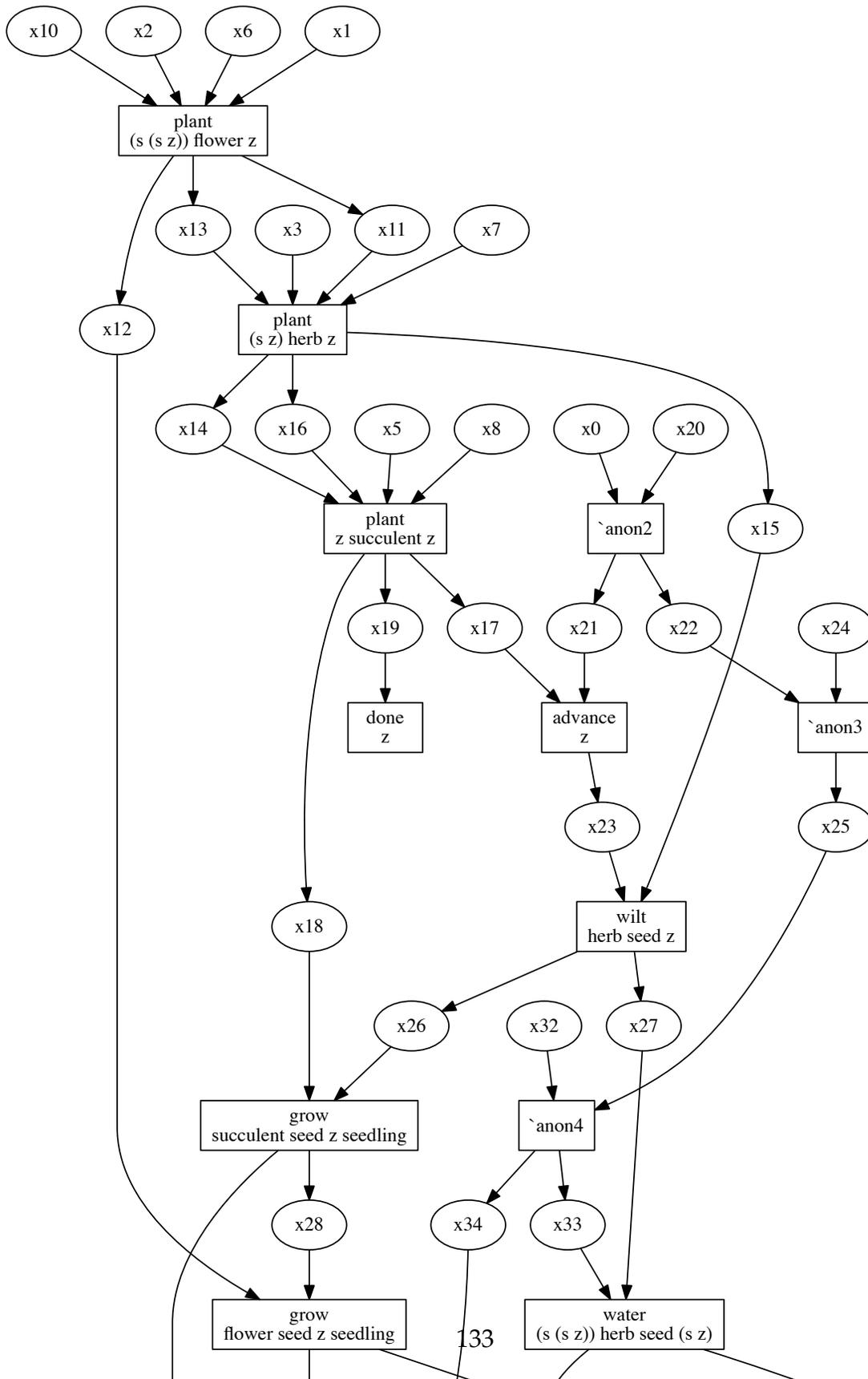
empty space : pred.
plant_at space : pred.
plant_type_at space plant_type : pred.
plant_phase_at space plant_phase : pred.
health_at space : pred.
needs_water space : pred.
watered space : pred.

```

Note that individual properties of the plant space may now be specified independently, using the identifier of the soil space as a tag to link all of those properties.

The play stage now includes an option to buy more soil space, and watering introduces a watered fact distinct from the plant's health:

Figure 5.7: Partial trace graph for second garden simulator iteration.



```

step : pred.
turns nat : pred.

stage play {
  get_seeds :
    turns (s N) * harvested Type
      -o have_seed Type * have_seed Type * turns N.

  plant :
    turns (s N) * have_seed Type * empty S * $day D
      -o plant_at S
        * plant_type_at S Type
        * plant_phase_at S seed
        * health_at S healthy
        * needs_water S
        * turns N.

  harvest :
    turns (s N) * plant_at S
      * plant_phase_at S mature * health_at S healthy
      * plant_type_at S Type
      -o harvested Type * empty S * turns N.

  clear :
    turns (s N) * plant_at S * health_at S dead
      * plant_phase_at S _ * plant_type_at S _
      -o empty S * turns N.

  water :
    turns (s N) * needs_water S
      -o watered S * turns N.

  buy_space :
    turns (s N) * harvested P
      * max_space S * next_space S S'
      -o empty S' * max_space S'.

  done : turns _ -o ().
}
#interactive play.

```

Now that we have separated out the plant type, plant phase, and health information of a soil space, it is easier to use the “generate one turn per plant” technique wherein one predicate (`plant_at`) is replaced by a temporary turn predicate (`plant_turn`), then consumed and replaced with the original predicate in each rule of the growth stage. Note that this trick depends on carefully maintaining the invariant that every plant will consume a `plant_turn` atom and produce a `plant_at` atom during the grow stage.

Maintaining this invariant was the source of several bugs found while developing the example.

```
qui * stage play -o stage advance_day.

plant_turn space : pred.
stage advance_day = {
  gen_turn : plant_at S -o plant_turn S.
}

qui * stage advance_day -o stage grow.

stage grow = {
  grow : plant_turn S * $health_at S healthy * watered S
        * plant_phase_at S Phase * next Phase Phase'
        -o plant_at S * plant_phase_at S Phase' * needs_water S.

  wilt : plant_turn S * health_at S healthy * $needs_water S
        -o plant_at S * health_at S wilting.

  drink : plant_turn S * health_at S wilting * watered S
        -o plant_at S * health_at S healthy * needs_water S.

  die : plant_turn S * health_at S wilting * needs_water S
        -o plant_at S * health_at S dead.

  none/dead
    : plant_turn S * $health_at S dead -o plant_at S.

  none/mature
    : plant_turn S * $health_at S healthy * watered S
      * $plant_phase_at S mature
      -o plant_at S * needs_water S.

  needwater/empty
    : $empty S * needs_water S -o ().

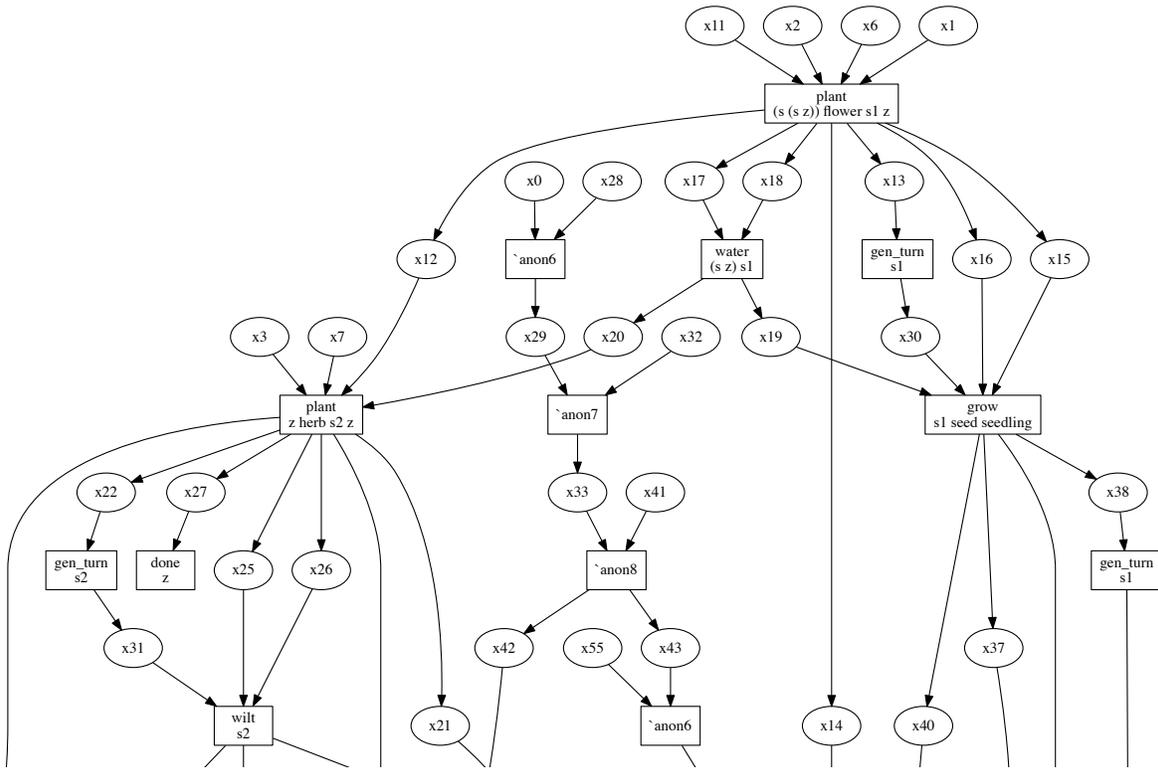
  watered/empty
    : $empty S * watered S -o ().
}

qui * stage grow -o stage play * turns (s (s (s z))).

context init = {day z,
  have_seed flower, have_seed herb, have_seed vine, have_seed succulent,
  empty s1, empty s2, empty s3, empty s4, max_space s4,
  turns (s (s (s z)))}.

#trace _ play init.
```

Figure 5.8: Partial trace graph for third garden simulator iteration.



Sample Interaction

In Figure 5.8, we give a partial trace graph for a sample interaction with the system.

5.4.5 Discussion

With this garden simulation case study, we have demonstrated an iterative design process for an original game mechanic based on interacting with a simulated, virtual world. Potential extensions that could be developed in future iterations include varying the plant types, such as making it so that succulents require less water, and modeling plant overgrowth, such as vines becoming arbitrarily long and threatening the health of other plants.

The plant growth stage can be seen as an application of generative methods, albeit a simple one akin to a state machine. Generative methods have perhaps been more applied to the domain of plant growth (e.g. tree generation) than any other domain [LD99, GPSL03, DHL⁺98, Pru86], and we posit that more sophisticated techniques could be modeled as an extension to this prototype to interesting effect, especially with visual rendering.

This model could be considered a prototype for a more extensive game about the creation and maintenance of peaceful worlds, an under-explored space of design in mainstream games. We have aimed to illustrate with this example the use of Ceptre, and especially programming idioms that have been common to other examples, for the design of a novel operational logic based on simple generative techniques.

5.5 Tamara

Tamara is a work of participatory theater by John Krizanc [Kri89]. The form of participation is quite limited: guests are allowed to *follow* characters when they depart a scene, after which point the departing characters will participate in a different scene, either by joining an ongoing one or starting their own. We use this play as an example of interactive storytelling that does not involve digital computers, yet has *computational content* in a sense: many traversals through the play’s story world are available by way of simultaneous scenes and the option to move between them when a character connects them.

This case study is unlike the others because we do not attempt to implement the interactive experience of attending *Tamara* as an interactive linear logic program. Instead, the study we have carried out pertaining to this play is twofold:

- First, we implement a transformation from the *script* of the play—in particular, information about which characters participate in which scenes, and where they go afterwards—to a dependency diagram that gives a holistic sense for which scenes may co-occur and where synchronization is needed. In some sense, this analysis could be considered a producer or stage manager’s tool for orchestrating the logistics of the play. Let us refer to the dependency diagram as a *scene graph*.

The transformation from script to scene graph is done by encoding the script as a linear logic program, then using proof search to create the scene graph (represented as a concurrently-structured proof term).

- Next, we implement a compiler that takes a proof term (representing the scene graph) along with a *scenes* file, which maps linear logic rules to the text (dialogue) of the scene, and turns it into a Twine game whose passages are scenes and whose links are choices to either follow a character who is exiting or remain with the other characters.

In this way, linear logic programming is only used as an intermediate representation between the informal script and the interactive performance (Twine game).

5.5.1 Tamara’s Story, Script, and Rules

Tamara is a historically-based tale of aristocrats and servants, set in the mansion of Italian poet Gabriele d’Annunzio in the 1920s, pivoting around the event of the Polish artist Tamara de Lempicka’s arrival as an artist in residence. The characters involved

are artists previously invited to the d’Annunzio estate, many of whom d’Annunzio took as lovers, and the serving staff.

The idea for the play came out of a discussion between Krizanc and his friends about a Chekhov play, wherein he expressed wishing he could follow the servants when they exited a scene, because they “were the only ones to know the underbelly of the social fabric.” As such, the characters in *Tamara* include a large cast of servants —Emilia, the maid; Dante and Mario, servants; Finzi, the guard; and Aelis, the housekeeper—in addition to the five wealthy artists who live in or visit the d’Annunzio estate—Luisa, de Spiga, Carlotta, Tamara, and D’Annunzio himself.

From an audience member’s perspective, the play’s mechanics are as follows: after arrival, guests are assembled, then divided into three starting groups. Each group follows a different character, who leads them to a different room in the mansion. At that point, the play begins in earnest, and the standard rules apply: whenever a character exits a room, you may follow them. You must always be following or watching the action of a character—that is, if a character is the *only* one in the room when they exit, you are obligated to follow them. However, you need not follow the same character throughout the play.

The script for the play is organized into standard acts, which are further subdivided *sections* labeled with an alphabetic character *A...U*, each further subdivided into numerically-labeled scenes. Sections designate some approximation of concurrent structure: their endings usually just follow substantial divergence or fragmentation in the characters’ story arcs, and their beginnings give an overview of the pursuant simultaneous scenes (see Figure 5.9).

Note that, despite the story’s “branching” structure, it is actually deterministic, as in, the play is performed the same way every time. In our terminology from Chapter 2, the story is purely *simultaneous* and contains no *alternative* nonlinearity. Thus, the logic program that we write should be deterministic. Lacking metalogical checks on the program to ensure this fact for us, we must do so by maintaining carefully the invariant that no component of narrative state is consumed by multiple rules.

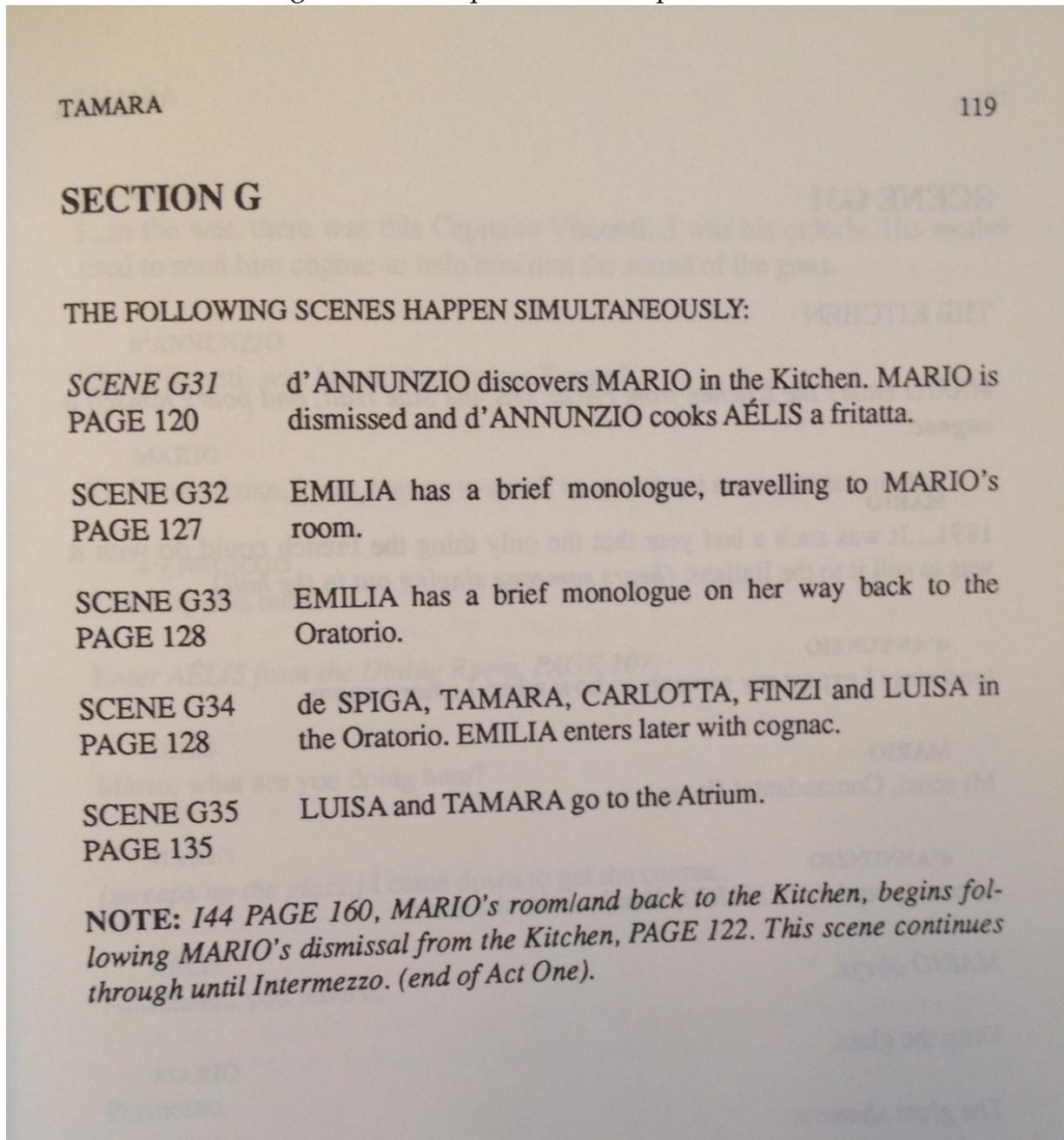
5.5.2 Encoding the Script Structure

Code for this section is available at <https://github.com/chrisamaphone/interactive-lp/blob/master/examples/tamara/tamara.cep>.

First, we must address the question of what a piece of narrative state should be. At first glance, it may seem that the answer is a *character* in the story. But since characters may occur in multiple scenes which need to be ordered with respect to one another, we will need to track more information in order to maintain determinism.

Previously, our approach has been to decompose the story into narrative resources that implicitly drive the causal model underlying the story. For instance, in the *Three Little Pigs* formalization from Chapter 2, we incorporated the narrative resource of the “brick house” state as an output of one scene (the building of the brick house) and input of another (the wolf confronting the pig in the brick house) to ensure that these scenes happened in a plausible causal order.

Figure 5.9: Excerpt from the script of *Tamara*



For this study, however, we are less interested in discovering emergent causal relationships: we just want to encode the one that is already present in the script for the play. Thus, to synchronize scenes deterministically, we make narrative states that represent characters *at a certain time and place*—where time is abstracted as a scene index. Concretely, we set up the encoding of sections *A* through *C* in the script as follows:

```

scene : type.
a1 : scene.  b2 : scene.
b3 : scene.  b4 : scene.  b5 : scene.
b6 : scene.  b7 : scene.  b8 : scene.
b9 : scene.  b10 : scene.  b11 : scene.
c12 : scene.  c13 : scene.  c14 : scene.
d15 : scene.  d16 : scene.  d17 : scene.

location : type.
sidehall      : location.
atrium        : location.
hall          : location.
oratorio      : location.
diningroom    : location.
luisaroom     : location.
dannunziroom : location.
leda          : location.
above-atrium  : location.
finzi-office  : location. % 10 locations.
offstage      : location.

%% characters
tamara  scene location : pred.
luisa   scene location : pred.
emilia  scene location : pred.
finzi   scene location : pred.
despiga scene location : pred.
aelis   scene location : pred.
dannunzio scene location : pred.
carlotta scene location : pred.
dante   scene location : pred.
mario   scene location : pred. % 10 characters.

```

The initial setup positions every character offstage for scene *A1* except for de Spiga, who starts the show, and Tamara, who does not enter until scene *D15*.

```

context
init = {
  tamara  d15 atrium,
  luisa   a1 offstage,
  emilia  a1 offstage,
  finzi   a1 offstage,
  despiga a1 atrium,

```

```

    aelis      a1 offstage,
    dannunzio a1 offstage,
    carlotta  a1 offstage,
    dante     a1 offstage,
    mario     a1 offstage
}.

```

Now we can write rules that correspond to the character coordination that must take place for each scene. The first several rules manage the rapid entering-and-exiting of the introductory scene of the play, which the whole audience actually witnesses as in an ordinary stage play:

```

stage all = {
%%%% SECTION A %%%

a1_dante_finzi
: dante a1 offstage * finzi a1 offstage -o dante a1 atrium * finzi a1 atrium.

a1_emilia_enters
: emilia a1 offstage -o emilia a1 atrium.

a1_dannunzio_enters
: dannunzio a1 offstage -o dannunzio a1 atrium.

a1_dannunzio_exits
: dannunzio a1 atrium -o dannunzio a1 dannunziroom.

a1_emilia_exits
: emilia a1 atrium -o emilia a1 sidehall.

a1_aelis_enters
: aelis a1 offstage -o aelis a1 atrium.

a1_carlotta_enters
: carlotta a1 offstage -o carlotta a1 atrium.

a1_carlotta_and_aelis_exit
: aelis a1 atrium * carlotta a1 atrium
  -o carlotta b4 hall * aelis a1 diningroom.

a1_mario_enters
: mario a1 offstage -o mario a1 atrium.

a1_mario_exits
: mario a1 atrium -o mario d15 atrium.

```

Then the audience is split into three groups, each following one of Dante, Finzi, or de Spiga:

```

a1_groups_split
: dante a1 atrium * finzi a1 atrium * despiga a1 atrium
  -o dante a1 luisaroom
    * finzi a1 diningroom
    * despiga b2 atrium .

```

At this point, the audience is divided and the story becomes truly simultaneous. Some of the groups subdivide further.

```

% Dante takes group 1 to Luisa and D'Annunzio, then heads to the atrium
% alone.

```

```

a1_grp1_luisa
: dante a1 luisaroom * luisa a1 offstage
-o dante a1 dannunziroom * luisa b3 luisaroom.

```

```

a1_grp1_dannunzio
: dante a1 dannunziroom * dannunzio a1 dannunziroom
-o dante b10 atrium * dannunzio b6 dannunziroom.

```

```

% Finzi takes group 2 to Aelis, then Emilia, then meets Carlotta in the
% hall.

```

```

a1_grp2_aelis
: finzi a1 diningroom * aelis a1 diningroom
-o finzi a1 sidehall * aelis b9 diningroom.

```

```

a1_grp2_emilia
: finzi a1 sidehall * emilia a1 sidehall
-o finzi b4 hall * emilia b7 leda.

```

Note the strict pattern that these rules follow: every character predicate on the premise side of the rule must match in scene and room identifier, and must be replaced on the consequent side of the rule with a corresponding character predicate in a later scene.

The remaining rules for sections *B* and *C* of the play follow this same pattern, and are produced below for completeness, although we do not expect them to be particularly meaningful to the reader without the script of the play itself.

```

%%% SECTION B %%%

```

```

% Group 3 stays in the Atrium with de Spiga.

```

```

b2_despiga_monologue
: despiga b2 atrium -o despiga b10 atrium.

```

```

b3_luisa_monologue
: luisa b3 luisaroom -o luisa b11 dannunziroom.

```

```

b4_carlotta_kisses_finzi
: finzi b4 hall * carlotta b4 hall
-o finzi b5 hall * carlotta b8 leda.

```

b5_finzi_monologue
: finzi b5 hall -o finzi c12 atrium.

b6_dannunzio_monologue
: dannunzio b6 dannunziroom -o dannunzio b11 dannunziroom.

b7_emilia_monologue
: emilia b7 leda -o emilia b8 leda.

b8_emilia_carlotta
: emilia b8 leda * carlotta b8 leda
-o emilia c12 diningroom * carlotta c12 atrium.

b9_aelis_monologue
: aelis b9 diningroom -o aelis c12 atrium.

b10_dante_despiga
: dante b10 atrium * despiga b10 atrium
-o dante c12 atrium * despiga c12 atrium.

b11_dannunzio_luisa
: dannunzio b11 dannunziroom * luisa b11 dannunziroom
-o dannunzio c12 atrium * luisa c12 above-atrium.

%%% SECTION C %%%

c12_finzi_exit
: %% de Spiga, Dante, Aelis, Finzi, Carlotta, D'Annunzio
 despiga c12 atrium
* dante c12 atrium
* aelis c12 atrium
* finzi c12 atrium
* carlotta c12 atrium
* luisa c12 above-atrium
-o despiga c12 atrium
 * dante c12 atrium
 * aelis c12 atrium
 * finzi c12 finzi-office
 * carlotta c12 atrium
 * luisa c12 above-atrium.

c12_gunshot
: luisa c12 above-atrium * finzi c12 finzi-office * emilia c12 diningroom
-o luisa c12 atrium * finzi c12 atrium * emilia c12 atrium.

```

c12_emilia_carlotta_aelis_luisa_leave_atrium
: emilia c12 atrium * carlotta c12 atrium * aelis c12 atrium
  * luisa c12 atrium
-o emilia c13 sidehall * carlotta c13 sidehall * aelis c13 sidehall
  * luisa c13 atrium.

c13_finzi_luisa_confrontation
: finzi c12 atrium * luisa c13 atrium
-o finzi d15 atrium * luisa c13 sidehall.

c13_emilia_aelis_carlotta_luisa_sidehall
: emilia c13 sidehall * aelis c13 sidehall * carlotta c13 sidehall
* luisa c13 sidehall
%% all exit back to atrium
-o emilia c13 atrium * aelis c14 atrium * carlotta c13 atrium
* luisa c13 atrium.

c12_take_presents_to_leda
: dante c12 atrium * carlotta c13 atrium * luisa c13 atrium
* emilia c13 atrium
-o dante c14 leda * carlotta c14 leda * luisa c14 leda
* emilia c14 atrium.

c14_emilia_enter_leda
: dante c14 leda * carlotta c14 leda * luisa c14 leda * emilia c14 atrium
-o dante c14 leda * carlotta c14 leda * luisa c14 leda * emilia c14 leda.

c14_all_exit_leda
: dante c14 leda * carlotta c14 leda * luisa c14 leda * emilia c14 leda
-o dante c14 atrium * carlotta c14 atrium * luisa c14 atrium
  * emilia c14 atrium.
%% n.b. technically they go "back" to scene c12, but this disrupts
%% a useful form of stratification...

c_final
: luisa c14 atrium * carlotta c14 atrium * dante c14 atrium
* emilia c14 atrium * aelis c14 atrium * dannunzio c12 atrium
* despiga c12 atrium
-o luisa d17 oratorio * carlotta d17 oratorio
  * aelis d17 oratorio * despiga d17 oratorio
  * dante d15 atrium * dannunzio d15 atrium
  * emilia d15 diningroom.
}

```

The program produced above is simply a by-hand transliteration of information found in the script. With it, however, we can use proof search to generate the concurrent structure of the play.

The raw proof term generated by a query or trace on this program is always concurrently equal to the following:

```

let {[X2, [X3, [X4, [X5, [X6, [X7, [X8, [X9, [X10, X11]]]]]]]]]} = X1 in
let {X12} = a1_carlotta_enters X9 in
let {X13} = a1_dannunzio_enters X8 in
let {X14} = a1_mario_enters X11 in
let {X15} = a1_emilia_enters X4 in
let {X16} = a1_dannunzio_exits X13 in
let {[X17, X18]} = a1_dante_finzi [X10, X5] in
let {X19} = a1_aelis_enters X7 in
let {X20} = a1_emilia_exits X15 in
let {[X21, [X22, X23]]} = a1_groups_split [X17, [X18, X6]] in
let {X24} = a1_mario_exits X14 in
let {X25} = b2_despiga_monologue X23 in
let {[X26, X27]} = a1_carlotta_and_aelis_exit [X19, X12] in
let {[X28, X29]} = a1_grp1_luisa [X21, X3] in
let {X30} = b3_luisa_monologue X29 in
let {[X31, X32]} = a1_grp1_dannunzio [X28, X16] in
let {[X33, X34]} = b10_dante_despiga [X31, X25] in
let {X35} = b6_dannunzio_monologue X32 in
let {[X36, X37]} = b11_dannunzio_luisa [X35, X30] in
let {[X38, X39]} = a1_grp2_aelis [X22, X27] in
let {X40} = b9_aelis_monologue X39 in
let {[X41, X42]} = a1_grp2_emilia [X38, X20] in
let {X43} = b7_emilia_monologue X42 in
let {[X44, X45]} = b4_carlotta_kisses_finzi [X41, X26] in
let {[X46, X47]} = b8_emilia_carlotta [X43, X45] in
let {X48} = b5_finzi_monologue X44 in
let {[X49, [X50, [X51, [X52, [X53, X54]]]]]}
  = c12_finzi_exit [X34, [X33, [X40, [X48, [X47, X37]]]] in
let {[X55, [X56, X57]]} = c12_gunshot [X54, [X52, X46]] in
let {[X58, [X59, [X60, X61]]]}
  = c12_emilia_carlotta_aelis_luisa_leave_atrium
    [X57, [X53, [X51, X55]] in
let {[X62, X63]} = c13_finzi_luisa_confrontation [X56, X61] in
let {[X64, [X65, [X66, X67]]]}
  = c13_emilia_aelis_carlotta_luisa_sidehall [X58, [X60, [X59, X63]] in
let {[X68, [X69, [X70, X71]]]}
  = c12_take_presents_to_leda [X50, [X66, [X67, X64]] in
let {[X72, [X73, [X74, X75]]]}
  = c14_emilia_enter_leda [X68, [X69, [X70, X71]] in
let {[X76, [X77, [X78, X79]]]}
  = c14_all_exit_leda [X72, [X73, [X74, X75]] in
let {[X80, [X81, [X82, [X83, [X84, [X85, X86]]]]]]]}
  = c_final [X78, [X77, [X76, [X79, [X65, [X36, X49]]]]]]

```

The visualized concurrent structure of this term is given in Figures 5.10 and 5.11.

Figure 5.10: Trace graph for *Tamara* encoding, part 1.

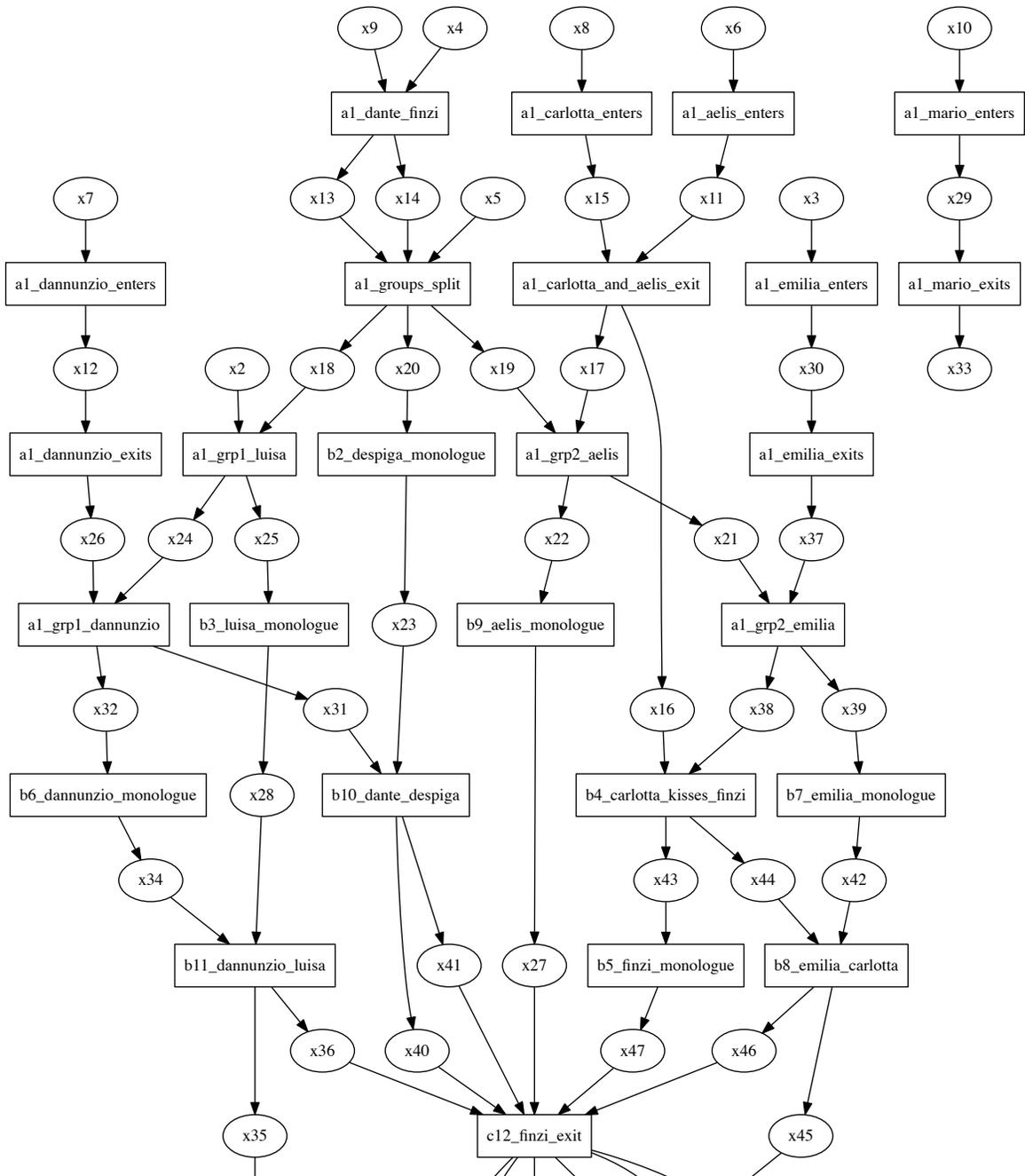
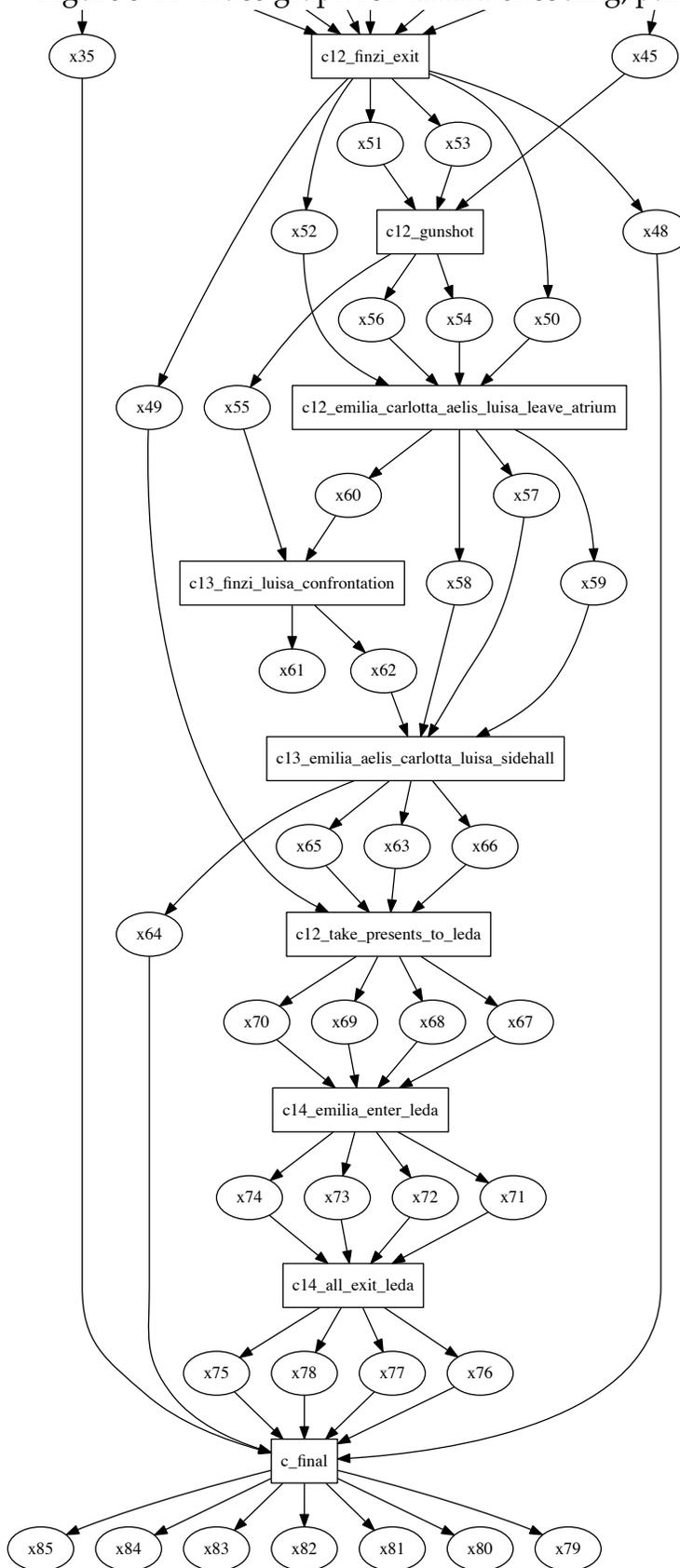


Figure 5.11: Trace graph for *Tamara* encoding, part 2.



In Celf, such a trace might result from the following query:

```
init -o
{ tamara      STamara  LTamara
* luisa       SLuisa   LLuisa
* emilia      SEmilia  LEmilia
* finzi       SFinzi   LFinzi
* despiga    SDeSpiga LDeSpiga
* aelis       SAelis   LAelis
* dannunzio  SDannunzio LDannunzio
* carlotta   SCarlotta LCarlotta
* dante      SDante    LDante
* mario      SMario    LMario}.
```

This query would additionally generate the final locations and scenes of every character:

```
#LAelis = oratorio
#LCarlotta = oratorio
#LDannunzio = atrium
#LDante = atrium
#LDeSpiga = oratorio
#LEmilia = diningroom
#LFinzi = atrium
#LLuisa = oratorio
#LMario = atrium
#LTamara = atrium

#SAelis = d17
#SCarlotta = d17
#SDannunzio = d15
#SDante = d15
#SDeSpiga = d17
#SEmilia = d15
#SFinzi = d15
#SLuisa = d17
#SMario = d15
#STamara = d15
```

5.5.3 Compilation to Twine

We next turn this structured trace, the CLF proof term, into a *playable drama* with the same form of interaction as the play itself: following (or staying with) characters. Doing so involves a very simple trick: we re-interpret the *simultaneous* branching in the story graph as *alternative* branching. That is, every out-resource of a scene can be interpreted as a choice to “follow” that resource to wherever it is needed next.

To realize this idea, we use the *Quiescent Theater* project described in Chapter 3. To recapitulate the idea of the project: we wrote a compiler from linear logic proof terms

to runnable Twine source code (also known as Twee). To make the output intelligible, we also introduced *scene files*, or mappings from linear logic program rules onto legible text, as follows:

Tamara

By John Krizanc, adapted by Chris Martens.

:: initial 2

Dante and Finzi welcome in the guests. "This is the home of Gabriele d'Annunzio,"
Dante informs you.

:: a1_dante_finzi 2

Dante and Finzi enter the atrium.

:: a1_emilia_enters 1

Emilia enters the atrium.

:: a1_dannunzio_enters 1

D'Annunzio enters the atrium.

:: a1_dannunzio_exits 1

D'Annunzio departs the atrium for his bedroom.

:: a1_emilia_exits 1

Emilia departs the atrium for the side hall.

:: a1_carlotta_enters 1

Carlotta enters the atrium.

:: a1_aelis_enters 1

Aelis enters the atrium.

:: a1_carlotta_and_aelis_exit 2

Carlotta exits toward the hall, and Aelis toward the dining room.

:: a1_mario_enters 1

Mario enters the atrium.

:: a1_mario_exits 1

Mario leaves the atrium.

:: a1_groups_split 3

Dante: "I'm going to go check on Signora Baccara."

Finzi: "I will visit the servants of the house, Aelis and Emilia."

Dante departs for Luisa's room, Finzi for the dining hall, and De Spiga
remains in the atrium.

```
:: a1_grp1_luisa 2
```

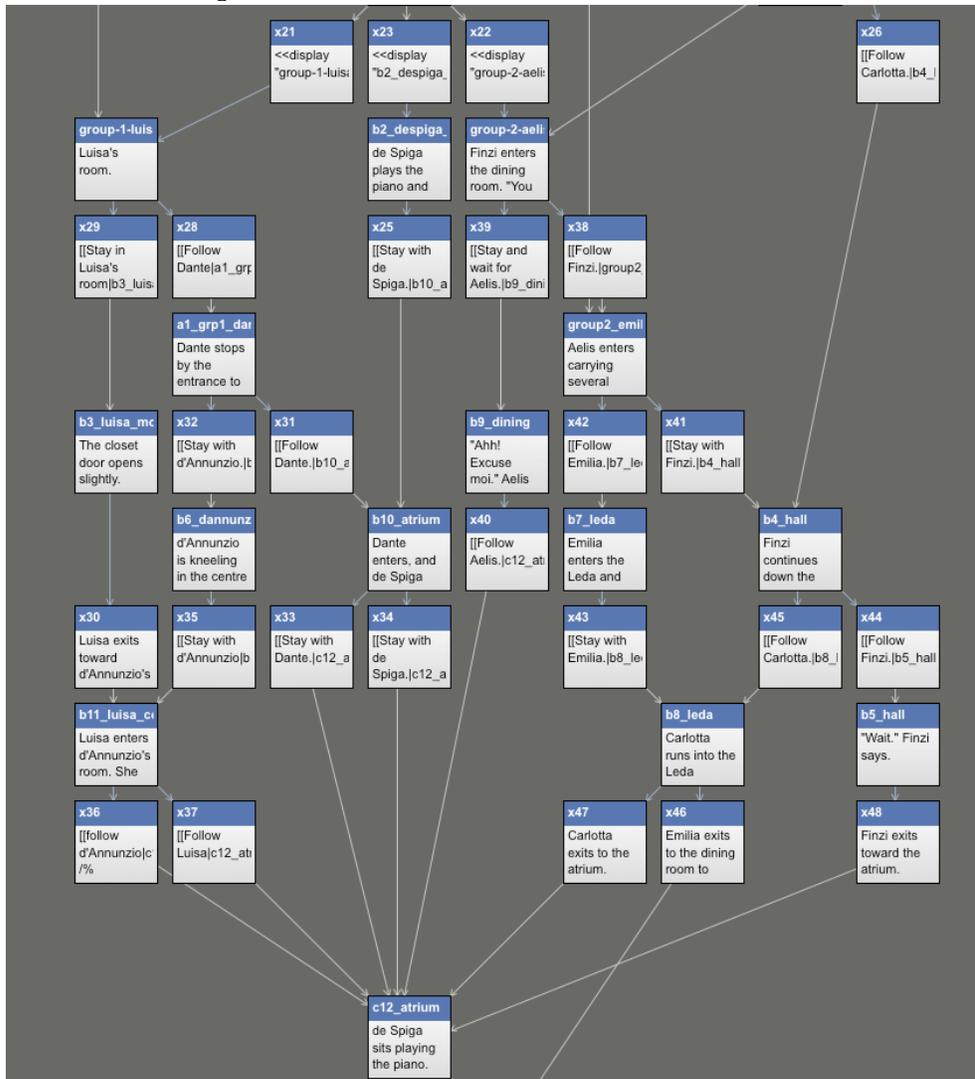
Dante enters Luisa's room. "Signora Baccara?" She is nowhere to be seen.
Dante departs Luisa's room for D'Annunzio's room.

...and so on. The number following the rule name (e.g. 3 in `::a1_groups_split 3` signifies the number of "followable" resources (characters) in the scene.

The remaining scene rules can be found on the GitHub page linked above. Again, these formal entities are simply transliterations of Krizanc's script, encoding complementary information to the linear logic program rules.

The compiler then works as follows: taking the linear logic program and scene file as inputs, it runs the logic program to produce a structured proof, parses that proof into the Twee skeleton, and instantiates the passages with text and links as specified by the scene file.

The resulting Twine game for Tamara has the same essential structure as the CLF proof term. A sample of it is shown below:



It can be played at <http://play.typesafety.net/world/tamara/>.

This project was joint work with Rob Simmons. We ran our compiler on a variety of linear logic programs and scene files, several of which were nondeterministic, unlike Tamara. The results can be found at <http://play.typesafety.net>.

5.5.4 Discussion

We have described a computational study of the theatrical play, *Tamara*, using linear logic to realize its simultaneous structure. We used linear logic as an intermediate language to transform the script of the play into a playable digital format, simulating the experience of an audience member.

Participatory theater has seen a surge in popularity and innovation in recent years, notably including the theatre company Punch Drunk and its acclaimed work *Sleep No More*, an immersive “installation art” performance [Pep11]. We imagine that authoring these works in such a way to guarantee *implementability* is rather difficult, especially in the presence of audience interaction that may chance the course of the story. We propose that computational tools might broaden participation in the authoring of such complex theatrical productions, perhaps making it more inviting to artists in game design as well. We also suggest that such tools could be an aid to producers, directors, and stage managers of such performances, allowing them to extract important dependency information between scenes without computing it by hand.

5.6 Discussion of Case Studies

We have presented five case studies, giving what we estimate to be a representative sample of the range of expressiveness for game and story mechanics in linear logic. We now discuss the strengths and limitations of linear logic programming for the domains we have studied.

First, there are a few common patterns in games that we have *not* illustrated in the examples above, but which would be feasible to build. The first of these is *endings* in games—so far, we have mainly discussed win-condition-free, open sandbox worlds (with the one exception of Tamara, which has a single deterministic ending). A common class of games with important “win state” detection is puzzle games, often stratified into levels where winning one level is a requirement for advancing to the next.

For an example, take Sokoban, the 2d tile-based crate-pushing game. Sokoban levels typically have *targets* on which all crates must be present to advance the level. Here is how we could codify that win condition check in Ceptre:

```
target tile : pred.  
crate_at tile : pred.  
  
uncounted_target tile : pred.  
maybe_win : pred.  
no_win : pred.
```

```

stage gen_uncounted = {
    target T -o uncounted_target T.
}
qui * stage gen_uncounted -o stage count.

stage count = {
    uncounted_target T * $crate_at T -o target T.
}
qui * stage count -o stage check * maybe_win.

stage check = {
    uncounted_target T * maybe_win -o target T * no_win.
}

win : pred.
qui * stage check * no_win -o stage play.
qui * stage check * maybe_win -o win * stage win.

stage play = {
    % ...
}

stage win = {
    win * level L * next_level L L' -o level L'.
}
% ...

t1 : tile.
t2 : tile.
t3 : tile.
t4 : tile.
context test = {
    target t1,
    target t3,
    crate_at t1,
    crate_at t2,
    crate_at t3}.

#trace _ gen_uncounted test.

```

While the idea of *level stratification* is itself not something we have covered with these examples, it should be clear by now how one could accomplish it with stages.

The other very common type of mechanic we have not discussed is the passage of time independent from player actions, or relatedly, asynchronous player input. Both of these concepts can be implemented as sensing predicates.

5.6.1 Limitations

There are a few common programming patterns that linear logic programming is *not* well-suited for, and we want to state these limitations to clarify what should be considered out of scope for the programming language in its current implementation (and that other systems do better).

- **Negation.** Any rule can check for the *presence* of an atom in the world state, but cannot check for its absence. As noted in Section 4.3.2, we can codify this behavior with stages, but it is often too cumbersome to do so in practice.

The reason why negation is left out of the basic logical formalism is that it violates the critical *frame property* of the logic: whenever it is the case that $\Delta \rightarrow \Delta'$ is a valid transition, it should also be true that $\Delta, A \rightarrow \Delta', A$ is a valid transition. (One reason that this property is so important is that it lets us understand rules with respect to an “open world,” i.e. any arbitrary context.) But if transitions can be contingent on the *absence* of a resource A , then this property no longer holds. Simmons and Toninho [ST12] suggest a logically sound basis for introducing negation under certain conditions that preserve this principle, but investigating the incorporation of those ideas into an implementation of Ceptre is left to future work.

- **Symmetric predicates.** Predicates ranging over two terms often want to be symmetric, i.e. whenever $p(A, B)$, $p(B, A)$ also holds (especially in social story worlds). If we want this to be the case, we have to manage this invariant by hand.
- **Boolean or exclusive predicates.** Many predicates have a usage pattern in which there is never more than one copy, or in which they are mutually exclusive with another predicate to represent a Boolean state, or in which there is always exactly one term X for which $p(X)$. These patterns again must obey invariants that are maintained by the programmer by hand.
- **Complex geometric calculations.** Our framework would probably be ill-suited to modeling games involving rapid computation of convex hulls or three-dimensional collision boundaries, unless those computations were encapsulated as sensing and acting predicates (i.e. programmed outside the confines of Ceptre).
- **Time sensitivity.** We do not yet have any data collected about whether Ceptre programs would be able to meet “frames-per-second” benchmarks for time-sensitive games. In general, we do not imagine Ceptre as the engine driving a released, production-quality commercial game—we envision it as a prototyping language, one part of a designer’s toolbox out of many.

5.6.2 Strengths

Linear logic programming has strengths as a prototyping tool that together position it as uniquely well-suited for game design prototyping. We believe these strengths lie in the modeling of game economies, multi-agent systems, generative methods, and interactive storytelling.

- **Game economies.** Linear logic programming shares with the Machinations [Dor11] approach to simulating game mechanics its affordances for describing *game economies*, or the resource exchanges that take place during player and automatic actions. This strength is exemplified by DungeonWorld, GardenWorld, and SiedlerWorld.
- **Multi-agent systems.** The use of first-order logic enables us additionally to specify complex multi-agent systems where each agent is privy to the same rules, as exemplified in BuffyWorld.
- **Interactive storytelling.** Both BuffyWorld and Tamara exemplify the expressiveness of linear logic for codifying interactive story mechanics that involve the coherent coordination of multiple actors, and the Quiescent Theater project shows how it can be used to generate structured, interactive stories.
- **Generative methods.** Finally, we argue that linear logic programming has great potential for modeling and experimenting with *generative methods* in games, based on the demonstrations in GardenWorld and BuffyWorld of generating plant life and television horror stories in collaboration with the player.

5.6.3 Potential

Some of the limitations of the system are, of course, only due to the bounded time and scope of this dissertation. Here we suggest high water-marks in future developments of and with linear logic-based programming methodologies. Two ideas in particular that we believe they can enable are:

Comparing Worlds

With the ability to express such a wide range of systems using a small number of programming constructs comes the ability to *compare* structure across domains. For instance, we notice that the program structure for the act/react architecture in BuffyWorld closely resembles that in GardenWorld; and the positive feedback loop pattern in GardenWorld itself resembles that in DungeonWorld. The desire to compare games structurally seems well-expressed by interest in *game design patterns* [BH04], but previous work on such patterns is typically restricted not only by operational logic but by specific formulations of game entities. With linear logical representations, we have a candidate for the formal expression of design patterns across distinct operational logics.

Combining Worlds

Along similar lines, being able to express distinct operational logics in the same logical framework could enable us to invent new game designs via a “mash-up” artistic approach, i.e. recombination. A possible future study could involve taking any pair of case studies in this chapter and exploring different ways of combining them, either by mapping the predicates from one domain into predicates of the other, or simply taking

the union of the available player actions and state space, then adding new ways for those state spaces to interact.

Articulating Strategies

We have only very briefly, in Section 5.2, discussed possibilities for augmenting Ceptre programs with certain “AI” strategies. In general, such augmentations could be used for a variety of purposes, such as playtesting, implementing adversarial NPCs, or in-game guidance (see Treanor et al. [TZE⁺15] for a recent survey of uses of AI techniques in game design). However, many such techniques depend on probabilistic sampling of a search space in order to adapt to the human player’s techniques over time. Currently, Ceptre’s implementation of randomness does not offer any guarantees about the distribution or offer any means of rule selection based on history. In order to maximize control over the programming of such agents, something like stochastic logic programming [Cus00] could potentially be integrated for modeling probability distributions and sampling.

5.6.4 Analyzing Game Dynamics

After articulating a set of mechanics (and optionally a particular player strategy), we may be interested in some emergent properties of the game rules. For instance, in a two-player, turn-based game, we can formalize a simple notion of *balance* as whether the first or second player in the game has a significant advantage, on average. Other game mechanic description languages such as Machinations [Dor11] include facilities for producing resource graphs and other summarization data over multiple (interactive or simulated) play-throughs of a game. cursory investigations suggest that similar tools for Ceptre would be easy to build and helpful for our stated goals of aiding iteration and analysis of games.